Reinforcement Learning for Gomoku

Chao Yu
UCSD ECE
PID: A99049546
chy018@eng.ucsd.edu

Brian Whiteaker
UCSD CSME
PID: A04302767
bwhiteak@ucsd.edu

Inyoung Huh
UCSD CSME
PID: A53213515
i1huh@ucsd.edu

Wen Liang
UCSD ECE
PID: A53214852
wel245@eng.ucsd.edu

Hua Shao
UCSD ECE
PID: A53220045
h5shao@eng.ucsd.edu

Abstract

We build a self-trained Gomoku playing AI using deep reinforcement learning(RL) and ideas of AlphaZero. We implement a convolutional neural network using Tensorflow, which is the policy and value network, and Monte Carlo tree search(MCTS) algorithm. The MCTS uses the neural network to guide the expansion of the tree and the self-play results are used to update the neural network during training. We examine the behavior of this model with varying game board sizes, and explore UCB variant policies. Our results show that the neural network is able to learn specific features of Gomoku game play and generate a reasonable policies. Also, the whole reinforcement learning model has robust performance in this Gomoku game. From our results, Alphago Zero works well on Gomoku game and it is more powerful than MCTS. UCB_{tuned} and UCB_{KL} can balance the exploration and exploitation in reinforcement learning.

1 Introduction

Gomoku, also called Gobang or Five in a Row, is an abstract strategy board game having high levels of complexity. As the board size increases the possible strategies undergo a combinatorial explosion. The problem of teaching a model how to play Gomoku can be solved by reinforcement learning, which has proven in recent years to be a very powerful machine learning algorithm for decision making problems, and action selection. In this project, we implemented AlphaZero deep reinforcement learning model to build a Gomoku playing AI.

Gomoku is usually played on a smaller 15×15 Go board, however, in our project we change the board size to 6×6 and 8×8 due to the limit of our machine resources. We also adjust the number of pieces-in-a-row to win the game. For example, in a 6×6 board, we set the number of pieces-in-a-row to be 4, our game becomes a larger version of Tic-Tac-Toe. Our Gomoku model learns from self-play indicating that the model learns from its own experience without any human prior knowledge.

We built the Gomoku playing AI based on the AlphaGo Zero paper but several changes were incorporated. First, AlphaGo Zero used a large 40-block residual network to train the policy and value network. In our project we constructed a simple neural network with 3 convolutional layers and 3 fully connected layers. Second, we tuned the reward of win, lose and tie. The reason being that Gomoku is not a fair game. It has been proven that the initiative player (black) can achieve 100% wins. However, in the later stages of training, black has many more wins than white, making the model "frustrated" to develop some defensive strategy. Moreover, the original AlphaZero model used the multi-armed bandit policy PCUB and we tried to use different UCBs in our project.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

2 Related Work

Reinforcement learning(RL) - an algorithm for interacting with an environment (game), learns how to map an action to a state for maximal reward. RL has been implemented to solve for a game's best strategy. An advantage of RL is it's lack of need for a teaching signal during the learning process, resulting in a more adaptive system.

One important aspect of RL is in solving the temporal credit assignment problem which assigns credit to each state, and its associated actions. The solution to this problem was the Temporal Difference (TD) learning algorithm. One of the successful applications of TD learning is TD-Gammon. TD learning was later applied to Gomoku in 2003 [4], however, the results were not impressive. Based on TD learning, Zhao et al [6] combined Adaptive Dynamic Programming (ADP) and presented a program called ST-Gomoku, which learned by self-play. Still, ST-Gomoku needed some human-play supervised learning patterns to generate good training results.

As recently as 2017, AlphaGo Zero [2], a computer program without any human knowledge, beat the world No.1 ranked Go player Jie Ke. It used a Monte Carlo tree search algorithm to find its moves based on previously learned knowledge, generating extremely effective game-play strategies. Based on the structure of AlphaGo Zero, we trained a simplified structure and generated models for Gomoku game-play.

Based on Song's work on Gomoku[8], we build a similar board and game class. However, we implemented our *own* MCTS algorithm and AlphaZero model because we found they have some misunderstanding of MCTS and the implementation is based on a tree structure where a state will not always converge to a same node in the search tree. Also, we added more parameters, tunned rewards in game, tried different $UCBs$ and did some fine-tunning processes.

