A Tutorial on AlphaGo Zero

Zhihan Yang
Carleton College
Email: zhihanyang2000@gmail.com

August 15, 2021

AlphaGo Zero is a ground-breaking algorithms that can learn to play Go at super-human level from scratch through self-play. In this tutorial, we seek the source of its magic by connecting it to previous Computer Go research, including pivotal insights from AlphaGo and earlier work. This tutorial also intends to be self-contained; it begins with a tutorial on Monte Carlo Tree Search (MCTS), the algorithmic framework behind all popular Go programs. Finally, we implement and analyze AlphaConnect4 Zero, a competent Connect4 program trained using the AlphaGo Zero framework. Code is available at github.com/zhihanyang2022/alpha-zero.

Table of contents

1. Monte Carlo Tree Search

In this section, we focus on a classic algorithm for doing Monte Carlo Tree Search (MCTS), called the Upper Confidence Bound for Trees (UCT), in the context of perfect-information, deterministic and zero-sum board games. Historically, UCT was an early breakthrough in MCTS because it converges to the Minimax solution as the number of simulations goes to infinity[1]. To better understand this, we will first take a look at the Minimax algorithm, the theoretically perfect planning strategy for such board games.

1. Browne, Cameron B., et al. "A survey of monte carlo tree search methods." *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012): 1-43.

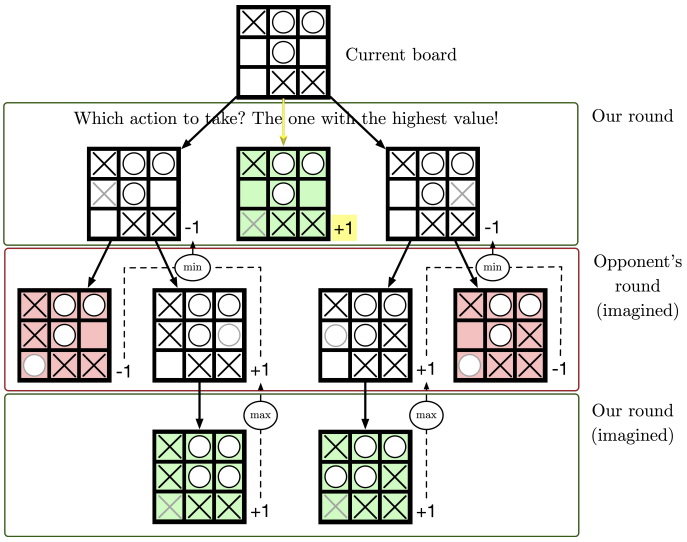


Figure 1. Minimax example using tic-tac-toe.

1.1. Minimax

1.1.1. Goal

In theory, the Minimax is the perfect planning strategy for perfect-information, deterministic, zero-sum board games. Given a board arrangement or board state, it computes and outputs the best move to pick. At this point, the exact interpretation of "perfect" and "best" is unclear because, intuitively, the best move should be opponent-dependent. However, as we will see, this is not the case in the context of perfect-information, deterministic, zero-sum board games; the best move against the strongest opponent is the best move against any opponent.

1.1.2. Algorithm

The best way to illustrate the Minimax algorithm is through a tic-tac-toe example (Fig 1). Suppose you hold the "X" stone and it's your round. Given the current board state, you need to select one of three possible moves. Starting from the current board state, Minimax builds out the game tree in its entirely, with each node representing a board state[2] and each leaf node representing a terminal board state. Each terminal state is rated by a reward, indicating how well the game ended from *your* perspective (e.g., a common choice is $+1$ for win, 0 for draw and -1 for loss). Then, Minimax backups the scalars from the deepest level to the shallowest level just below the current board state, taking the max in your round and taking the min in your opponents round. This makes sense because, intuitively, you are the *maximizer* of your reward and your opponent is the *minimizer* of your reward. Equivalently, one could modify Minimax to take the max in both your and your opponent's round but flip the reward in your opponent's round; this is called the Negmax variant of Minimax. The fact that this backup is greedy assumes that you and your opponent are optimal. Finally, Minimax picks the move that has to highest possible reward obtainable under this assumption, which is the safest assumption to make if the skill level of your opponent is unknown. For example, you might win by taking the leftmost or the rightmost move, but only when your opponent is random or intentionally bad at the game.

