Version 0

Scripts were written independently of each other, which creates tons of logical errors so that the system was not doing anything meaningful or even crashed. To make the system run, several fundamental things have to be consistent among all scripts: board representation, player representation, and board-to-cell mapping.

- **Representations**

In this AlphaZero for NoGo, four essentially related but different representations can be found; each is a one-to-one and onto mapping of the other three:

(1) the 1-D Go board with border points,
(2) the 2-D Go board,
(3) the 2-D canonical board that will be fed into the neural networks, and
(4) the 1-D policy vector $\boldsymbol{\pi}$ across all the points on the board.

All these four representations essentially give different numbering of the same points, but the rule of mapping has to be made clear and consistent.

- **Experiment results**

Version 0 may not crash, but it did not give the expected result.

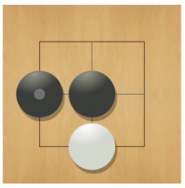

Fig. 1: a 3x3 Go board example

A3	B3	C3
A2	B2	C2
A1	B1	C1

Fig. 3: Numbering of 2-D board

0	0	0
1	1	0
0	2	0

Fig. 4: 2-D board representation

0	0	0
-1	-1	0
0	1	0

Fig. 5: 2-D canonical board representation

16	17	18	19	20
12	13	14	15	
08	09	10	11	
04	05	06	07	
00	01	02	03	

[3,3,3,3, 3,0,2,0, 3,1,1,0, 3,0,0,0, 3,3,3,3,3]

Fig. 2: Numbering of 1-D board with border points and 1-D board representation

7	8	9
4	5	6
1	2	3

$\boldsymbol{\pi} = [\pi 1, \pi 2, \pi 3, \pi 4, \pi 5, \pi 6, \pi 7, \pi 8, \pi 9]$, where $\pi 2 = \pi 4 = \pi 5 = 0$

Fig. 6: Numbering of 1-D policy vector $\boldsymbol{\pi}$

Version 1

Version 1 focused on fixing bugs in game-specific APIs and MCTS. The functions of these APIs also heavily rely on the consistency of board representation. Most of problems occurred in the MCTS algorithm.

- **Game-specific APIs**

These APIs should be implemented according to the representations from Version 0.

- **MCTS and in-search move selection**

The original design was to let the system learn on the fly: when an invalid move was selected by PUCT, the system masked this move out with 0 in the policy vector and valid moves vector, then continuing the search without disruption. However, the handling of tryouts of moves seems to be problematic. The try clause where the play command resides broke the program flow, making the search consume more recourse. It also introduced instability that could cause the search to go infinitely deep recursively until python aborts. To fix those issues, some new code has to be deployed for checking states, but they were hard to understand and very expensive, slowing down the whole search process dramatically. For simplicity and efficiency, the system eventually uses get-valid-moves API and do PUCT only among valid moves.

- **Endgame result**

Backing up the endgame result is also subject to error in the first implementation. Checking the returned value of the terminal state is especially important. Printing the selected moves and final policy on the screen is good for debugging.

- **Experiment results**

Version 1 ran without errors, but the win rate against UCB player was stuck at 20%.

Version 2

In version 2, a new script was added solely for self-play. Self-play was originally embedded in the Coach script, but a stand-alone self-play script is much easier for understanding, debugging, and, most importantly, multiprocessing.

- **Self-play game records**

It is crucial that all self-play game records being generated are correct. These game records as input will be fed into the neural networks. Garbage in garbage out. Checking correctness involves checking the value and symmetries for each state. A stand-alone script provides the convenience to generate and return sample game records for human inspection. In particular, the records should be checked for symmetries: the board must match the policy still after rotating or flipping the two. Since game APIs are needed to get symmetries, it is a good time to once again check board representations to make sure that rotation and flipping are working correctly.

- **Multiprocessing**

Majority of the PCs today have at least a 6-core CPU and 8 GB of memory, so it is very compelling to harness the full power of the hardware and do multiprocessing. Each of the self-

play games or arena games is an independent game and suitable for multiprocessing. In training 7x7 NoGo, multiprocessing with 4 processes can reduce the time needed for 400 self-play games *to* less than 22% on Cirrus. The number of parallel processes that can be initialized depends on the size of the model. Each process will create a copy of the model for its own inference, so the bottleneck is likely to be the GPU memory size. If all code is correct but the system throws an error, then this error may be caused by running out of GPU memory and may be fixed by reducing the number of processes.