3 Structure and Methods

Figure 1 demonstrates the overall structure of our model implementation. The *GameBoard* class keeps track of the current position of all game pieces, next player, and other states of the current game. The *Game* class functions as an interface for players to make new moves on the game board and checks the current state. The *TrainingProcess* class arranges the self play process, data collection from self play and training of the neural network. The *PolicyValueNetwork* works as both the Monte Carlo Tree Search policy and value estimate function. The *MCTS* class is an implementation of Monte Carlo Search Tree (with or without the roll-out process). The *MCTSPlayer* class works as an interface during Gomoku self-play mode, human-play mode, and compete mode.

3.1 Convolutional neural network for Policy network

Convolutional neural networks (CNNs) have been applied successfully to the analysis of visual imagery and image classification. In the image processing field, CNN's using a kernel can extract the features of images and perform image classification. In Gomoku, there are two types of CNN used: policy network and value network. Both types of networks take as input the current game state, represented as an image. The convolutional network kernel can extract features of games and infer the overall situation of game. A softmax activation function is used at the output of the policy network mash output into the range 0 to 1 to represent the probability of each move. On the value net side, we use the tanh function for the output of the value net because the value should be in range -1 to 1.

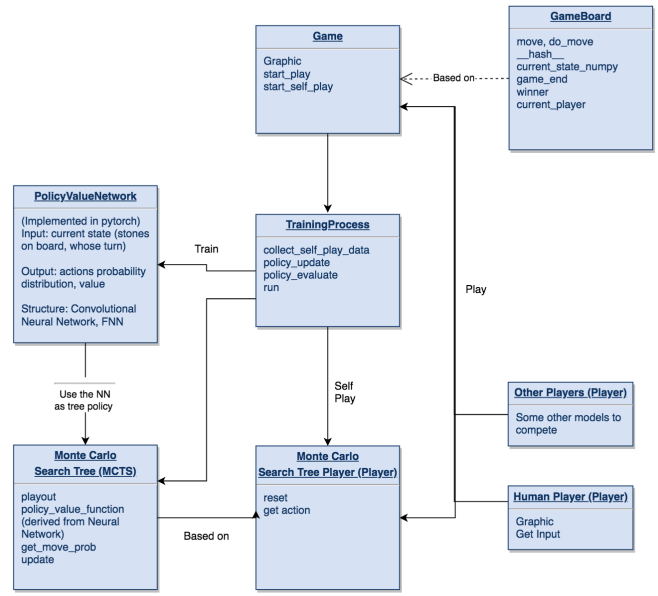


Figure 1: Diagram representation of our implementation structure

3.1.1 Model

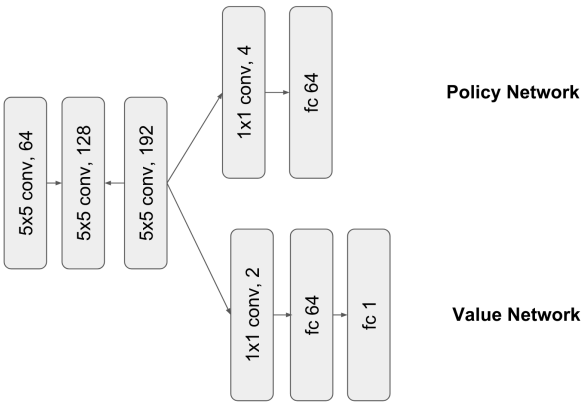


Figure 2: Policy Value network

Figure 2 shows the basic model we applied for our Gomoku game. This is a minimal model required to train our Gomoku game. In experiments we will use this model for comparison to other models. The input layer takes as input the state of the board. For the convolutional layer, we used 5×5 filters (3×3 for 4 pieces-in-a-row) and gradually increase the number of filters. For the final layers, it branches out to two different networks. Through policy network, it yields the probability distribution output of policy network using several fully connected layers and softmax function. Through the value network, it yields a value estimate of the current state of the game as an output of the value network using the fully connected layer and tanh activation layer.

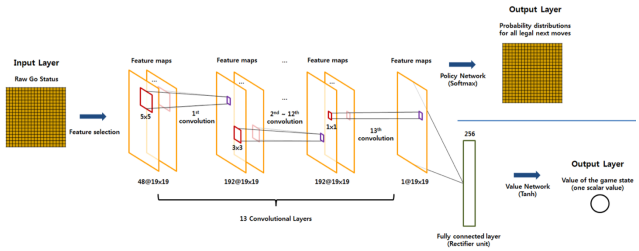


Figure 3: AlphaGo Zero Convolutional Model[2]

Figure 3 shows the neural network model used in AlphaGo Zero. The input layer takes a board state as input. In AlphaGo Zero, they used a complexed CNN model with 13 convoluational layers with 192 filters. In our model, we used 3 (5×5) convolutional layers with 64, 128 and 192 filters to make it solvable for the resources we have.