1.1.3. Implication for Go

Theoretically, Go can be solved perfectly using Minimax. However, in pratice, Go's branching factor is too large for Minimax to build out the game tree. To mitigate this problem, there are pruning variants of Minimax, but they are beyond the scope of this tutorial.

2. Not necessarily unique.

1.2. Upper Confidence Bound for Trees

Upper Confidence Bound for Trees, or any other member of the MCTS family, also builds out a game tree like Minimax. However, the tree is constructed incrementally and, although the nodes in this tree also represent game states, they store extra things. In this sub-section, we walk through three important questions. What do nodes and edges in the tree represent? How is the tree constructed from scratch? And most importantly, how to select a move afterwards?

1.2.1. What do nodes and edges in the tree represent?

Each node represents a game state, and each edge represents a valid move in the game state represented by its parent node. From this setup, it may seem like we need to store a game state in each node; this memory cost turns out to be unnecessary. Because the game is deterministic, we only need to store (a copy of; to prevent overwrite) the simulator with the state represented by the root node; the state associated with another non-root node can be obtained by traversing from the root node to that node, and sequentially executing (in the simulator) the actions represented by the edges encountered the way.

Additionally, a node stores a visit count n and a value sum Q, and Q/n (if $n=0$, then return 0) represents the state value of the state s' represented by this node. Since the game is deterministic, this value is also the action value of (s,a) where s is the state represented by the parent node and a is the action that leads from s to s'.

1.2.2. How is the tree constructed from scratch?

The UCT algorithm (Fig 2) loops over the following three steps until time or resource is exhausted:

1. **Selection and rollout**

   Follow the tree policy to traverse from the root node to a leaf node (when it is iteration 1, the root node is a leaf node); the tree policy is a UCB rule that selects the child node based on their value while taking into account of uncertainty:

   $$\max_j \left(\frac{Q_j}{n_j} + 2C\sqrt{\frac{2\ln n}{n_j}} \right) \text{ or } \max_a \left(\frac{Q(s,a)}{n(s,a)} + 2C\sqrt{\frac{2\ln n(s)}{n(s,a)}} \right)$$

   where n_j and Q_j are the visit count and value sum of the j-th child node, and $C>0$ is a hyperparameter.

   Apply all moves along the way sequentially (each edge represents a move) to get a simulator with the correct game state associated with the leaf node. *Follow the rollout policy in the simulator* untils the episode terminates; this generates a game outcome in the form a scalar.

2. **Backup**

   The scalar outcome is backed-up for all nodes in the tree encountered by the tree policy. In particular, for each of these nodes, the visit count n is incremented by 1 and the value sum Q is incremented by the scalar outcome. Note that the outcome will need to be flipped properly (i.e., multiplied by -1) at every level for a two-play game.