The Arena script was modified to support multiprocessing as well in Version 2.

- **Experiment results**

Version 2 is the first working system that can produce a very strong player as shown in Fig. 7. It turns out that the low win rate of Version 1 is caused by the incorrect symmetries mechanism. After fixing this bug, Version 2 reaches a win rate of 79% against UCB player in 5x5 games. However, the win rate flattens out and even starts to decrease after reaching the peak.

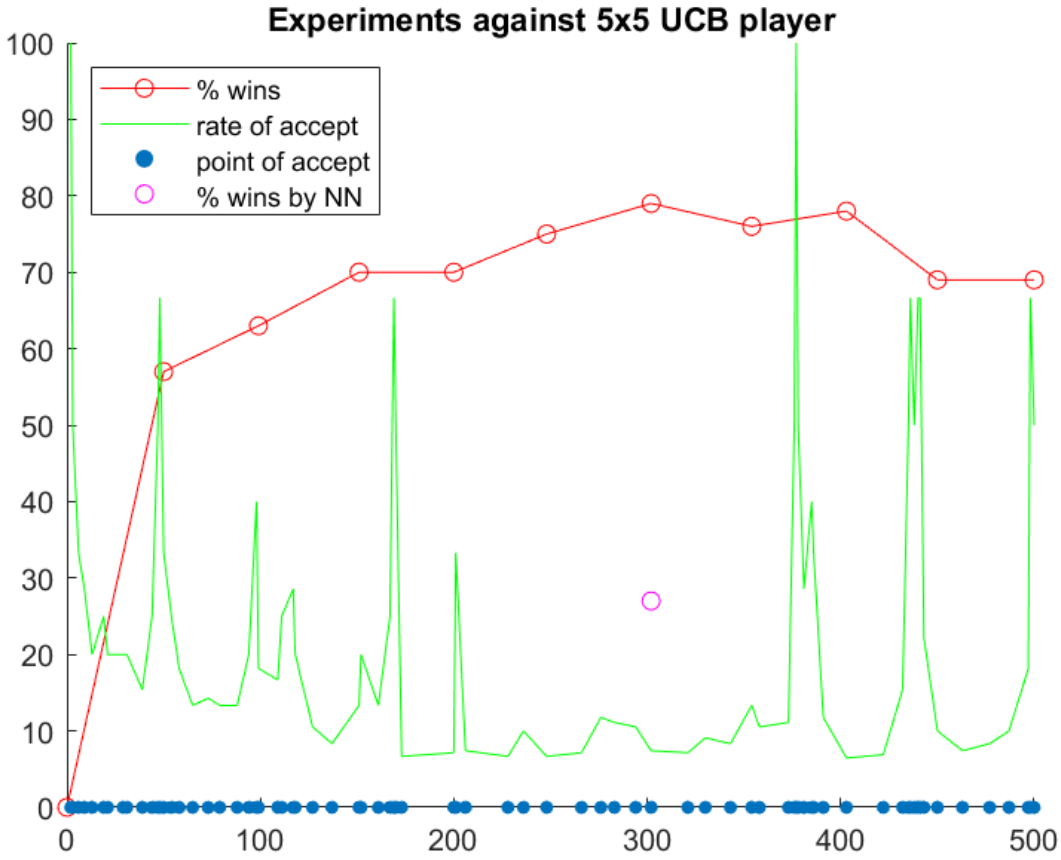

Fig. 7: Experiment results of AlphaZero Version 2 against UCB player on 5x5 NoGo board. The x-axis indicates the number of iterations of AlphaZero. The experiment settings were: (1) both players had the same simulation budget – 100 Sims/valid move (2) the search tree in AlphaZero was cleared between its turns to play.

Version 2.5

Version 2.5 adds the feature of attaching the program to GoGui for visual analysis. This version is worth being documented because it really shows the importance of retrieving and analyzing game records for further debugging or improvement.

- **Interface for GoGui**

If playing only the NN policy without MCTS, Version 2 gives a win rate of 27%, which is significantly lower than expected. However, only the experiment results were preserved, and the system did not collect any generated policy at each step. No further analysis but merely speculations could be proposed.

For the sake of easy and detailed analysis, an interface of the system for GoGui was implemented. After being attached to GoGui, the system can, at each step, print out the NN and MCTS policies which can then be visually compared with current board and human intuition.