3.2 Monte Carlo tree search

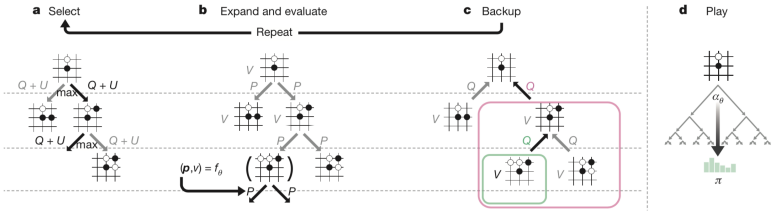


Figure 4: Monte Carlo tree search in AlphaZero [2]

By taking random samples in the decision space and building a search tree according to outcomes, Monte Carlo tree search (MCTS) finds optimal decisions in a given domain [1]. It can be used with little domain knowledge and successfully solves complex problems. This algorithm is most efficient when the search space of possible paths is extremely large as in Gomoku and Go. MCTS is divided into 4 steps: Selection, Expansion, Evaluation and Backpropagation. In Selection step, it chooses the state by maximizing $Q + u$ value from current state s to specific next state L in each simulation. u value will be explained in next UCB part which controls the balance between exploration and exploitation. For states encountered for the first time, the model will expand the leaves with a new node. Next, the rollout policy assists in evaluating those new nodes and the values can be used as a coarse estimation of the current state value. If it comes to the end of game, it will get a reward from the game agent. At the last step, we backpropagate the adjusted reward value back up the path to the root of the tree. One loop of these four steps is called one playout. By taking many playouts in MCTS, the algorithm can build up an idea of the expectation of rewards from current state and what is the best action to take.

For the AlphaZero, the policy and value network involves in the expansion and evaluation steps and combines those 2 steps together. The leaf will be evaluated by the neural network instead of rollout policy. Moreover, the output probability distribution will be stored in outgoing edges to its children and assist the growth of the tree. The policy and value network can be regarded as a coach that guide the tree grow from one node to a large search tree. After the growth of the tree is completed, the next action is took according to the the $N_{sa}^{1/\tau}$, where the N_{sa} is the visit count of action a from the current state s and the τ is a term of temperature.

3.3 Upper Confidence Bounds(UCB)

The UCB policy plays a key but delicate role in the effectiveness of MCTS search. The question is how best to explore a huge game-play space for best solutions, given a game state, without wasting large amounts of computing time to do it? Furthermore, how robust are the strategies found? If we miss attempting a strategy, this could leave open the possibility of a loss. In the area of Multi-Armed

Bandits (MAB's) this idea of possible loss due to choosing a suboptimal action is known as regret. In terms of RL and Go this has been called forgetfulness, meaning the likelihood of choosing some optimal-in-actuality action, has decayed and is nonexistent. So the action is not fully explored. Consider the example of epsilon-greedy MAB's. Epsilon is a value on [0,1] which dictates how many time intervals are spent in exploration versus exploitation. Imagine exploitation is high and at an early time under a time horizon a reward occurs that would have occurred regardless of action chosen. Then a highly exploitative MAB will believe this is the best option and always choose to exploit it. So it is gullible. UCB provides effective reduction of such a problem.

Each node in the MC tree, a.k.a UC tree (UCT)[5], is a UCB MAB whose arms/actions are the available in-game actions given a current state. These actions connect to a child node, which is again a UCB MAB. The rewards at the end of the game are propagated back up their path and discounted with distance from the winning leaf. This discounting method as opposed to using a flat reward is referred to as PUCT[2].

With many passes through a node, the estimated value of the node has tight confidence bounds on the best arm/action. Over time the unchosen arms have increasing confidence bound radius. The increasing upper confidence bound is used as a bonus added to the estimated value to make it more attractive as a choice. This means that although the optimal arm (adopting standard MAB terminology) is known, periodically UCB will explore sub-optimal arms just to make sure it didn't miss anything. In loss or accuracy plots this is manifested as a small oscillation.

