6.882 Final Project - AlphaGomoku

Zhongxia Yan

Abstract

Reliable labeled data is a frequent limitation to algorithms in artificial intelligence. In the context of solving multi-player board games, such as Gomoku, human labeled data is slow, relatively scarce, and often unreliable, as humans make mistakes. Moreover, relying too much on labeled data, either generated by humans or by another algorithm, may hamper an algorithm's ability to surpass the performance of the generator. Thus, it is desirable to create artificial intelligence that learns how to play a board game with only the rules of the game and zero prior knowledge. Here we re-implement a version of AlphaZero for the purpose of solving the board game Gomoku, also known as Five in a Row. Like AlphaZero, we utilize Monte-Carlo Tree Search aided by deep neural networks to evaluate positions and select moves, and train these deep neural networks via reinforcement learning from self-play data. The neural networks predictions improve the quality of the MCTS move selections, which translates to better training data for the neural networks in the next iteration. With curriculum learning and a moderate amount of training, our models learn intelligent behaviors somewhat better than the average human-level game-play.

1 Introduction

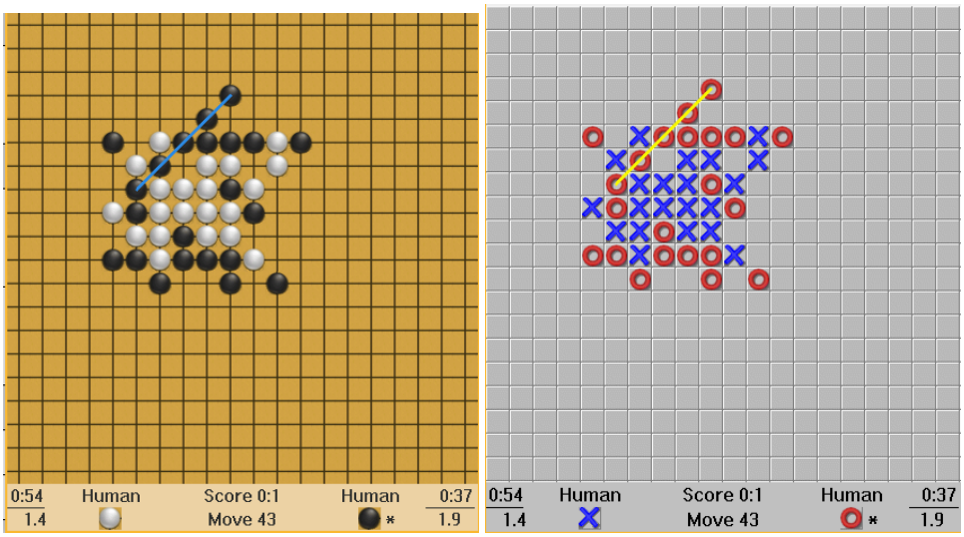
Supervised and reinforcement learning often rely on exorbitant amounts of data to achieve good performance, thus reliable labeled data is a frequent limitation to algorithms in artificial intelligence (AI). Human labeled data is in general costly to acquire in bulk, even for data relating to every day tasks that humans perform. With regards to solving multi-player board games with reinforcement learning (RL), it is infeasible for humans to label correct actions for arbitrary board states, as a) finding the optimal move at any state is not a trivial task in many games and b) generally only experts, which form a small subset of the general population, can provide such labeling. Furthermore, even labels provided by an expert may be erroneous. Finally, learning solely from human data may restrict performance of machine learning models to human performance, which may not be optimal.

Recently, AlphaGo Zero achieved superhuman performance in Go, utilizing Monte-Carlo Tree Search (MCTS) guided by neural networks trained solely from self-play data (7). AlphaZero is a generalization and improvement of AlphaGo Zero to games other than Go, such as Chess and Shogi (8). Despite using no human expert data, AlphaZero and AlphaGo Zero become stronger AI agent on Go than AlphaGo (6; 8), which was trained initially with supervised learning on human expert moves then fine-tuned with self-play data. AlphaZero consists of a policy network to provide prior probabilities for choosing actions MCTS search, and a value network to provide evaluations for non-terminal game states to reduce search depth. In addition, AlphaZero incorporates Upper Confidence Bound (UCB) to encourage exploration (3).

1.1 Gomoku

Here, we apply the techniques described in AlphaZero to the game of Gomoku, also known as Five in a Row. Gomoku is a two-player competitive game played on a game board of a 20 by 20 grid. The first player to place five consecutive, uninterrupted pieces along any row, column, or diagonal wins the game. Each player attempts to connect five pieces and prevent the opponent from connecting

five pieces. If the entire board is occupied without any player winning, then each player settles for a draw. We focus on the freestyle variant of Gomoku, where players take turns to place one piece per turn at any unoccupied position on the board. While game-play could be started from any starting board state, we focus on starting from an empty initial state. We note that in real-world game-play the game board is often initialized to be non-empty to reduce the advantage that the first player has, but we do not deal with this due to the difficulty of generating fair starting positions. In summary, the game that we address can accurately be described as tic-tac-toe played on a 20 by 20 grid, with a goal of connecting five “X"s or “O"s. Figure 1 shows some of the common display styles for Gomoku.



(a) Classic style (b) Tic-tac-toe style

Figure 1: Two display styles of Gomoku, note that game positions in a) and b) are equivalent.

