INTELLIGENT GOMUKO: REINFORCEMENT LEARNING WITH ADVISOR

FINAL PROJECT REPORT

Zhao hexu
2019012399
zhi 90

Zhang bingliang
2019013280
zhi 90

ABSTRACT

AI has shown great power in board games. For example, AlphaGo beat one of the best player Lee Se-dol in Go game which is widely believed impossible for machine-intelligence to handle. And some of its improvements even demonstrate more astonishing performance. However, all these methods heavily rely on extremely large computational power. In this project, we transfer these methods to Gomoku and propose several different tricks to alleviate the unaffordable computational burden problem. And we also construct a complete and flexible framework that could be further used for other games.

***K*eywords :** Deep Reinforcement Learning, MCTS, Gomoku

1 Introduction

As recent booming of AI and deep reinforcement learning, a bunch of super human performance has been made by AI. The emergence of the AlphaGo and its variants, like AlphaZero demonstrated the extraordinary power of deep reinforcement learning over zero-sum, fully observable problem[1]. What's more, AlphaZero was trained fully by self play, without human domain knowledge, which sounds very trick and surprising[2].

However, the crux behind such extraordinary performance is the rigorous requirement for computational power. According to the announcement of developer of AlphaGo in Google, to train AlphaZero, more than 5000 TPU were employed during the training process[3], more precisely self play process. On the other hand, since the inherent difficulty of Go, it's hard for non-experted people to design great features or guidance of it. So in order to break the mystery of the state-of-art of AlphaGo, we focus on a much easier and more tractable chess game: Gomoku.

Instead of just solving game Gomoku in the following project, we also mainly focus on how to balance the intelligence of agent with the training cost. In another word, we want to find a simple with to do a trade-off between training cost and training result.

Some extent common AI framework of relevant algorithms are reimplemented so as to evaluate our proposed mehod-"reinforcement learning with advisor". Through comparisons between them, as well as their better scope of application, we can better use them in our future study and scientific research.

In addition, our framework is not only useful in Gomoku, but also scalable to other board games. It is a general architecture for fully observable and zero sum game. The framework is capable to be directly applied after some abstractions.

2 Existing works

The Gomoku originated in ancient Japan. It is a board game focusing on reasoning and calculation with great logic difficulty. Allis in 1994 had proposed an algorithm that is mathematically proven that the first player has win-strategy.

But people are also trying to find other well-performed algorithms[4]. Before AI has shown its great power in recent years, these algorithms are mainly based on search[5].

Before describing these search problems, some methods are based on some basic algorithms like greedy algorithms with handcrafted rules. For example, a greedy aggressive algorithm is to give the next black move on the place which could extend the length of consecutive black chess pieces. However, local best strategy does not always imply global best strategy. Although there are other more complex hand-crafted methods, the game is even more complex. When playing with human, such static algorithms will always be fooled eventually.

As for search-based methods, it could almost surely surpass human's capability. They are primarily based on Game Tree with min-max search, and $\alpha - \beta$ pruning[6].

Game Tree In a game theory, we could regard the states of the game as a tree structure in which each state represents a node on tree. Root node represents the initial state and leaf nodes correspond to a final game state that the winner and loser are determined. Node's children are these that could be reach after one move. If we assume the root is at depth 0 and the player that goes first is black part. Then all the nodes with even depth are for black part to move, and odd depth nodes are for white part to move. Such tree is essentially a search tree with two players move alternatively.

Min-max Search Min-max search is based on game tree such that we define a score function for each node on the tree[7]. Define the scores for even depth leaves are -1 and scores for odd depth leaves are 1. For other even depth nodes, they are black part's turn and want to maximize the score since score 1 means the white lose. Thus these nodes' scores are the maximum of their children's scores. For other odd depth nodes, their scores are are the minimum of their children's scores similarly.