The variants UCB_{tune} and UCB_{KL} augment the information used in bonus calculation. For UCB_{tune} consider the case of a first reward being very small and a second very large reward. So the variance is high across the two iterations. When added to the standard bonus for that arm it is highly likely to be chosen. This results in a better "understanding" of the reward for that arm. As knowledge improves, variance reduces. So the variance will have less impact to the standard bonus. For UCB_{KL} a comparison is made to the most recent best estimated for the node overall. The arms are considered to be drawing from independent distributions and so a mean is calculated for previous times and then compared using KL divergence. The bonus is then taken as the supremum of sequence bounded above.

The Upper Confidence Bound U is a bonus added to each arm value Q to help the tree selecting the node it would traverse in each simulation. For each node, UCB Q was based on the prior probability P and visit count N on the node and total visit count for its parent node. We also implement a scaling hyperparameter C_{puct}The equation for UCB was:

$$U(s,a) = Q(s,a) + C_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \tag{1}$$

The improved variant of UCB named UCB-tuned, the equation was:

$$\begin{aligned} U(s,a) &= Q(s,a) + C_{puct} \cdot P(s,a) \cdot \sqrt{\frac{ln\sum_b N(s,b)}{N(s,a)} min\left(\frac{1}{4}, Var(N)\right)} \\ Var(N) &= \sigma^2 + \sqrt{\frac{2ln\sum_b N(s,b)}{N(s,a)}} \end{aligned} \tag{2}$$

The σ^2 represented the variance of the Q value.
The last variant we implemented was UCB_{KL}. The following are the equations of UCB_{KL}:

$$max\{q \in \theta : N[\alpha] \cdot d\left(\frac{s[a]}{N[a]}, q\right) \leq log(t) + c \cdot log(logt)\} \tag{3}$$

$$d(p,q) = plog\frac{p}{q} + (1-p)\,log\frac{1-p}{1-q} \tag{4}$$

3.4 Loss Function

When training the network, at the end of each game of self-play, the neural network could provide the estimate of the policy π from the current state and also a predicted score v in the range -1 to +1.(+1 indicates winning the game, -1 indicates losing the game). The loss function we used to evaluate our

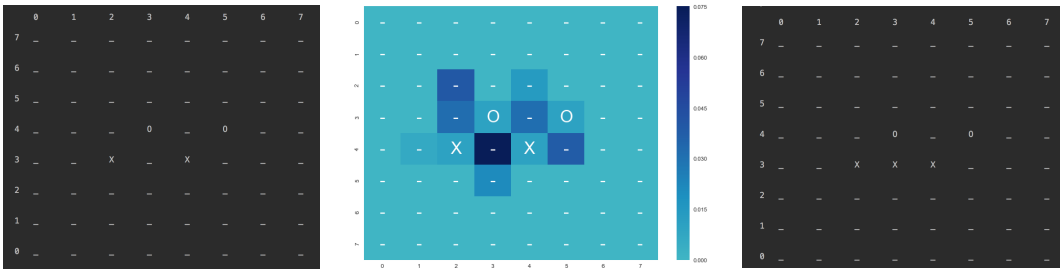
model was the same as AlphaGo. The equation is:

$$L = (z - v)^2 - \pi^T log P + c\|\theta\|^2 \tag{5}$$

The loss function was the sum of mean squared error and cross entropy error. $(z - v)^2$ was the sum of mean squared error. $-\pi^T$logP was the cross entropy error and $c\|\theta\|^2$ was the regularization factor to prevent overfitting.

4 Results

We use 400 playouts in MCTS on 6×6 board and 800 playouts on 8×8 board . Figures 5 illustrate the progress of the model's move selection based on an opponent's action. The sequence was captured during play between human and model which was trained for 3500 epochs. The "O" represents human moves and "X" stands for the agent's moves. From figure 5a, we can see which is the state that agent should move to next. Figure 5b is the heatmap of Q value. After calculating, the heatmap displays potential moves the agent can make. For higher probability the color of the square is darker. From the figure, the dark blue square with highest probability should be the next move the agent takes. Finally after calculation, a move is chosen by the agent and is shown in figure 5c.