2 Other Related Work

2.1 Competitive Multi-player Game Agents

Many work have created AI agents that aim to compete in competitive multi-player games, and this section only highlights a few strategies used in AI agents for competitive multi-player games. One common technique is to create rule-based agents that incorporate heuristics designed by human experts to pick actions or evaluate state, but these heuristics can be tedious to design and may not work well in practice (5). On the other hand, minimax search, given exponential time with respect to the number of moves, can find optimal policy to deterministic multi-player games by performing a search over all possible future game states. While pruning approaches such as alpha-beta may reduce the branching factor at every search node alphabeta, the overall time complexity is still exponential with respect to the number of moves and may be infeasible for games with large branching factor, such as Go. The Greenblatt Chess Program combines minimax search and heuristics to create a chess program that achieved a rating of around 1400, which beat humans that don’t play tournaments 80% of the time (4). Deep Blue defeated the reigning world chess champion Garry Kasparov in 1997 with heuristics-augmented minimax search. Stochastic methods like Monte-Carlo Tree Search (MCTS) address the search space problem by using stochastic samples for search but lose optimality guarantees, and in practice may achieve performance similar to or better than minimax search mcts.

2.2 Gomoku Agents

Freestyle Gomoku starting with an empty game board is solved by threat-space search and proof-number search and the first player to play can guarantee a win (2). Despite this, we thought it instructive to implement AlphaZero to solve the same task and see what behavior emerges. Fur-

thermore, opening rules have been designed for Gomoku and a closely game, Renju, to remove the first-mover advantage, and thus the game with these rules has not been provably solved.

Gomocup is an annual worldwide tournament for AI playing Gomoku (1). Gomocup is played in both the freestyle Gomoku rule with a 20 by 20 game board as well as the Renju rule with a 15 by 15 game board, although we focus only on the freestyle rule here. The tournament provides balanced openings, or starting board state, designed by experts so that the first mover does not have an advantage. The tournament lists its top performing AI agents, which include Embryo, Rapfi, and Yixin. As an example, Yixin performs on-par with or superior than the top human players in the world, and its executable is openly downloadable (10). These AI agents are often implemented with search techniques, and some agents adapt methods from AlphaGo and AlphaZero to Gomoku. We do not play against these agents, but only note their availability and the availability of their replay data; we choose to implement AlphaZero to rely on self-play only to see what behavior emerges.

3 Methods

As we for the most part re-implement AlphaZero, we describe AlphaZero with less details than the original paper (8), and we refer readers to the original paper to resolve any potential ambiguities.

3.1 Representations

3.1.1 State

In Gomoku, the game state can be perfectly represented by the current pieces on the board and an indicator for which player takes the next move. If D is the board dimension, then the state can be represented as a shape $(2, D, D)$ binary matrix, where the first (D, D) slice represents the pieces for the current player to move (1 if a piece is present, 0 if a piece is absent), and the second (D, D) slice represents the pieces of the opponent. Upon every move, after adding the move into the current player's slice, the first and second slices are exchanged, and the current player is set to be the opponent. Despite this perfect encoding, we append two additional binary slices of (D, D) each to represent the coordinate of the previous two moves (where 1 indicates the unique coordinate of each move); this is because Gomoku is often a very local game, and indicating the coordinates of the previous two moves may help the policy and value networks learn to pay attention to regions of recent previous activity. Finally, we follow the recommendation of a another re-implementation of AlphaZero on Gomoku and append a fifth (D, D) binary slice (9), where the values are either all 1's to indicate that the current player is the first player or all 0's to indicate that the current player is the second player. The reasoning is that being the first mover usually sees an advantage in Gomoku, and adding this layer may help provide the value network with a biasing term to learn the correct state values (9). In total, our state contains five slices and has shape $(5, D, D)$.

3.1.2 Policy

Given an input state, we obtain policy predictions from the policy network, and we structure the policy network to output a 20 by 20 matrix; the values of the matrix are between 0 and 1 and collectively sum to 1, representing a valid probability distribution. Each cell in the 20 by 20 matrix represents the prior probability of placing the next piece at that coordinate. During MCTS, we ensure that we do not place a piece at a coordinate that already contains a piece, no matter what the policy prediction is (although in practice the policy network learns to predict zeros or very small values ($< 10^{-5}$) for occupied coordinates.

3.1.3 Value

We encode the value of a state as a real number in $[-1, 1]$, where higher values represent better outcomes for the current player. When a game is finished, a value of $+1$ represents a win for the current player, -1 represents a loss, and 0 represents a draw.

3.2 Monte-Carlo Tree Search Self-play

We follow the procedure in Silver et al. (8) for using Monte-Carlo Tree Search to generate full games with self-play to train the neural networks. We summarize the details below.

3.2.1 Generating a Move

We define a leaf state (as opposed to an intermediate state) as either a state that is terminal (i.e. a player has won or there is a draw) or a state that has not been evaluated by the neural network yet during this game. Each intermediate state s_i keeps track of its value prediction v_i and prior probability prediction $P(s_i, a)$ from the neural network, where a can be any move. In addition, each intermediate state keeps track of the Q-values of each move from that state $Q(s_i, a)$ for all a in valid moves, and a count $N(s_i, a)$ of the number of times that move a was seen.

Instead of directly using $P(s_i, a)$ obtained from the neural network, we use $P'(s_i, a) = (1 - \epsilon)P(s_i, a) + \epsilon\delta(s_i, a)$, where $\delta(s_i, a) \sim Dir(\alpha)$, adding a different randomly sampled Dirichlet noise with hyperparameter α every time we compute $P'(s_i, a)$. ϵ controls how much the noise mixes with the $P(s_i, a)$. Using $P'(s_i, a)$ instead of $P(s_i, a)$ helps with exploration.