$\alpha - \beta$ **Heuristic** The shortcoming of such search strategy is its exponential number of states to search. Thus it is almost impossible to find optimal solution when problem is big. Thus $\alpha - \beta$ pruning is introduced[6]. In min-max search, many nodes do not really need to dive in deeply. When search in a white node and its current score is smaller than its father's score. Since its score will only decrease in the future and its father wants to maximize the score. Further search in the white node's subtree is worthless, thus we could return. This pruning will not affect the rightness of answer [8].

Pure MCTS The MCTS algorithm successfully solve 9×9 Go game in 2008. MCTS is an efficient algorithm to solve game with large branch and reduce the possible search space based on UCT (unper confidence bound for tree).[9] By doing enough time of sampling, the policy returned by MCTS is an approximation of optimal policy. However, pure MCTS highly rely on sample number and rollout method. It is likely to miss some threatening branches, which will lead to failure. [10]

MCTS + Network Approximation The similar method of AlphaZero has been adopted directly to solve gomoku problem. However, because of unique property of Gomoku (ends fast, local focusing) and limit of computation power, it does not work well. Curriculum Learning method is proposed in [11], it works better than directly selfplay, but it needs a lot of manually designed representative states for network to learn, which is time-consuming and troublesome. Another method makes use of both CNN(convolutional neural network) and RNN(recurrent neural network), which is proposed in [12]. However, experiments show there is no significant improvement after adding RNN.

Limitation of Existing Work Gomoku is a relatively easier chess game with middle size branch factor. However, directly applying $\alpha - \beta$ pruning min-max search is still expensive since the computation cost increases exponentially as search depth increases. So there are two factors to balance:

- (1) Computation Consumption;
- (2) Prespecified Features and Domain Knowledge

Naive selfplay with MCTS, network approximation is able to get rid of tedious rule based features, but it is super conputationally expensive during training since it is slow to converge. Similarly, designing hundreds of prespecified features may save a lot of computational cost, but it increases difficulty in implementation and design. So, what we need now is trying to find an appropriate way to makeing trade off. On the one hand, the learned model is able to be trained efficiently, in another word, converge much faster. On the other hand, use as few as possible prespecified features that are simple to design.

3 Our methods

In this section, we will give our proposed method for solving Gomoku: “Reinforcement learning with advisor". And we also reimplement some existing method for evaluation and comparison, like greedy method, min-max search method with alpha-beta pruning, as well as pure reinforcement learning without any domain knowledge.

3.1 Our Greedy Method

In our work, we implement a greedy method to choose a position for placing the next piece. The basic idea is to only focus on the next step without long-term scheduling. More precisely, being faced with a board, the current player will use a heuristic function to give each empty position a score which approximately represents its importance. The heuristic function is given as the following,

$$score_0[P] = \sum_{d \in directions} \frac{maxScore}{10^{5-len_0^d(P)}}$$

Here, $score_0[P]$ and $score_1[P]$ represent the predicted obtained score if the position is placed with 0 and 1, respectively. $directions$ has size 4 that includes vertical, horizontal, diagonal, anti-diagonal directions. $len_0^d(P)$ is the maximum length of consecutive white pieces that goes through point P and is along direction d. $maxScore$ is a hyperparameter that represents the maximum score in each step.

As we could easily see from the formula, higher $len_0^d(P)$ leads to higher score. The score function essentially approximates the immediate gain towards forming a longer same color consecutive pieces. We indeed could naively use $score_0[P]$ as final heuristic function for white's action, but that does not take the opponent's situation into account. Therefore, our method use $score_0[P] + \lambda score_1[P]$ as final heuristic function. Here, we use a hyperparameter λ to balance the degree we take opponent's situation into consideration.

3.2 Our Min-max Search

We reimplement the min-max search with alpha-beta pruning described above. And we also add a trick to further boost its efficiency. The fundamental idea for alpha-beta pruning (also for other pruning tricks) is to give up further searching when the possible maximum answer is worse than already obtained answer. Thus we definitely want to find to good answer as soon as possible to later pruning meaningless search branches.