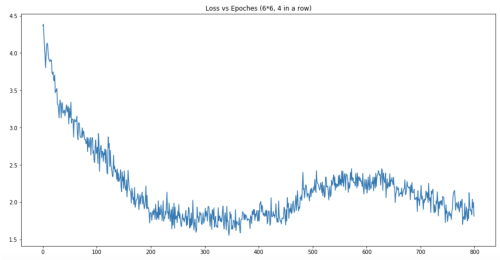
(a) step1 (b) heatmap (c) step2

Figure 5: HeatMap of the step

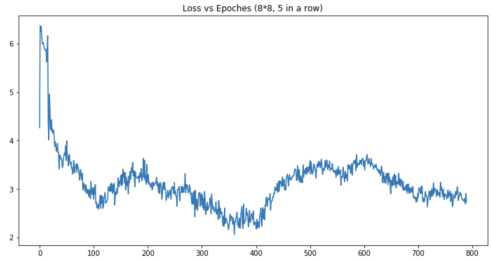
Loss vs. epochs is demonstrated in figure 6ab. The loss keeps decreasing during the first 800 epochs of learning. Since it often encounters new patterns during learning procedure, the loss is oscillating over the entire process. After evaluating the new patterns, the loss will decrease again. Therefore, the trend of loss is decreasing with more and more epochs of learning. The loss of 8×8 board is more unstable than 6×6 board. The oscillation is the result of curious exploration. In the 8×8 the oscillations are steeper across time steps when they occur as a result of more possible actions to be explored. This would mean that loss contributions from these new action choices require a more complete exploration by the model. In the 6×6 model we see more gentle oscillations due to less possible actions. In figure 6c, we used MCTS with 500, 1500 and 2500 times of simulations to demonstrate the performance improvement of our model. The 3 horizontal line describe the level of performance for these 3 MCTS algorithms so a point at the blue line means the model at that time performs a 25:25 result with MCTS with 500 simulations in our 50 rounds test. At last, our AlphaZero model exceeds MCTS algorithm with 2500 simulations in 6×6 board game.

4.1 Comparison of UCBs

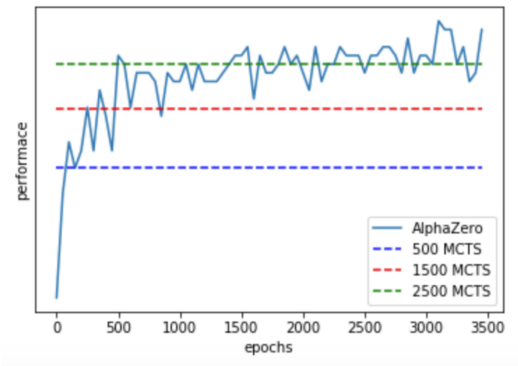
Figure7 demonstrates the loss and entropy loss during training UCB, UCB_{tuned} and UCB_{KL}. Figure 8 illustrates the counts of win against MCTS during training. From the plots, we can see that UCB has the lowest loss and entropy. Although UCB_{KL} has the highest loss, its performance on winning games is better than another two UCBs. Further analysis is shown in discussion section.



(a) Loss vs Epochs (6*6, 4 in a row)

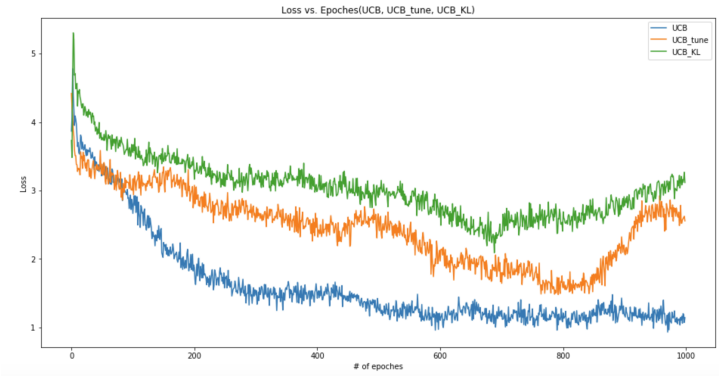


(b) Loss vs Epochs (8*8, 5 in a row)

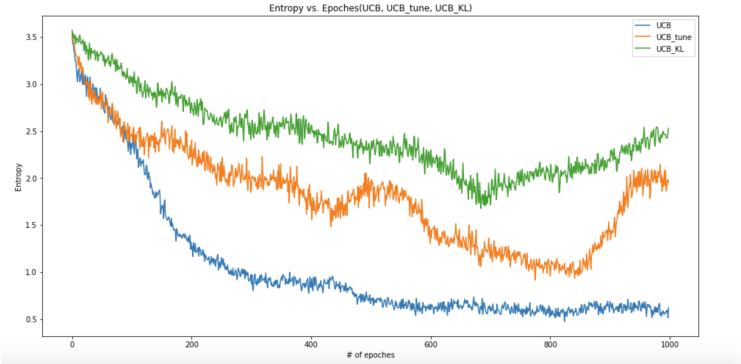


(c) Performance of 6*6, 4 in a row

Figure 6: Loss and performance of win



(a) Loss vs Epochs



(b) Entropy vs. Epochs

Figure 7: Loss and Entropy of training with UCB, UCB_{tuned}, and UCB_{KL} (1000 epochs)