To generate a move for a state, we define the state provided as the root state s_0, which is initially also a leaf state. We perform K MCTS rollouts from s_0. For each rollout, we start from the root state s_0 and at any intermediate node s_i we select the next move using the Q-function and UCB criteria with some scaling factor c_{puct}

$$a_i = \arg\max_a Q(s_i, a) + c_{puct} P'(s_i, a) \frac{\sqrt{\sum_a N(s_i, a)}}{1 + N(s_i, a)}$$

until we encounter a leaf state s_n, generating a sequence of states s_i for $i \in [0, n]$ and a sequence of moves a_i for $i \in [0, n-1]$. If this s_n is not terminal, we evaluate this state with the neural network to get the value prediction v_n and the policy prediction $P(s_n, \cdot)$. We propagate the value from the leaf state to along all intermediate states s_i in the path back to the root. At each intermediate state s_i, we update the Q-value $Q(s_i, a_i)$ with v_n and increment $N(s_i, a_i)$ by 1.

After K rollouts as described above, we choose a move a by referring to the counts $N(s_0, a)$. When we want to generate a move as a training sample for the neural network, we pick move $\hat{a} \sim N(s_0, a)^{1/\tau}$, where τ is a temperature hyperparameter than controls amount of exploration (higher τ induces more exploration). When we want to generate a move during competitive gameplay, we greedily pick $\hat{a} = \arg\max_a N(s_0, a)$. We normalize $N(s_0, \cdot)$ into $p^*(s_0, \cdot)$ to be used as the label for the policy network for state s_0.

3.2.2 Generating a Game

To generate a full game, we run the procedure outlined above for generating a move repeatedly until the game ends, using the same neural network model to provide evaluations for both players in the game. At the end of the game, we assign the appropriate value ($+1$, -1, or 0) to each of the states we generated moves for. Each value serves as the value network label for that particular state. Once we generate a game, we use the generated policy and value labels to supervise the policy and value networks, respectively.

3.3 Neural Network

We follow the procedure in Silver et al. (8) to train the policy and value networks with labels generated from MCTS, although our neural network architecture may be different than the architectures in the original paper. The policy and value networks actually share a set of common layers, as in Silver et al. (8), and from here on we may refer to the policy and value networks as the policy and value heads, respectively.

Let $f : s \to (P(s, \cdot), v_s)$ represent the policy and value heads' outputs collectively, where s is an input state, $P(s, \cdot)$ is the predicted prior MCTS probability and v_s is the predicted value. Then the loss is computed as

$$\mathcal{L}(s; p^*(s, \cdot), v^*) = (v_s - v^*)^2 - p^*(s, \cdot)^T \log P(s, \cdot) + c_{L2}||\theta||^2$$

3.4 Replay Buffer

We keep a replay buffer of fixed size R of states from games. We shuffle the replay buffer every epoch and train the neural network with the entire content of the replay buffer every epoch. If

MCTS starts producing more game states than R, we evict the least-recently generated game states to maintain a size of R.

3.5 Symmetry

We follow the procedure in Silver et al. (8) to use symmetry to improve the consistency of neural network predictions. Since Gomoku is insensitive to rotation and flip, we apply every non-redundant combination rotate and flip to all states in each generated game before adding all rotated and / or flipped states to the replay buffer. This allows us to generate data with 8x faster rate and improve evaluation consistency. In addition, when we evaluate any state during MCTS, we first apply an arbitrary rotation and flip to the state before feeding the state into the neural network, and we invert the flip and rotation for the predicted prior probability. This further improves consistency.

4 Experiments

We heavily use multi-processing to divide and specialize processes. We utilize one process for training the neural network (train process), one process for evaluating states with the neural network (eval process), and 20 to 60 processes for running MCTS in parallel for generating data (MCTS processes). The MCTS processes query the eval process for state evaluations, which are performed in batch on the eval process. The MCTS processes forward any generated game states to the train process. The train process periodically forwards the newest model parameters to the eval process. Unless otherwise stated, we use training batches of size 1024 and eval batches of size $4/5$ times the number of MCTS processes. The train and eval processes each ran on either a Titan X Pascal or GTX 1080 GPU.

We run experiments on several board sizes D with some variations in hyperparameter and neural network architecture. As the architecture is not tuned and not of too much importance, we only describe an approximate architecture, and direct interested readers to our code. We report the hyperparameters and architectures used in each section. Unless otherwise stated, we set UCB weight $c_{puct} = 4$, $c_{L2} = 10^{-4}$, learning rate $lr = 5 * 10^{-4}$, and Dirichlet noise $\epsilon = 0.1$. The last activation in the value head is always Tanh, and the last activation in the policy head is always Softmax. The rest of the activations are always ReLU. Note that unless otherwise stated, we often just picked the first hyperparameters that worked and we did not have time or resources to perform optimization over the hyperparameter space, and this accounts for hyperparameter discrepancies between experiments.

4.1 Connect 3, 3x3 Board

Here we defined the goal as to connect 3 pieces in a row, essentially solving the game of Tic-Tac-Toe. The shared network consists of 3 ResNet blocks of output depth 16, the value head consists of one convolution (conv) and two fully-connected (FC) layers, and the policy head consists of one conv and one FC layer. We set Dirichlet noise $\alpha = 0.03$, temperature $\tau = 1$, and $K = 100$ rollouts per move. We used a replay buffer size of $R = 200000$. Note that we were still experimenting with hyperparameters here, so hyperparameters here might not be logical.