3. **Expand**

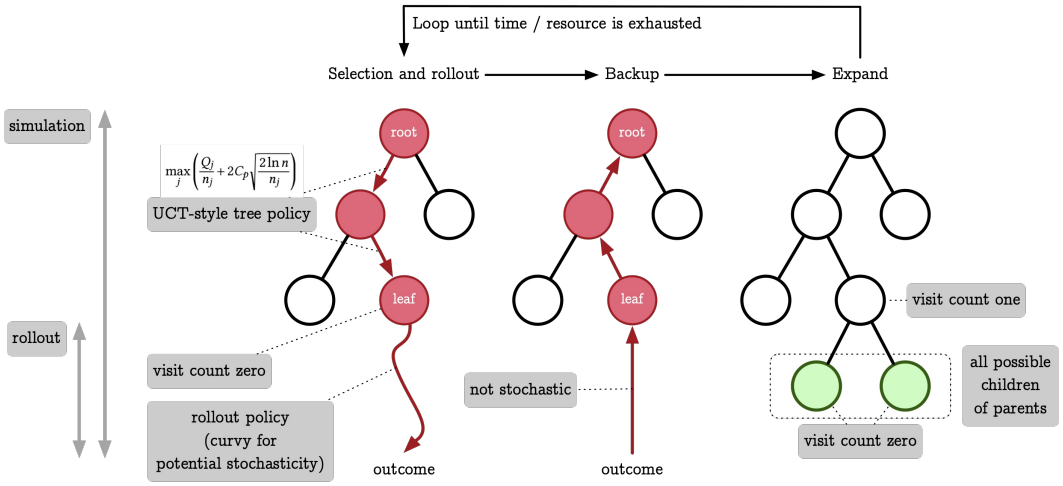


Figure. The UCT algorithm.

For each possible (move, next state) tuple from the state represented by the leaf node, we add a new edge and a new node below the left node. In future iterations, rollouts will be collected directly from these new nodes, until their visit counts become greater than 1.

1.2.3. How to select a move afterwards?

This question is best answered using pseudocode:

```
best_move = None
best_ucb_score = - inf
for (move, child_node) in root_node.children:  # suppose this is a dictionary
    if child_node.ucb_score > best_ucb_score:
        best_ucb_scre = child_node.ucb_score
        best_move = move
```

1.3. Prior Probabilities

We see that the rollout policy is uniformly random over valid moves and that the tree policy exploits moves that look better so far based on previously simulations, which involves random rollouts. Then, the natural question is whether we can incorporate the knowledge of some existing policy, which we shall call the *guiding* policy. It turns out that this can be done in two ways:

1. Use the guiding policy as the rollout policy.

2. Use the guiding policy p as part of the tree policy so that the move chosen a is computed by

$$\max_a \left(\frac{Q_j}{n_j} + 2C \cdot P_j \cdot \frac{\sqrt{n}}{n_j + 1} \right) \text{ or } \max_a \left(\frac{Q(s,a)}{n(s,a)} + 2C \cdot p(a|\, s) \cdot \frac{\sqrt{n(s)}}{n(s,a)+1} \right).$$

Generally speaking[3], the better the the guiding policy, the better the move proposed by MCTS.

3. We do not intend to prove this claim here.

1.4. Value Functions

The purpose of doing rollouts is to evaluate the value of a leaf node under the rollout policy. However, the outcome of a single rollout is a high-variance estimator, which can be exacerbated by longer games. Therefore, whenever possible, we should use the value of function of the rollout policy instead of using the outcome of a single rollout.

2. AlphaGo

AlphaGo is the first Go program to achieve super-human performance. However, from the paper itself, one learn that AlphaGo is similar to previous state-of-the-art (SOTA) Go programs, in the sense that both AlphaGo and other methods rely on MCTS guided by policies trained to predict human expert moves. The key contribution in AlphaGo is the use deep neural networks rather than shallow linear functions of handcrafted features to represent policies and value functions.

In sub-section 1.3 and 1.4, we learned three ways by which a guiding policy can be used; it turns out that all of them appear in AlphaGo:

- **Supervised-learning policy network (SL policy).** A very deep residual convolutional neural network trained to predict human expert move given raw board information.
- **Rollout policy.** A linear softmax network trained to prediction human expert move given local handcrafted values, which allows for fast inference.
- **Reinforcement-learning policy network (RL policy)**. The result of fine-tuning SL policy using policy gradient, which outperforms SL policy in one-vs-one matches. The training details is not the focus of this tutorial.
- **Value function of RL policy.** The training details is not the focus of this tutorial.

The AlphaGo team has found the following combination to be the most effective:

- **For tree policy.** Use SL policy to generate prior probabilities. RL policy was not used simply because it performed worse; the AlphaGo team postulated that human moves are more diverse[4], which helped SL policy to be more diverse.
- **For evaluation of leaf nodes.** Use a weighted sum of the outcome of a rollout using the rollout policy and the value function.