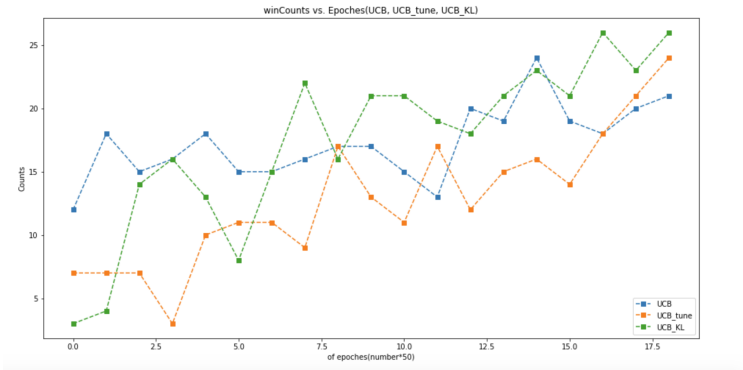


Figure 8: Counts of wins to MCTS with 500 situations: UCB, UCB_{tuned}, and UCB_{KL} (1000 epochs)

5 Discussion

From figure 5, we can see that the neural network can output a viable distribution for the available positions. The heatmap shows that the neural network can extract features of game-play strategy and evaluate the benefits of possible chosen actions. For example, there are many available positions in figure 5(b) but the $(row4, colomn3)$ and position (row 4, column 5) makes 3 stones in a row have highest possibilities. By repeated playouts/simulations in MCTS, the model takes the optimal action by considering both the output of the policy value network and its accumulated playout experiences. In figure6, the loss decreases with the increase of training epochs, which means that our neural network is attempting to explain some variances in the self-playing process. For self-playing, we provide some examples in section 7 appendix.

In figure 7, all three models with different UCB's have decreasing loss and entropy loss. Also, the loss decreases faster in UCB model. However, in figure 8, the UCB_{tuned} and UCB_{KL} model have more wins compared with MCTS algorithm with 500 playouts while the performance continues increasing, by comparison, the UCB model's performance does not improve much after the 700 epochs. Therefore, there is no clear correlation between loss and real performance for reinforcement learning. What is most important is effective and efficient exploration of possible actions. According to our result, more exploration can induce more confusion in our model and more loss in the policy and value prediction since there are many more new situations to explore. We noticed that the improvement for UCB model is slow and this could be an indication of overfitting. So a better balance in exploration and exploitation also helps to overfitting.

Another interesting result is that our model agrees with the truth that the 1st player has a guaranteed win strategy in 15×15 board but not in our 8×8 board. However, there are some guaranteed 1st player win strategies for 4 pieces-in-a-row game on 6×6 board. Then, we found that the neural network in a 6×6 game becomes "lazy" and just replays the most win strategy shown in figure 10. Because the neural network is pretty sure that this strategy is optimal, the heatmap shows the output distribution of the policy value network concentrates on the best actions. For defensive side, the output distribution is much more distributed without great threats such as 2-in-a-row situation for 4-in-a-row game and 3-in-a-row situation for 5-in-a-row game.

Some threats are shown in figure 9f, figure 9h, figure 10f and figure 10h, the distributions are concentrated much more on the positions to defend. Moreover, we found that it is a little bit hard for our model to learn defensive strategies if the most win strategy has been learned. This is because it is hard for the defensive player to get positive rewards from a win. Therefore, we introduced a tuned a positive reward for the defensive player if it can create a tie, thus, a non-negative reward encouraging it instead of a negative reward frustrating it. Using this tuned reward, the training process could possibly see a benefit. We found that reinforcement learning on an unbalanced game is very interesting and could be a good topic for future work.

6 Conclusion

The AlphaZero model works well on Gomoku game in a smaller board. The policy value network are implemented using TensorFlow. With plenty of learning epochs, the result shows that the neural network is able to catch features in the game. It also has learned both attack and defense strategies the game and can output a reasonable possibility distribution for actions to take in current state. We ended up spending two days finishing training the 8×8 board Gomoku model and spending 5 hours to train the 6×6 board Gomoku model. And our AlphaZero Gomoku model has better performance than MCTS model with 2500 simulations. The comparison between UCB, UCB_{tunned} and UCB_{KL} shows that a better balance between exploration and exploitation using UCB_{KL} can help to overcome overfitting and can improve the efficiency of a reinforcement learning process.