4.2 Connect 4, 6x6 Board

After several failed attempts, we simplified the network from the 3x3 Board experiment to facilitate training. The shared network consists of 3 conv layers, the value head consists of one conv and two FC layers, and the policy head consists of one conv and one FC layer. We set temperature $\tau = 1$. Here we speculated that more MCTS rollouts per move is needed to cover a larger board, so we set $K = 3000$. We realized that $\alpha = 0.03$ from the previous experiment might produce Dirichlet noise that modify the prior probabilities of single coordinates too much, so we increased α to 0.12 to distribute the modification over a broader set of coordinates. We use a replay buffer size of $R = 10240$.

4.3 Connect 5, 20x20 Board, First Attempt

We increased the network size from the previous experiment to account for increased difficulties of the problem. The shared network consists of 6 conv layers, the value head consists of one conv follow by two FC layers, and the policy head consists of one conv followed by one FC layer. We reduced α to 0.03 to try to account for the larger board size, which tends to reduce the concentration of Dirichlet noise modification. We attempted to gradually reduce τ from 1 to 0.25 and ϵ from 0.2 to 0.1 to reduce exploration as the model gets stronger. We also attempted to gradually increase replay buffer size R from 20480 to 81920 to increase the number of training samples once the network is more well-trained.

4.4 Connect 5, 8x8 Board

Here, we decided to try to train a fully convolutional model to see if the model can be trained on an 8x8 Board and adapted to larger board sizes, as in curriculum learning. The shared network consists of 9 conv layers; the value head consists of 4 conv layers, a global max pool, and two small FC layers; and the policy head consists of 6 conv layers. We set $\alpha = 0.06$. We start with temperature $\tau = 1$, $K = 5000$ MCTS rollouts per move, and replay buffer size $R = 20480$, then modify to $\tau = 0.5$, $K = 10000$, and $R = 40960$ at epoch 40000 as the evaluations improved in performance.

4.5 Connect 5, 12x12 Board

We attempted curriculum learning here and used the final neural network model in the 8x8 board (at epoch 60000) as pre-training; this is possible because the network is fully convolutional. We increase the number MCTS rollouts per move K to 15000 to account for the larger board size, keeping other parameters the same. We reduced the temperature τ from 0.5 to 0.35 after 2000 epochs.

4.6 Connect 5, 20x20 Board, Second Attempt

We used the final neural network model (at epoch 8000) in the 12x12 board as pre-training and kept all other parameters.

5 Results

Due to lack of a good evaluation metric for performance, we note that the author often played against models himself to evaluate the performance. The author is a somewhat seasoned veteran in Gomoku and has rarely lost to his friends in the past, so playing against the author is a crude measure of how well the model matches up against an amateur human player. We use the term crucial move to denote a move or one out of a few moves that must be made to prevent losing immediately, noting that crucial moves are routine in Gomoku and are usually easy to see (almost a reflex) for experienced human players – although inexperienced players will sometimes or often miss crucial moves. Subfigure 2b demonstrates a crucial move for connect 4. Since MCTS rollouts are slow (400 rollouts per second on 12x12 board), the author sometimes play against solely the policy network predictions, which uses $K = 0$ MCTS rollouts and is instant.

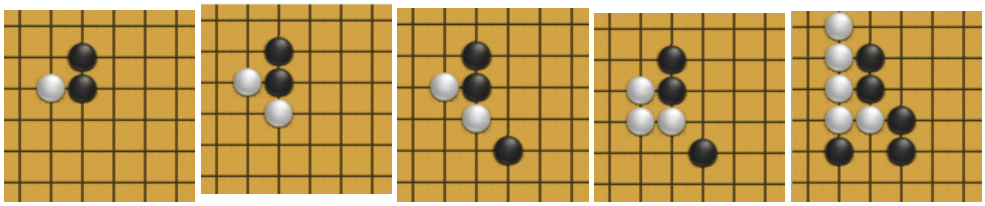
5.1 Connect 3, 3x3 Board

We achieved adequate performance after only 1200 epochs as evaluated by playing the author. The trained agent was able to tie the author every game. We don't report numerical results here, as we later found a bug in generating the label for the value head, and conclude that MCTS alone is able to solve Tic-Tac-Toe. In any case, the 3x3 Board is more of a debug experiment than anything interesting.

5.2 Connect 4, 6x6 Board

We achieve satisfactory performance after 4700 epochs (as evaluated by playing the author) and around 1 hour of training time. We note that the setup of connecting 4 pieces on a 6x6 board inherently has a 13 move forced win for the first-mover. The trained agent is able to consistently

beat the author when it moved first, but loses to the author when the author moves first. Moreover, We show some demonstrations of intelligent behavior in Figure 2, but in the interest of time and space do not report training plots.



(a) (b) (c) (d) (e)

Figure 2: Connect 4 on a 6x6 Board. a-d) shows sequential game play. In b), the model (white) correctly blocks the threat posed in a); had the model not blocked there, the author (black) would have connected 3 in a row, which is impossible to block. In d), the model correctly creates 2 unblocked two-in-a-row's, which the author cannot block. This leads to the model's victory in e)

5.3 Connect 5, 20x20 Board, First Attempt

Despite many epochs of training and much adjustments, we could not achieve any sort of satisfactory play with this experiment (as evaluated by playing the author). The model learns to connect the 5th piece after 4 pieces are already in place, but otherwise seem to exhibit somewhat random behavior. In particular, we were not able to find a temperature that works: when τ was too big (i.e. the earlier epochs), the model has high variation in the moves that it chose, but almost always missed crucial moves; when τ was too small (i.e. the later epochs), the model often picked one or a few moves with certainty, but these moves were still not the crucial move(s) required. In the interest of time, we do not report further numerical values or figures for this experiment.