Our trick is based on this observation. In every situation, instead of randomly iterating empty positions, taking the action and going to the next situation. We first sort empty positions with respect to their importance, decreasingly. Then we try taking actions according this order. The heuristic for importance is same as used in the above greedy method. Because it is believed if we choose the next step with most gain, we could more likely find a good result.

And we also adopt a common trick that limits the depth of search process i.e. when we reach a particular depth we immediately return with approximated value.

3.3 Pure Reinforcement Learning Method

In the pure reinforcement setting, a policy-value network is adopted, which approximates the *optimal stochastic policy* and *optimal state value function*, given the current Gomoku board state. To balance the representation power and the computation pressure, a 2 blocks dense net (totally 31 convolution layers with small size of channels) is used in our preliminary experiment. [13]

The network is then trained through self played data guided by current best network and stabilized by Monte Carlo Tree Search(MCTS). That is, in each step, the agent will follow an exploration and exploitation policy to search in the tree. After searching a prespecified number of iterations, agent selects the most premising action to take. The network is then trained by random selected mini-batch of self played data, which is augmented based on some symmetry property in Gomoku.

So the whole training pipeline is, generating self play data, training the network, using updated current best network to generate self play data, and repeating until convergence (very low training loss) or having a relatively great intelligence.

The more detailed illustration is given below.

Policy Value Network The parameterized dense network f_θ takes current board state s_t as input and gives the approximated policy and state value function, $f_\theta(s_t) = (\mathbf{p}, v)$.

- **State s:** For a $a \times a$ Gomoku board, the state s here is represented by a $4 \times a \times a$ tensor. Expanding the four channels in first dimension, $s = [C_t \quad O_t \quad C_{t-1} \quad O_{t-1}]$. Here C is for the current player who will take action at time t while O for the other player. Then the two dimension tensor C_t is defined as:

$$C_t[i,j] = \begin{cases} 1 & \text{(i,j) is already down by player C at time t} \\ 0 & \text{else} \end{cases}$$

  and same definition for O_t.
- **Policy p:** The output $\mathbf{p}$ is the approximation to the optimal policy, that is $\mathbf{p}(a|s_t) \approx P^*(a|s_t)$. The returned policy is not directed used by agent, instead it is stored in nodes of MCTS as *prior probability*. With higher prior probability, it is more likely to be visited in MCTS.
- **Value v:** The output v is a scalar value between -1 and 1, which gives us how likely the current player will win the game given the current state s_t. Higher the v is, the higher winning probability is estimated.

Monte Carlo Tree Search (MCTS) The MCTS consists a bunch of tree nodes and especially a root node r. Here we regard MCTS as a stabilizer M of estimated policy and value returned by network. It takes current state s_t as input and returns the stabilized approximated policy $\pi(a|s_t) = M(s_t)$. Each tree node represents a board state and root note r represents s_t. In each tree node, it preserves some statistic information, such as visiting time, average winning rate (winning is in the view of node's parent) and prior probability from policy-value network.

Once given the current state s_t to MCTS root r, it will do the following four steps.

- **Selection.** First find a path from root to a leaf node based on policy that maximizes PUCT index. PUCT is a nice way to trade off the exploration and exploitation.
- **Expansion.** If the leaf node is not the terminal state of the game, the network will be called to return $(\mathbf{p}, v)$ in the view of current node. Then we expand current leaf node, letting those one step reachable nodes to be its children. And store the prior probability correspondingly.
- **Propagation.** Once we reach the leaf node, we are able to receive an evaluation value to the represented state. It can be either network estimated state value or terminal state reward. Then we back propagate the value to the root node, with sign flipping through the path.

This process will repeat for a fixed number of time. Then based on the soft-max visiting time of children of root node, a stabilized stochastic policy is given. After taking an real action by agent, root of MCTS will move to the corresponding child, keeping the subtree of that child and pruning the remaining subtrees.

Self Play Dataset In self play, during the whole game, both agents follows the stochastic policy returned by MCTS, which indeed can be seen as "play with self". In the self play, two types of supervision are collected.