The first player always has huge advantages so that we want to add more rules and constraints to the Gomoku game in our future work. Regrading to MCTS, there are some areas of possible inquiry. The use of 1/4 in the calculation of the UCB_{tune} bonus is based on the variance of a Bernoulli arm with reward 0 or 1. There may exist a better value as the rewards are discounted moving back up the tree. The current value of 1/4 may be too large. A problem in finding a better value could involve finding bounds on the probability drift of the tree.

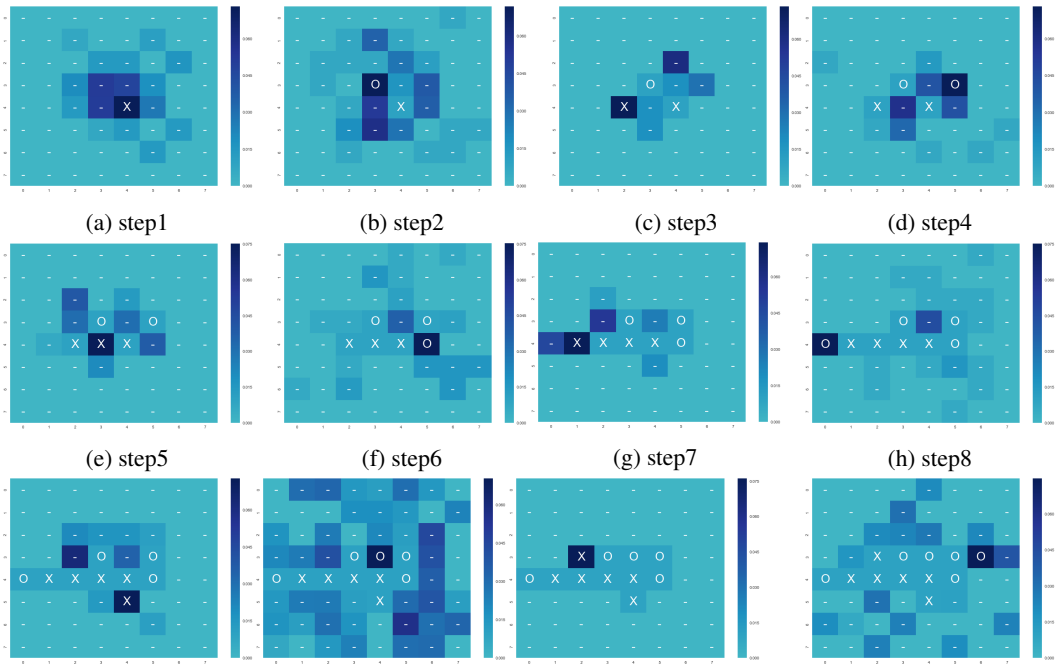
Another MCTS related inquiry is the value of a scaling parameter to UCB. Although the theoretical ideal is $\sqrt{2}$ for UCB, there was some mentions in literature of numerically optimizing this. Perhaps individualized value of per node could be found to improve UCB_{tuned}'s performance.

7 Contribution

Brian Whiteaker, Chao Yu, Hua Shao contributed with work on UCB and variants, heatmaps, analysis of results, and report writing. Wen Liang contributed on implementation of MCTS and AlphaZero model, analysis, and report writing. Inyoung Huh contributed with work on convolutional network and MCTS and report writing.

8 Appendix

Figure 9 shows the playing process of 8×8 play board and 5 in a row (heatmap) and Figure 10 shows the playing process of 6×6 play board and 4 in a row (heatmap).