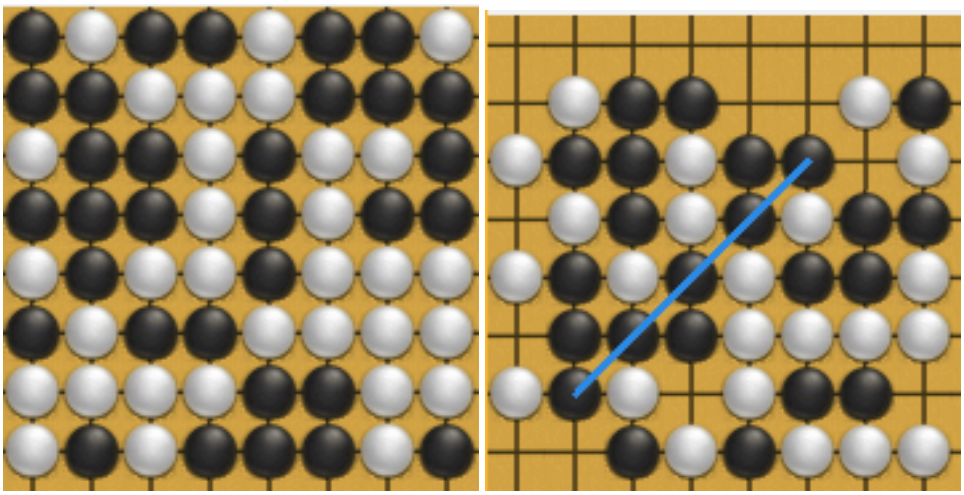
5.4 Connect 5, 8x8 Board

After 60000 epochs and 26 hours of total training on GTX 1080s, the model learns to play Gomoku satisfactorily. We display the training curve in Figure 5. Self-play games for generating training data demonstrate intelligent behavior, sometimes even filling up the entire game board (Figure 3). The author also evaluated the model by playing against just the policy network, with $K = 0$ MCTS rollouts per move. The model won once and lost once against the author with just the policy network, indicating that the policy network's predictions are consistently accurate. We expect the model to be even more robust with $K > 0$. On the other hand, the value network's evaluation often fluctuated somewhat between moves or exhibit other unexpected behavior (for example the network evaluated its value at the initial state as -0.2, even though it is the first mover and should have a positive value).

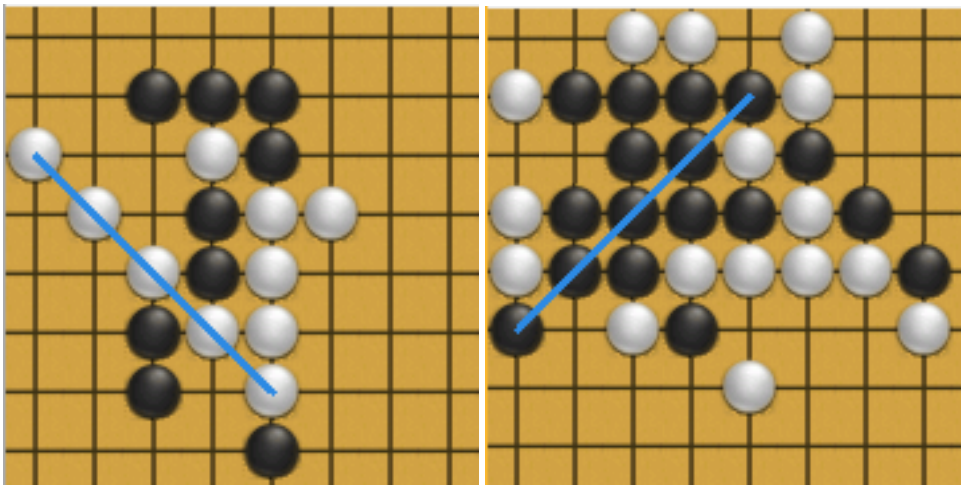
We set $K = 5000$ rollouts of MCTS. We compare the performance of the model at epoch 40000 to the performance of unguided MCTS by playing the models against each other, alternating black and white, and found that the model at epoch 40000 won 20 / 20 times. We also compare the performance of the model at 60000 epochs to the performance at 40000 epochs, and found that the model at 60000 epochs won 11 times, drew 4 times, and lost 5 times.