- **Stabilized Policy.** Each action, the stabilized policy returned by MCTS can be seen as most steps look ahead to the policy returned by the network. That is we use current $\pi(a|s)$ as supervisor to adjust $\mathbf{p}(a|s)$.
- **Stabilized State Value.** At end of each game, the actual winning or losing is observed, denoting by z. z is regarded as a short term best supervision to the estimated v by network. Thus we use current z to as the supervisor of v.

To be specific, the data collected in self play is a set of "state-policy-value" pair $\{(s_i, \pi_i, z_i)\}_{i=1}^n$. Then the network is trained to fit the data.

Data Augmentation In order to make better use of self play data with symmetric property of Gomoku. A single "state-policy-value" pair is augmented to symmetric 8 different "state-policy-value" pairs by rotating and flipping. The data pair is shuffled before selecting into mini batch to make sure i.i.d. assumption still holds in some way.

Network Training The network is trained by currently collected self play data. For a single "state-policy-value" pair (s, π, z), the loss function is given by mixing the mean square loss of z and cross entropy loss of π. Suppose $f_\theta(s) = (\mathbf{p}, v)$, then the loss is

$$\mathcal{L}\left(f_\theta(s), (\pi, z)\right) = (z - v)^2 - \sum_{a \in \mathcal{A}} \pi(a|s) \cdot \log \mathbf{p}(a|s) \tag{1}$$

For a small mini batch $S = \{s_i, \pi_i, v_i\}_{i=1}^n$ of size n. The empirical loss is:

$$L_S(f) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f_\theta(s_i), (\pi_i, z_i)\right) \tag{2}$$

To better adapt to complex, highly non-smooth policy space, the Adam optimizer is used to automatically adjust the learning rate based on gradient information.

3.4 Reinforcement Learning With Advisor

3.4.1 Advisor for network initialization

In the last section, we propose our MCTS method. The method is essentially unsupervised since it is not provided with any games in advance. All the data is generated from the algorithm itself via selfplay. But an obvious problem is that two naive players might not provide good games to learn from at beginning. Thus it is really hard for network to converge. That induces an idea that instead of letting two naive players do self play from scratch, we could first let the MCTS learn from a good advisor. More precisely, we let the Policy Value Network learn the game played by min-max search. The algorithm is mainly divided into three steps.

1. We do selfplay using min-max search and obtained hundreds of thousands of games.
2. We use these games as train dataset to train the Policy Value Network with random initialization.
3. Let the trained Network do self play to further generate data and then further train itself using these new data.

This method is pretty like how we human learn. When we do not have any related knowledge facing a new type of problem, it is always better to learn from an advisor compared with directly diving into the problem. And when we have learnt somewhat, independent exploration on this problem tends to become more effective.

Besides, in order to have diverse games when generating data, we use $\epsilon-$*greedy* when choosing action during generating data, i.e with probability $1 - \epsilon$ we choose the best action, otherwise we choose action according to the following distribution.

$$P(a^*) = \frac{e^{V(a^*)}}{\sum_{a \in A} e^{V(a)}}$$

Where A is the set of available actions and $V(a)$ is the value obtained in min-max search after taking action a.

For every step, the minmax search will assign a score to each action representing the likelihood to win after taking the action. Based on this score, the minmax search could calculate a probability distribution to choose actions. And this calculated probability distribution is exactly what our model learn. In our method, we provided two ways "softmax" and "onehot" for generating this probability distribution. Here, we suppose $V(a)$ is the score we obtained after taking action a and $P(a)$ is the probability to choose action a.

- **softmax**

$$Pr(\hat{a}) = \frac{e^{V_{\hat{a}}}}{\sum_a e^{V_a}}. \tag{3}$$

- **onehot**

$$Pr(\hat{a}) = \begin{cases} 1 & \hat{a} = argmax_a V_a \\ 0 & otherwise. \end{cases} \tag{4}$$

Besides, we also obtain an exponentially decayed return along these games. And we could then use these return and probability distribution to train Policy Value Function via cross entropy and square loss.