3. AlphaZero

AlphaZero is the second[5] Go program to achieve super-human performance without using human knowledge, and it also defeats AlphaGo consistently. AlphaZero can be inspired as follows. The success of AlphaGo largely rely on a large dataset of human expert games. Therefore, intuitively, in

4. This make sense, because human players have a diverse range of skill levels.

5. The first is AlphaGo Zero. However, AlphaGo Zero has some unnecessarily sophisticated details which have been later removed in AlphaZero. Therefore, for clarity, we skip AlphaGo Zero in this tutorial.

order to train an agent that's even better than AlphaGo, we need a large dataset of games played by players that are better than human experts. Where can we find such players? The answer is AlphaGo. We could repeat this process of data collection, supervised learning[6] and applying MCTS indefinitely, and we should be getting better and better Go players over time. This is in fact an accurate description of what AlphaZero does; the only difference is that AlphaZero starts with random players instead of human experts to showcase the possibility to learn from scratch.

This obvious shortcoming of the aforementioned approach that data collection via self-play is very costly; it is not possible to re-collect a large dataset every time the policy improves. But let's first assume that this is possible for simplicity. With the current MCTS policy[7], we generate many self-play episodes, in which each timestep is represented by the 3-tuple:

$$(\boldsymbol{s}, \boldsymbol{\pi}, z)$$

where $\boldsymbol{s}$ is the board state in the perspective of the current player, $\boldsymbol{\pi}$ is the a probability matrix returned by MCTS and z is episode outcome in the perspective of the current player. As a sidenode, AlphaZero uses two techniques to introduce diversity into the self-play process:

- **First 30 moves.** Suppose $\boldsymbol{s}$ is the current game state. After MCTS, let

$$\boldsymbol{\pi}(a) = \frac{N(s,a)^{1/\tau}}{\sum_{a'} N(s,a')^{1/\tau}}$$

  where τ is set to 1 to encourage exploration.

- **After the first 30 moves.** During this time, τ is set to a very small value, and hence

$$\boldsymbol{\pi}(a) \approx \begin{cases} 1 & \text{if } N(s,a) \geq N(s,a') \text{ for all } a' \in \mathcal{A} \\ 0 & \text{otherwise.} \end{cases}$$

  To encourage extra exploration, Dirichlet noise is injected to the positions in $\boldsymbol{\pi}$ that corresponds to valid moves via a weighted average between the original probabilities and the Dirichlet noise.

Using this dataset $\{(\boldsymbol{s}, \boldsymbol{\pi}, z)\}$, we can train the policy and value network from scratch via mini-batch supervised learning (Fig 3) using the following loss function:

$$J(\theta) = \sum_{i=1}^{M} \underbrace{(v_\theta(\boldsymbol{s}_i) - z_i)^2}_{\text{squared error}} \underbrace{-\boldsymbol{\pi}_i^\top \log p_\theta(\boldsymbol{s}_i)}_{\text{cross-entropy}} + c \underbrace{\|\theta\|_2}_{L_2}$$

where p_θ and v_θ are the policy and value network respectively, and they share parameters; $c > 0$ is a hyperparameter for L_2 regularization to prevent overfitting. Upon convergence, p_θ has become a better policy and v_θ has become the value function for the new p_θ; with their guidance, we also have a better MCTS policy, using which we can collect a new dataset and repeat this process.

6. Learn the SL policy just like in AlphaGo. AlphaZero does not have RL policy or rollout policy. Besides SL policy, it only learns the value function of the SL policy, as we will see in a bit.

7. As hinted by the footnode above, this MCTS policy is guided by SL policy and its value function trained on data collected from the previous MCTS policy. At iteration 1, the SL policy and its value function are initialized randomly.