5.5 Connect 5, 12x12 Board

Fortunately, the pre-training on the 8x8 board seems to transfer well to the 12x12 board. We display the training curve in Figure 5. 8k epochs of training 12x12 took around 12 hours on GTX 1080s. The model retains many reflexes developed in the 8x8 case, and this helps the model make crucial moves in the 12x12 case. Model performance is not as robust as in the 8x8 case, and the model seems to have some weak bias for starting the game closer to corners, probably due to the 8x8 pre-training. For example, if we let the model moves first, it would place the first piece at (4, 4) or (7, 7), even though (5, 5) or (6, 6) is the true center of the board. When the model uses $K = 0$ MCTS rollouts per move, it occasionally misses a crucial move against the author. Furthermore, the value



(a) (b)

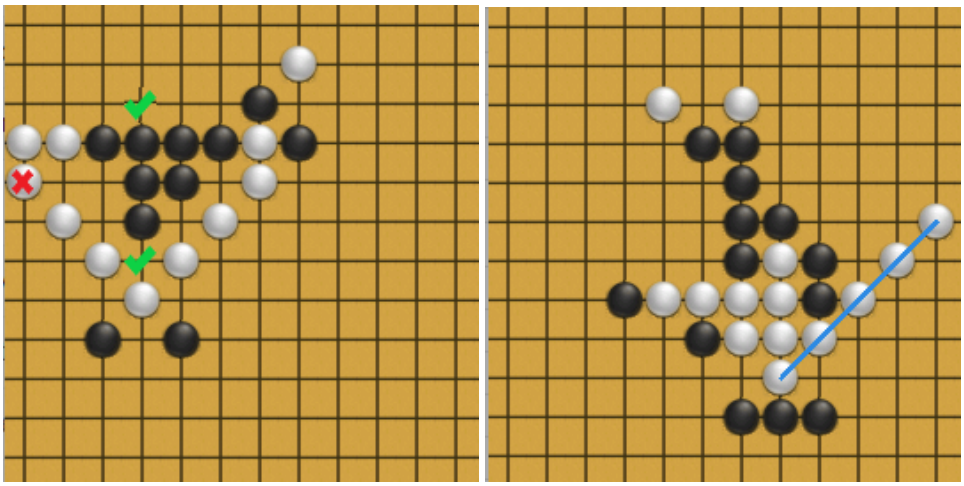


(c) (d)

Figure 3: Connect 5 on a 8x8 board. a) and b) show sample training data generated during self-play. c) The model (white) defeats the author (black), who made a suboptimal move earlier on. d) The author (black) defeats the model (white)

network's output is more inconsistent than on the 8x8 board, and the value network may even output a value of $+1$ when the true value is essentially -1. However, using $K = 15000$ rollouts of MCTS improves robustness significantly, and the model is able to capitalize on a few weak moves by the author in the beginning of the game to develop an winning advantage 20 moves later (Figure 4). We note that $K = 15000$ MCTS rollouts per move takes around 40 seconds per move when state evaluations are done on a GTX 1080 GPU. This shows that the network evaluations are at least good enough to lead to a robust agent with enough rollouts.

With $K = 15000$ rollouts of MCTS, we compare the performance of the model at epoch 8000 to the pre-trained model from the 8x8 Board, and found that the model at epoch 8000 won 19 / 20 times. With $K = 0$ rollouts and just comparing the policy networks of the same two models, we find that the model at epoch 8000 won 20 / 20 times. When we compare the model at epoch 8000 with $K = 15000$ to just its policy network (with $K = 0$), we found that the full model with $K = 15000$ wins 17 / 20 times. These finds suggest that additional MCTS rollouts provides significant assistance to the prior predicted policy.



(a) (b)

Figure 4: Connect 5 on a 12x12 board. a) The model (white), using only the policy network and $K = 0$ MCTS rollouts, overlooks a crucial move (green checks) and places a piece incorrectly (red X). The author (black, about to move) can quickly force a win. b) The model (white), armed with $K = 15000$ MCTS rollouts, wins a game against the author.

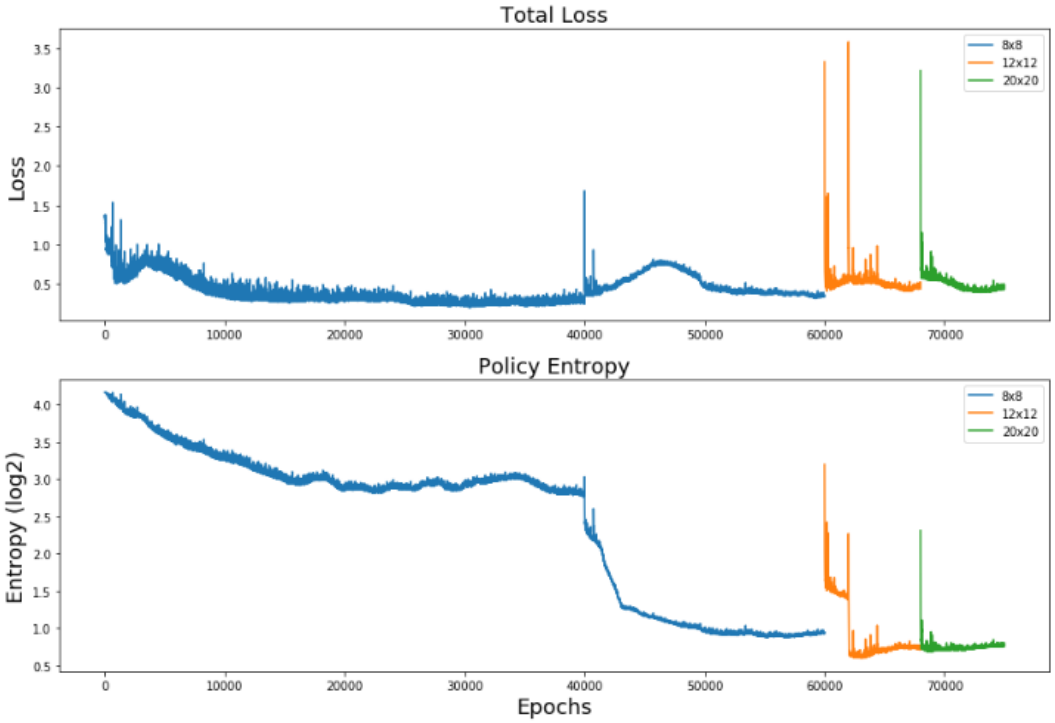


Figure 5: Training metrics for connect 5 on 8x8, 12x12, and 20x20 boards. As we trained 12x12 after 60k 8x8 epochs and trained 20x20 after 8k 12x12 epochs, we plot the three training episodes sequentially. Note that we lowered temperature at epoch 40k and 62k, which corresponded to drastic drop in policy entropy. The loss sometimes increases and sometimes decreases.