After training, we then repeat the above proposed 'selfplay sampling' and 'network training' procedure until convergence.

3.4.2 Advisor for rollout function

Instead of simply using network returned value v as "rollout" value for MCTS, some different types of rollout functions are considered. Formally, for a particular game G and two agents A_1, A_2, given a stochastic policy $\pi_1(\cdots \mid x), \pi_2(\cdots \mid x)$ adopted respectively by two agents, for any state x, define random variable $G(x \mid \pi_1, \pi_2)$ as:

$$G(x \mid \pi_1, \pi_2) = \mathbb{I}\{A_1 \sim \pi_1 \& A_2 \sim \pi_2, \text{take actions orderly and } A_1 \text{ wins}\} \tag{5}$$

Indicator function $\mathbb{I}(event)$ return 1 if $event$ is true, otherwise -1. The *optimal rollout function* $R^*(x)$ is defined to be the expected value if both agents follow *optimal policy*:

$$R^*(x) = \mathbb{E}_{a\sim\pi^*}[G(x|\pi^*)] \tag{6}$$

However, just like problem encountered before, it is very hard to directly learn exact $R^*(x)$ by doing selfplay and MCTS. So we come up with some reasonably well rollout functions to estimate it.

- **Min-max Rollout (mmRoll)** Given π_{mm} to be the policy for min-max search agent, R_{mm} is defined as:

$$R_{mm}(x) = \frac{1}{N}\sum_{i=1}^{N} G_i(x \mid \pi_{mm}) \tag{7}$$

- **Random Rollout (ranRoll)** Given π_{mm} to be the random policy, R_{ran} is defined as:

$$R_{ran}(x) = \frac{1}{N}\sum_{i=1}^{N} G_i(x \mid \pi_{ran}) \tag{8}$$

- **Mixed Rollout (mixedRoll)** Given any possible rollout function $\hat{R}$ and network return v, define R^{θ}_{mixed} as:

$$R^{\theta}_{mix}(x) = \hat{R}(x) \cdot \theta + v \cdot (1 - \theta) \tag{9}$$

4 Achievement and Experimental Result

4.1 Code related

A complete architecture and pipeline for training and evaluation is constructed, this platform could also be transfered to use for other games. We provide our source code (2000 lines of code in total) on https://github.com/TarzanZhao/AIProject.

The whole tasks are implemented in following modules:

- Game simulator module that support game between different agents.
  - Human player.
  - Agent using MCTS algorithm.
  - Agent using Min-max search.
  - Agent using Greedy algorithm.
  - Agent that could provide *getAction* interface.
- Reimplement three canonical methods:
  - MCTS with Policy Value Network given a specific rollout function.
  - Min-max search with alpha-beta pruning.
  - Greedy method.
- Design a complete training architecture and pipeline as shown in fig 1.
- Global argument, log, time counting, file storing and retrieving module.
- UI and visualization for experimental result.

4.2 Experiment setting related

- We do experiment on different size of boards including 8×8, 10×10 and 15×15.
- The tested conditions for winning include having four (8×8) or five (10×10, 15×15) pieces in a line.