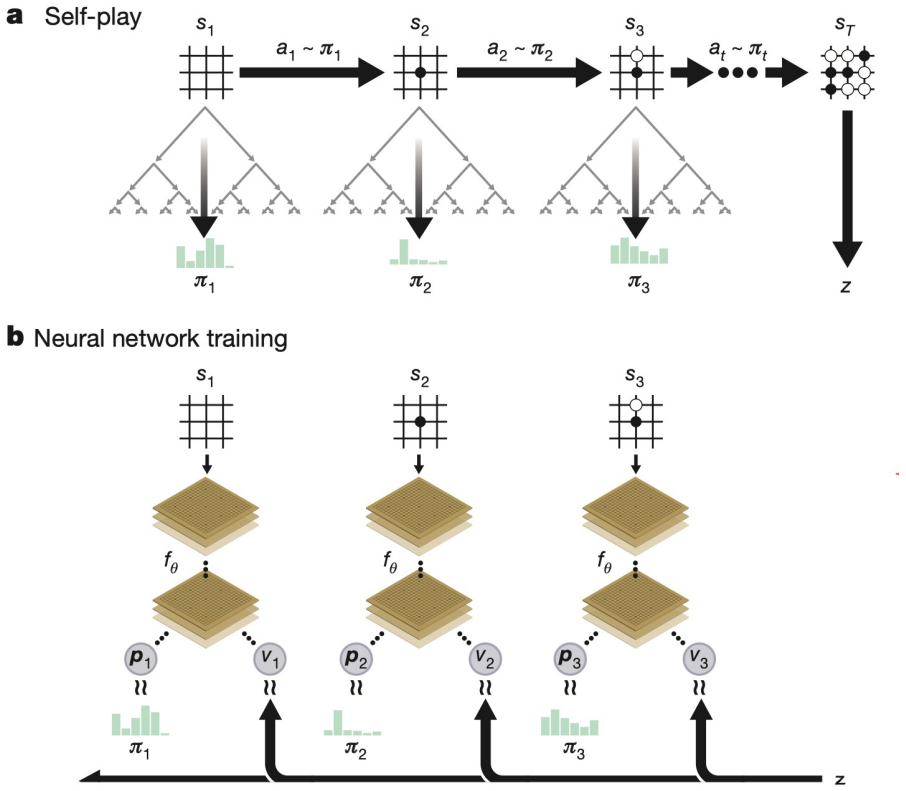


Figure 2. How AlphaZero is trained (screenshot from AlphaGo Zero paper).

However, the fact that self-play is costly prohibits the aforementioned approach. To resolve this issue, AlphaZero takes a *moving-average* approach. In each iteration only a few self-play episodes are generated and stored in a very large buffer that also contains episodes by previous MCTS policies, and a batch of episodes is randomly sampled from the buffer for neural network training. The neural network is also not trained from scratch; AlphaZero takes a few gradient steps (each with a different batch) on the neural network from the previous iteration (which guided the MCTS policy in this iteration of self-play) before collecting more episodes. In the AlphaZero paper, generating episodes and training seem to happen in parallel.

4. AlphaZeroConnect4

4.1. Connect4

Connect4 is a simple game on a 6x6 board (flat) where each player attempt to connect 4 stones of their own color. The connection can be vertical, horizontal or diagonal. When played optimally, the first-hand player is guaranteed to win and the second-hand player is guaranteed to lose.

4.2. Implementation and training details

The implementation is best understood by reading code in the open-source Github repository. For most parts, AlphaZeroConnect4 is exactly the same as AlphaZero described in the previous section.