During the tests on GoGui, a weird phenomenon was observed. The NN policy was not symmetric: the two symmetric points w.r.t. the current board at that state had two different values produced by NN. Since all symmetries bugs had been fixed, this type of error should not exist. This implied that there was error at a "lower" level. It eventually turned out to be an error in 2D board representation which was taken for granted at the beginning.

Version 3

The main improvement of Version 3 is adding two features: ResNet and Dirichlet noise.

- **ResNet**

ResNet is used so that the neural networks can learn the training examples more accurately. It also prevents the model from overfitting. The difference between the original architecture and the new one is shown in Fig. 8 and 9.

- **Dirichlet noise**

Dirichlet noise is added to the policy π during self-play games to facilitate more explorations and, hopefully, mitigate the degradation issue. Fig. 7 shows some degradation in AlphaZero towards the end of the training process, which contradicts what Silver et al claimed in their AlphaGo Zero paper.

- **Experiment results**

Version 3 can produce very strong players. However, the degradation issue can still be observed in Fig. 10 and 11.

- **Concluding remarks**

The performance of AlphaZero player also depends on the properties of a game. For those games with more obscure strategic patterns or hard to be represented or comprehended in nature, it will be much harder for neural networks to learn the features in those games, hence less strong.

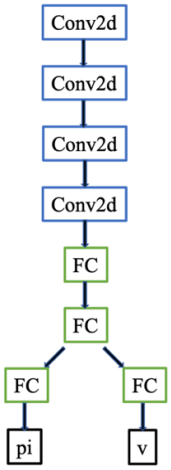

Fig. 8: Original architecture

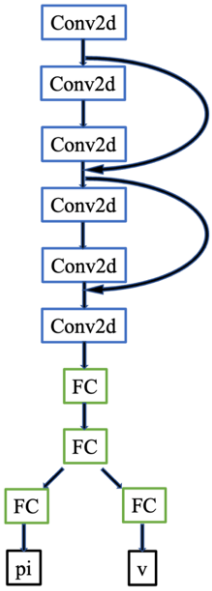

Fig. 9: New architecture

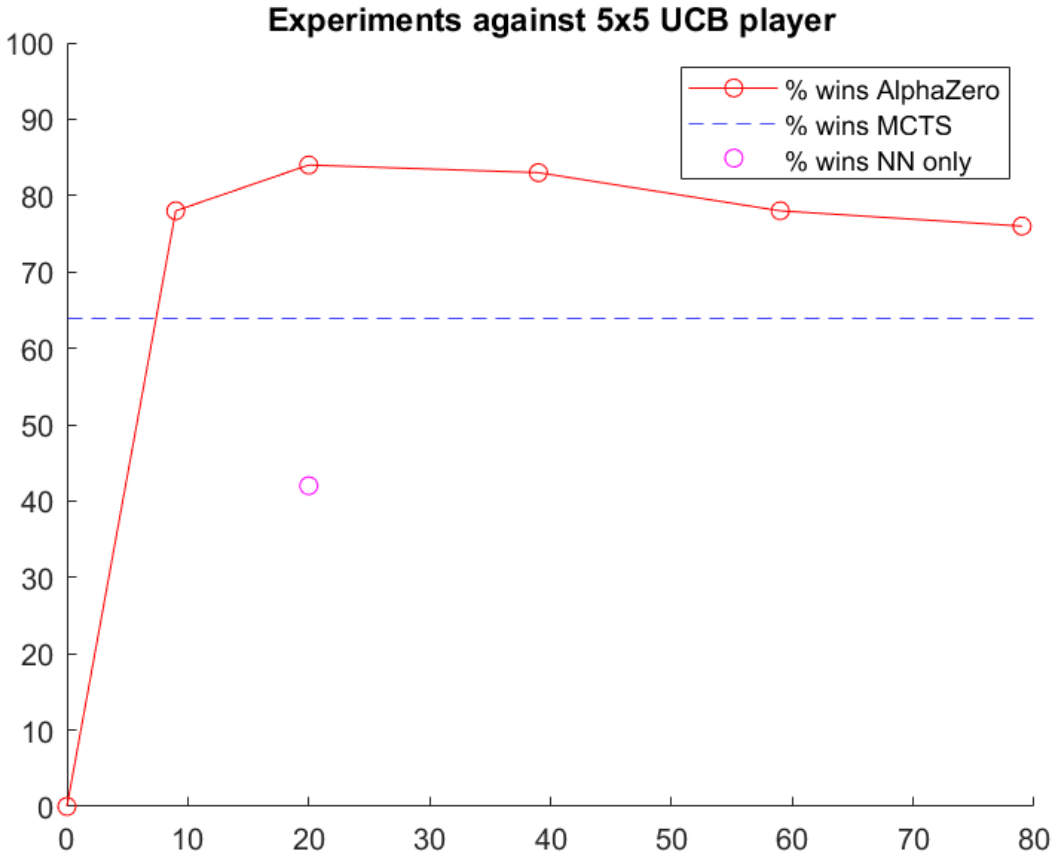

Fig. 10: Experiment results of AlphaZero Version 3 against UCB player on 5x5 NoGo board. The x-axis indicates the number of iterations of AlphaZero. The experiment settings were: (1) both players had the same simulation budget – 100 Sims/valid move (2) the search tree in AlphaZero was cleared between its turns to play.

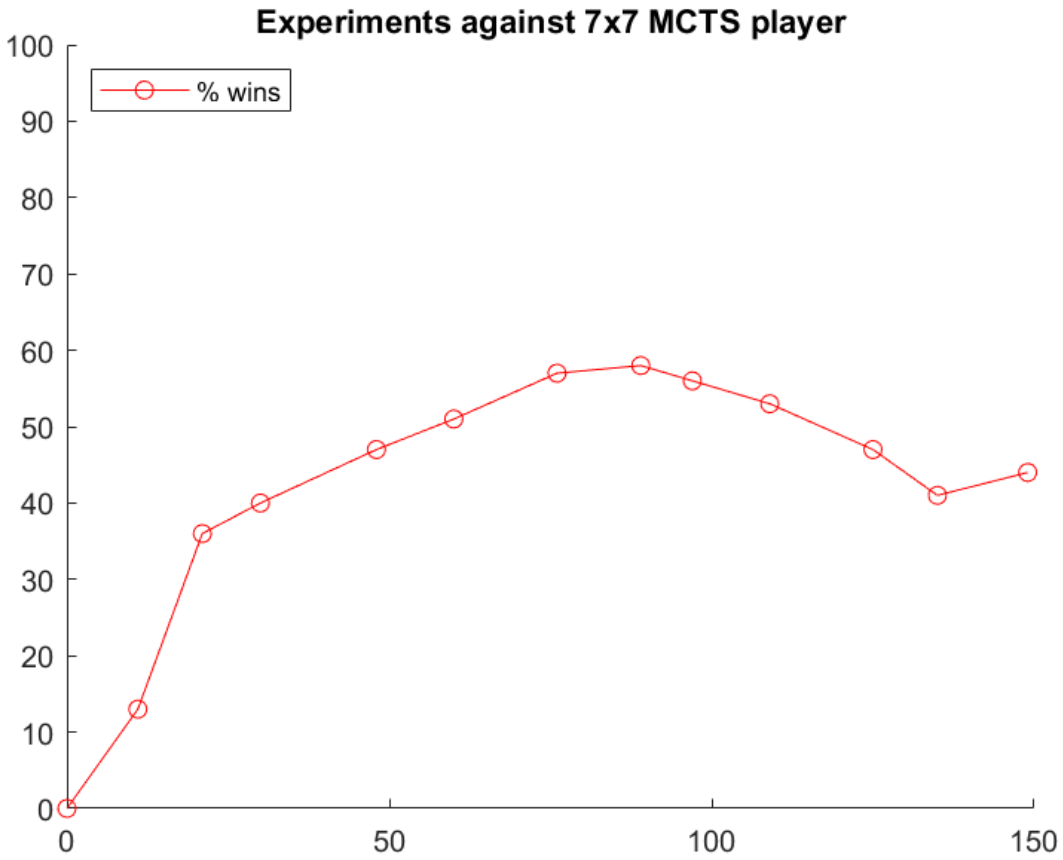

Fig. 11: Experiment results of AlphaZero Version 3 against MCTS player on 7x7 NoGo board. The x-axis indicates the number of iterations of AlphaZero. The experiment settings were: (1) both players had the same simulation budget – 100 Sims/valid move (2) the search tree in AlphaZero was cleared between its turns to play.