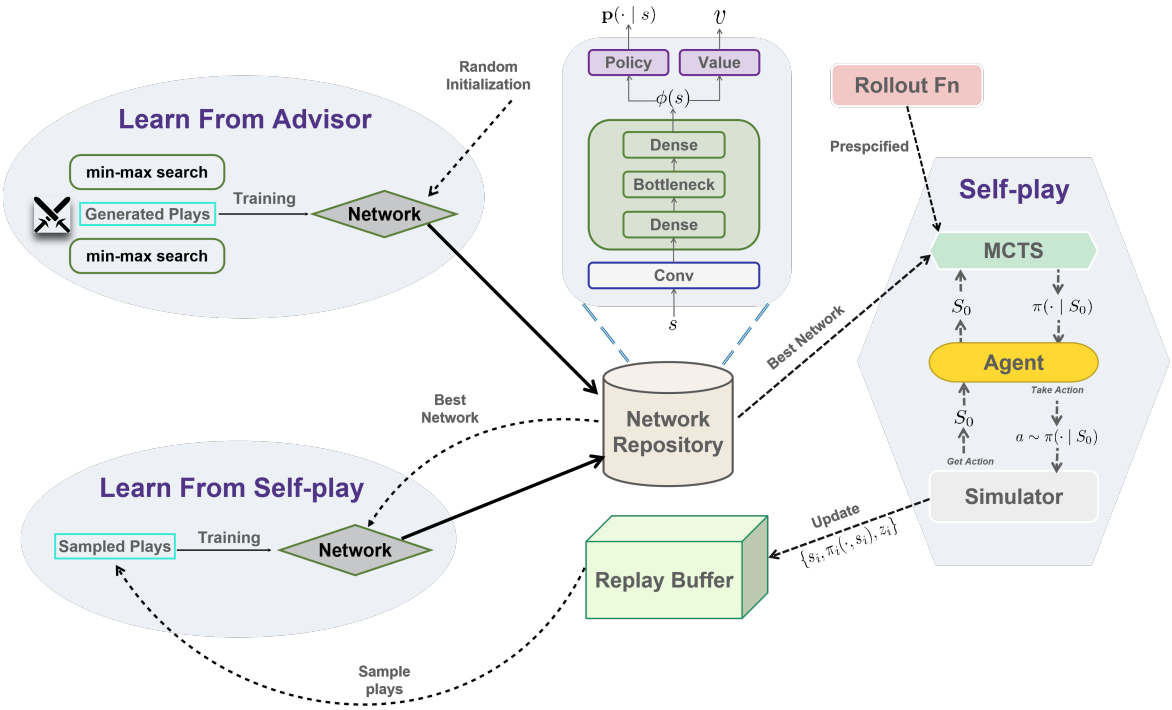


Figure 1: Architecture and Pipeline for Training Process

4.3 Data Related

Data from Min-max search We use 4-depth Min-max search to generate thousands of rounds of games for Reinforcement Learning With Advisor to learn.

The Min-max search with depth 4 could perform pretty well and thus we believe it will give us accurate score for actions leading to provide reasonable probability distribution from which our model will learn.

On top of the quality of probability distribution, we also pay attention to diversity of games. Because, we want our model to learn diverse strategies under different scenarios. Thus we force the min-max search to use ϵ-greedy when choosing actions. It is worth-mentioning that, relatively high $\epsilon(e.g.\ 0.3)$ always results in quickly finishing and too high $\epsilon(e.g.\ 0.5)$ always provide noisy games including many unjustifiable and also rarely appeared actions. Thus to make games reasonable and relatively long, most game data are generated using ϵ below 0.2 which is further divided into two levels $\epsilon \sim Uniform(0.02, 0.1)$ and $\epsilon \sim Uniform(0.1, 0.2)$ representing low and high randomness, respectively. To make our model obtain robustness facing non-intelligent opponents, we indeed generated small portion of data played by player with low randomness and high randomness. To take comprehensive situations into consideration, the player with low randomness could be either first or second player.

Data from self play To distinguish different stages more clearly, in the following passage, we use *rounds* to denote the number of rounds of data generating and network training. And let *epochs* denote the number of passing whole data to network during each round. The number of expanding time in MCTS for each move is denoted as *tree iterations*.

If not specified, the training setting is always 400 tree iterations, 100 epochs and 50 rounds. In each rounds, 25 entire selfplay games are collected and added to replay buffer. Mini-batch is randomly sampled from replay buffer for network training.