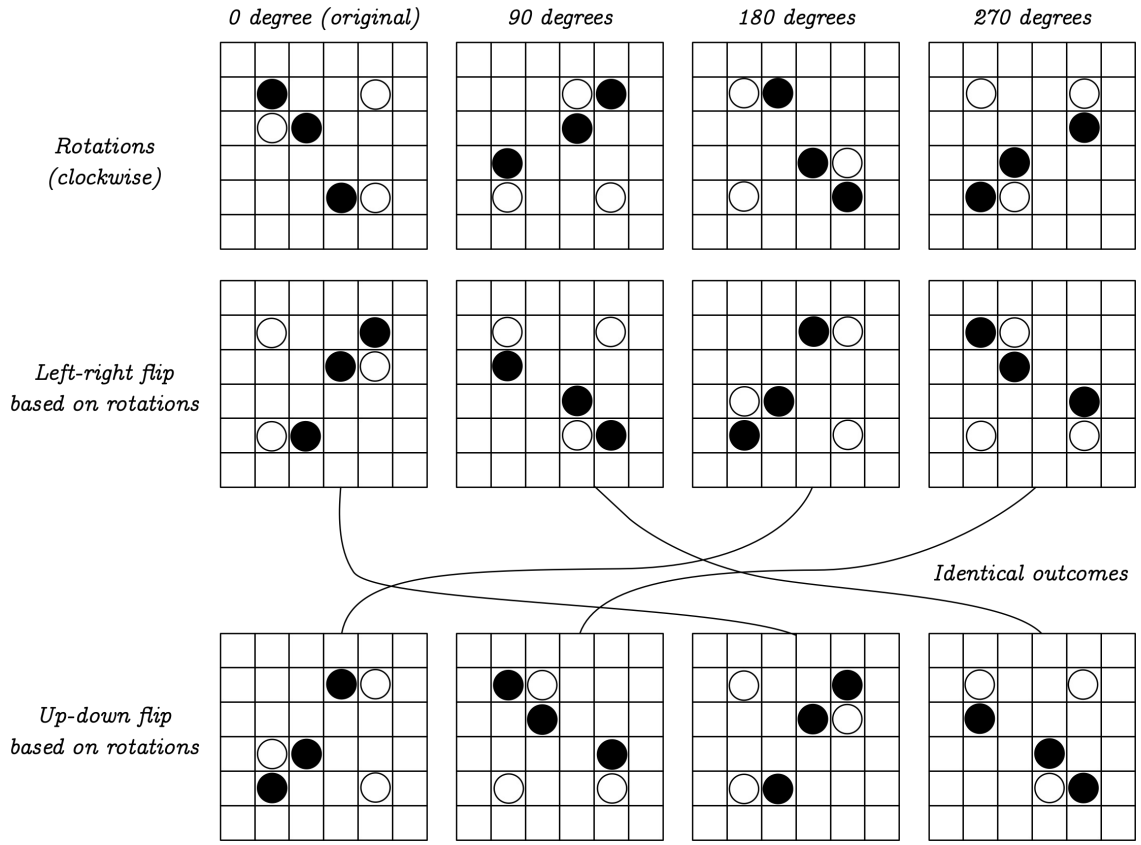


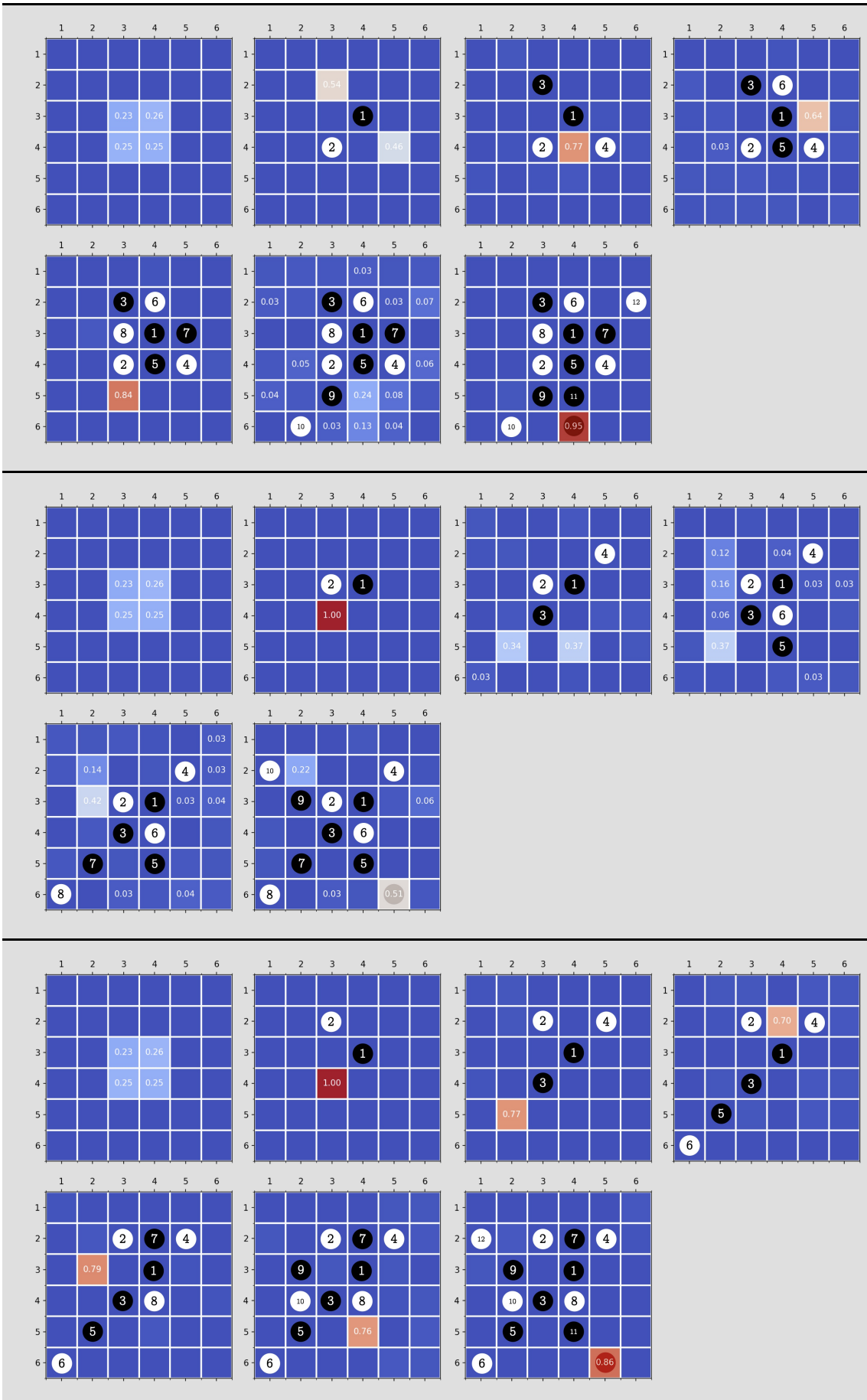
Figure 3. Eight unique geometric transformations of a game state.

However, because self-play is very time-consuming when written in Python, we incorporate data augmentation introduced in the AlphaGo Zero paper. More specifically, a game state is augmented as eight unique geometric transformations of itself (Fig 3), assuming that the game state does not itself contain some sort of symmetry. This turns each self-play game into eight, which drastically improves the efficiency of self-play.

4.3. Example first-hand game plays versus a human player

Below are three game plays between AlphaZeroConnect4 (AZC4) (black stone) and a human player (white stone). The first hand for all three games were AZC4 to demonstrate its ability to guarantee a win in different situations when offered the first hand. On the other hand, measuring its performance as the second hand is not straightforward because it is guaranteed to lose against a strong human player. Each board image shows all the stones that have been placed so far, and the prior probabilities predicted by the convolutional policy network for the AZC4's next move. It's important to note that the prior probabilities (along with the value function) is used to guide the MCTS[8] and the chosen move does not always match the move with the highest prior probability.

8. Ran for 50-1000 simulations (the exact number of randomly picked from this range) to induce stochasticity.


0.23 0.26
0.25 0.25
0.54
0.46
0.64
0.03
0.77
0.84
0.03 0.07
0.05
0.06
0.04
0.24 0.08
0.03 0.13 0.04
12
11
0.95
1.00
0.12 0.04
0.16 0.03 0.03
0.06
0.34 0.37
0.37
0.14 0.03
0.42 0.03 0.04
0.22
0.06
0.51
0.70
0.79
0.76
0.86