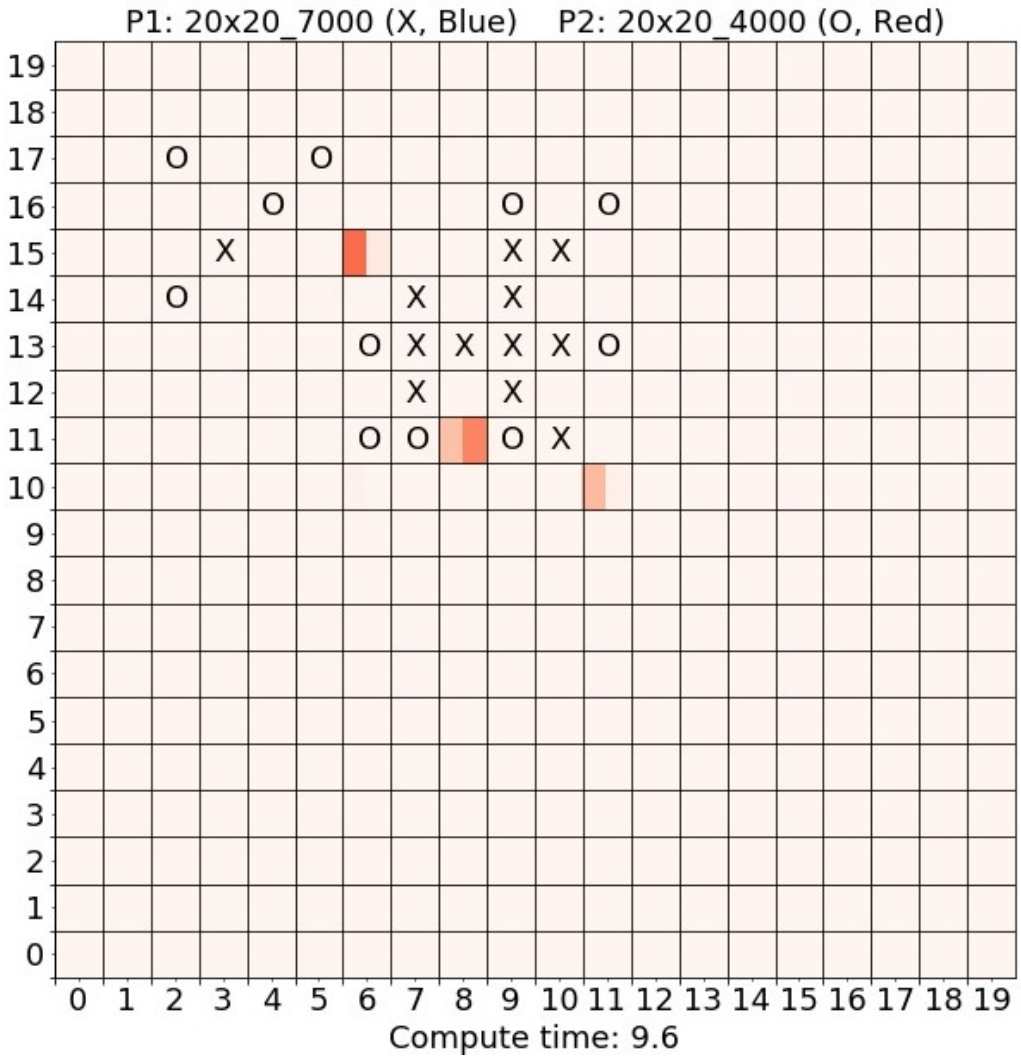


Figure 6: Competitive play between model with 7000 training epochs (X, first player, current player to move) and model with 4000 training epochs (O), with $K = 15000$ MCTS rollouts. Each cell of the game board is divide into left and right half. We display the prior policy evaluation distribution (left half) and the MCTS rollout move distribution (right half). Recall that the selected move is the move with the highest right-half value. Color scale is from 0 (white) to 1 (red).