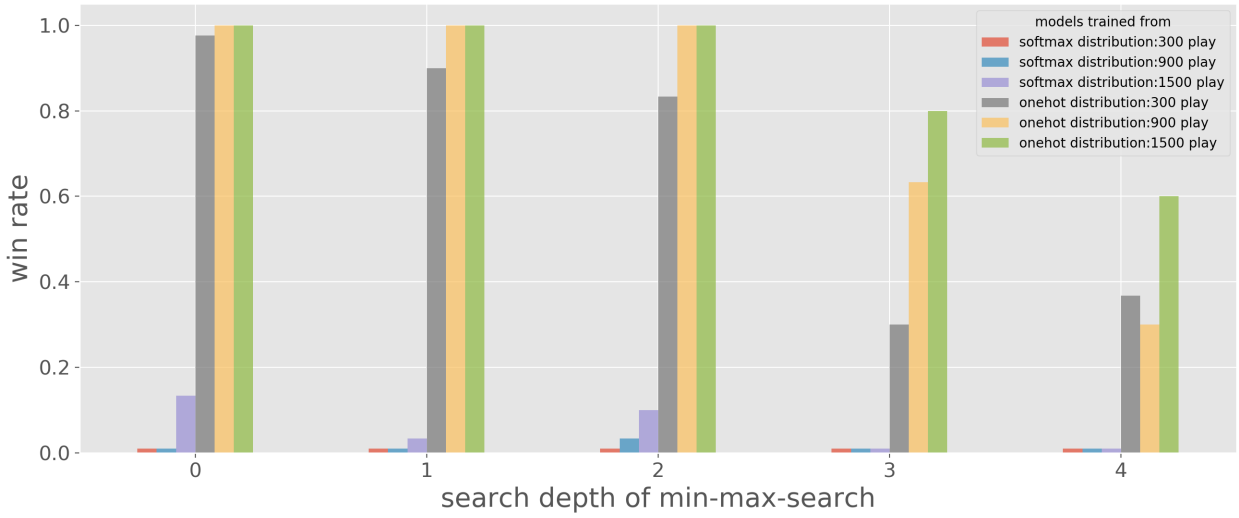


Figure 2: win rate for competing with min-max-search

4.4 Experimental Result

4.4.1 Pure Reinforcement Learning

The model trained by pure reinforcement learning encountered some problems. First is "collapsed self play". The selfplay is extremely simple and as result model can hardly learn something useful from it. Second is "redundant self play". Because of lacking prior guidance, network tends to randomly play at beginning and hard to converge. Some representative selfplay data is shown in Appendix.

4.4.2 Reinforcement Learning With Advisor

Advisor for initialization And to evaluate the influence of number of plays envolved in training. We provide three different sizes of min-max search generated datasets including 300, 900 and 1500 games, respectively. And we also compare the effectiveness of using 'softmax' or 'onehot' probability distribution for initializing (training) Policy Value Function.

To test these different kinds of configurations of Advisor for initialization. We let the models trained from these configurations compete with min-max search agents with different search depth. We could see these models' capability through their winning rate. The whole result is shown in fig 2.

The X-axis and Y-axis represent the search depth of competed min-max search and the winning rate of our models when competing with these min-max search agents, respectively. In the name, the prefixes 'softmax distribution' and 'onehot distribution' are the used probability distribution during training. And suffixes '300 play', '900 play' as well as '1500 play' are the sizes of used datasets.

From the diagram, we could easily observe some patterns which show us the underlying law.

- It is astonishing that our trained models outperform its advisor. It starts learning from min-max search of depth 3 and finally beats min-max search. As we could see, training on 1500 play could win its father min-max search of depth 3 with probability even up to 80% and is even able to win the depth 4 version with more than 50% probability. That shows our method's effectiveness.
- Training with onehot distribution significantly outperform these trained using softmax distribution. Actually, trained using softmax distribution indeed has inteligence but could not consistently show good intelligence. It always makes mistakes and thus have extremely low winning rate. But it indeed far better than a random agent as we could see the 'softmax distribution:1500 play' could at least have chance to win min-max search with depth 2. Comparatively, models trained using onehot distribution could outperform min-max search agent.