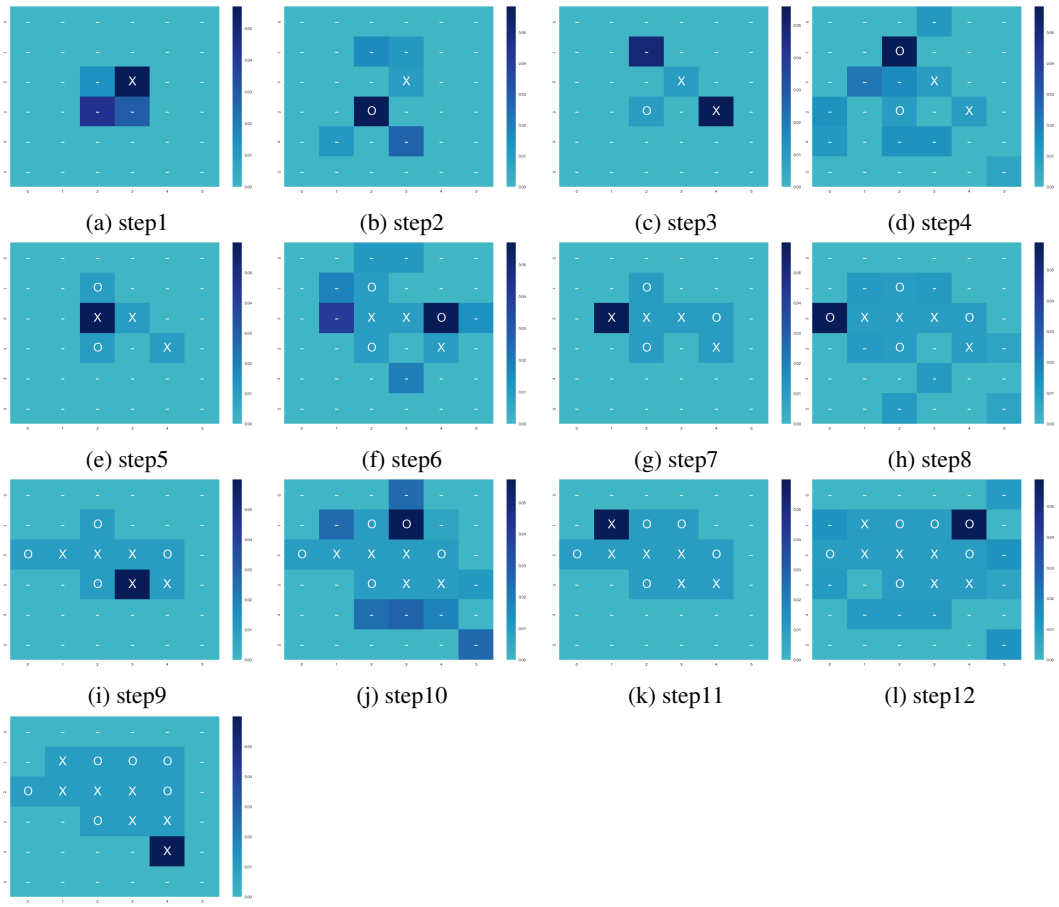


(a) step1 (b) step2 (c) step3 (d) step4

(e) step5 (f) step6 (g) step7 (h) step8

(i) step9 (j) step10 (k) step11 (l) step12

Figure 9: Heatmap of 12 playing steps 5 in a row(8×8)



(a) step1 (b) step2 (c) step3 (d) step4

(e) step5 (f) step6 (g) step7 (h) step8

(i) step9 (j) step10 (k) step11 (l) step12

(m) step13

Figure 10: Heatmap of 4 playing steps 4 in a row(6×6)

References

[1] Browne, C.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.

[2] D Silver, J Schrittwieser, K Simonyan, I Antonoglou, A Huang, A Guez, T Hubert, L Baker, M Lai, A Bolton, Y Chen, T Lillicrap, Fan Hui, L Sifre, G van den Driessche, T Graepel, D Hassabis. Master the Game of Go Without Human Knowledge. (2017). Doi:10.1038/nature24270.

[3] D Silver, A Huang, C J Maddison, A Guez, L Sifre, G van den Driessche, J Schrittwieser, I Antonoglou, V Panneershelvam, M Lanctot, S Dieleman, D Grewe, J Nham, N Kalchbrenner, I Sutskever, T Graepel, T Lillicrap, M Leach, K Kavukcuoglu, D Hassabis. Mastering the game of Go with Deep Neural Networks & Tree Search. Nature, vol. 529 (2016), pp. 484-503.

[4] J.M. Mo, Study and Practice on Machine Self-Learning of Game-Playing. Master Thesis, Guangxi Normal University, 2003

[5] Kocsis, Levente, and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning." Lecture Notes in Computer Science Machine Learning: ECML 2006, 2006, pp. 282–293., doi:10.1007/1187184229.

[6] Kuleshov, V., & Precup, D. (2014). Algorithms for multi-armed bandit problems. arXiv preprint arXiv:1402.6028.

[7] Zhao, D.B. & Zhang, Z., & Dai, Y.J. (2012). Self-teaching adaptive dynamic programming for Gomoku. Neurocomputing. 78. 23-29. 10.1016/j.neucom.2011.05.032.

[8]Song, X.J.(2018). $https://github.com/junxiaosong/AlphaZero_Gomoku$