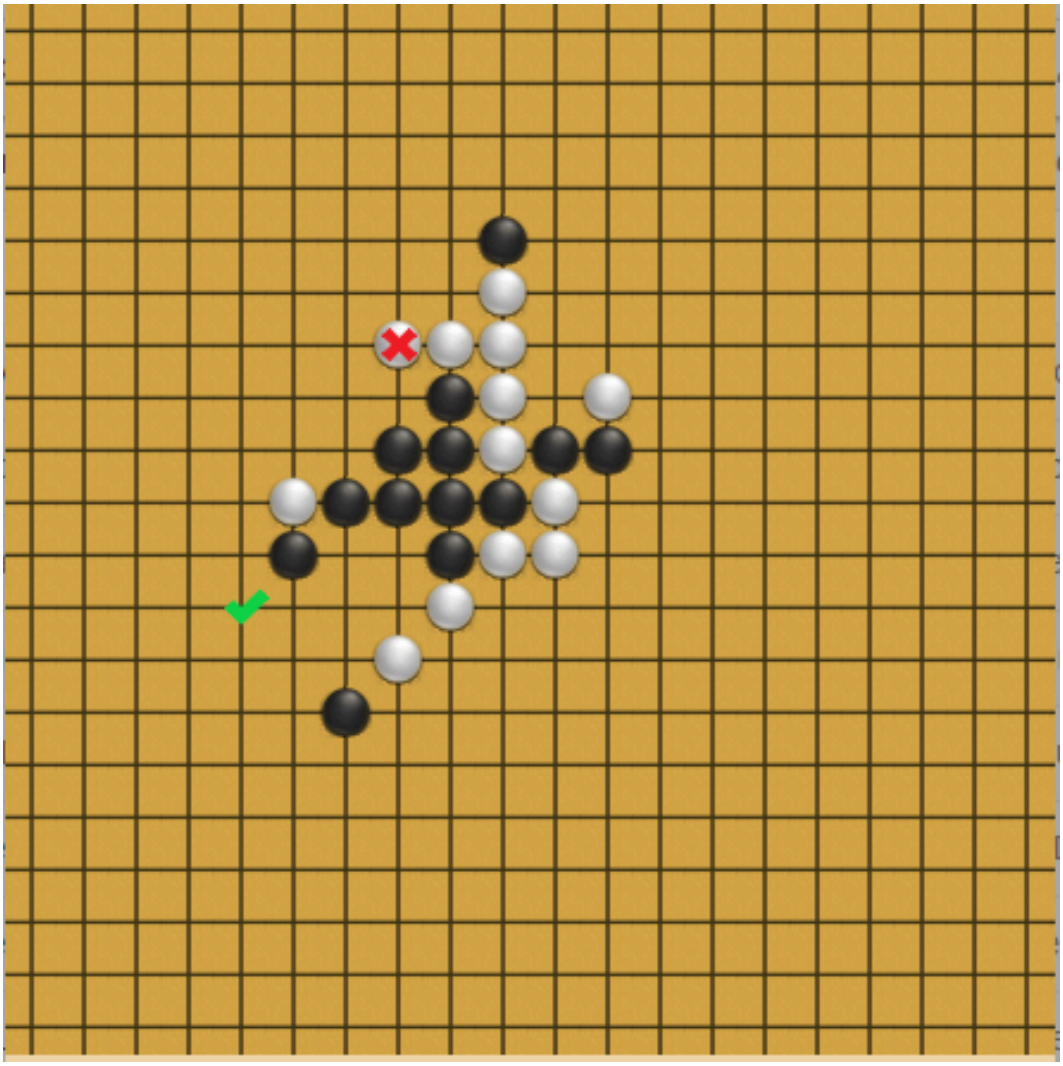


Figure 7: Connect 5 on 20x20 board. The policy-only model (white) with $K = 0$ MCTS rollouts overlooks a crucial move (green check) and makes an incorrect move (red X). The author (black, next to move) is about to win.

5.6 Connect 5, 20x20 Board, Second Attempt

The pre-training on 12x12 board seems to transfer well to the 20x20 board, and often retains the reflexes for crucial moves from small game boards. We display the training curve in Figure 5. 7000 epochs of training 20x20 took around 14 hours on GTX 1080s. As evaluated by the author, the policy network occasionally fails a crucial move with $K = 0$ MCTS rollouts (Figure 7). With $K = 15000$ MCTS rollouts, the model is more robust and does not seem to miss any crucial moves, but seems to plan worse for the long term than the 8x8 or 12x12 model.

With $K = 15000$ rollouts of MCTS, with compare the performance of the model at epochs 0, 2000, 4000, and 7000. Due to time constraint, we were only able to play a few pairs against each other. We observed that epoch 2000 wins against epoch 0 17/20 times, epoch 4000 wins against epoch 2000 18/20 times, and epoch 7000 wins against epoch 4000 14/20 times and against epoch 2000 18/20 times. This shows that the model improves with training, but we cut off the training here due to lack of time. We show an example of policy evaluations and MCTS traversal during game play in Figure 6.

6 Conclusion

We mostly successfully (as defined by our original expectations) implemented AlphaZero for the game of Gomoku with moderate amounts of training on at most two GPUs. We found that curriculum learning for self-play helps greatly and learns reasonable models in reasonable amounts of time, while starting with the full 20x20 Gomoku does not result in good agent behavior, at least with our current time and GPU constraints. The author ranks the overall performance of these AlphaZero models to be above-average human-level play – they would fair well against the average human that does not play Gomoku or play Gomoku occasionally.

References

[1] Gomocup. `https://gomocup.org/detail-information/`. Accessed: 2019-05-15.

[2] LV Alus, HJ van den Herik, 1, and MPH Huntjens. Go-moku solved by new search techniques. *Computational Intelligence*, 12(1):7–23, 1996.

[3] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

[4] Richard D Greenblatt, Donald E Eastlake, and Stephen D Crocker. The greenblatt chess program. In *Computer chess compendium*, pages 56–66. Springer, 1988.

[5] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[6] 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, 2016.

[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, 2017.

[8] 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, 2017.

[9] Junxiao Song. Alphagomoku. `https://zhuanlan.zhihu.com/p/32089487`, 2018. Accessed: 2019-05-15.

[10] Kai Sun. Yixin. `https://www.aiexp.info/pages/yixin.html`. Accessed: 2019-05-15.