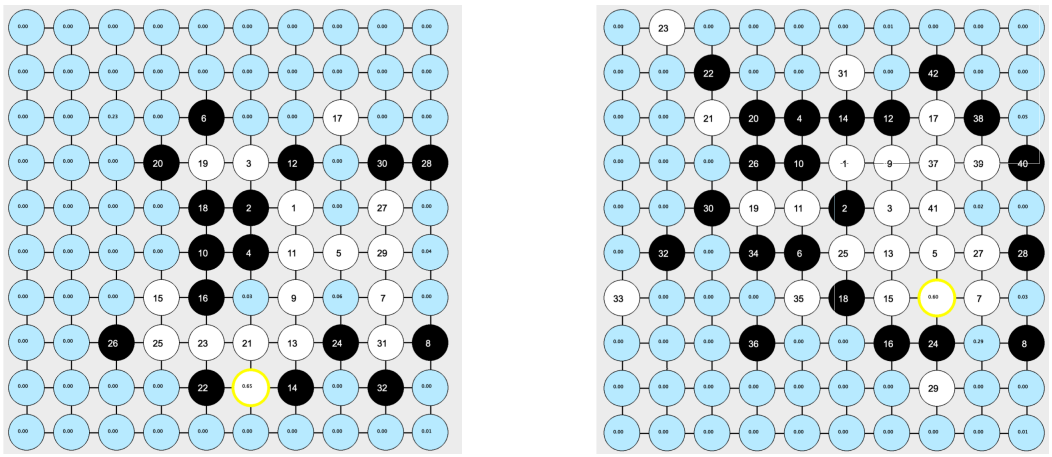


Figure 3: Games play from Advisor for Initialization

Training on merely 300 play could beat(win rate is higher than 50%) min-max search of depth 2. 900 play and 1500 play are capable of beating min-max search of depth 3 and depth 4, respectively.

- Training on more data leads to better performance on both using softmax and onehot distribution. This phenomenon is exactly in accordance to what we observed in deep learning community. More data imply more comprehensive sampling of population and thus more comprehensive situations. Our models learn more and thus could perform better.

From the games(fig 3) played by this method, we could see its intelligence. On both games, the agent using advisor for initialization is white part and goes first. First, we look at the left image. The white piece with yellow boundary is a wonderful move. Since it provides two directions(left-up and right-up) to further attack, no matter which direction its opponent blocks, it could choose the other direction to place piece. And this piece also prevents two black pieces from connecting with each other. From this move, we could see our method has great intelligence to find some optimal(even must-win) actions. And from the right game, we could see many four-black-in-a-line with two white pieces on its heads indicating our method is stable to detect and shatter the black's attack for every time. And it also catches the opportunity to attack and finally beat its opponent in a protracted way.

Advisor for rollout function According to the experimental results, the performance of network is dramatically improved after using rollout function, especially with mmRoll. The rollout function is mentioned in "Our Method" part, which takes in current state s as input and outputs an estimated value for the state.

To illustrate the super power of rollout function, the network is trained and evaluated with baseline agent. And result is shown in fig 4.

In graph (b), the model is **trained** by pure network return and **evaluated** by 4 rollout functions in legend. That means network has already learned a very good prior distribution $\mathbf{p}(\cdot, s)$, but it learns a relatively poor v. Since by only replace rollout function, we are able to dramatically improve network performance.

In graph (a), the model is **trained** by mmRoll and **evaluated** by 4 rollout functions in legend. That means, network are able to learn both good prior distribution $\mathbf{p}(\cdot, s)$ and value v, since in evaluation, four different types of rollout functions perform almost equally well.

So it is not hard to find that by using mmRoll function, network tends to learn a better prior distribution and value estimation. To further understand why network trained with prespecified rollout function achieves much better result, we fetch and examine the used selfplay data. As expected, the selfplay data is much better than before, which contains more steps, more attack and defence strategies. Some representative selfplay data is shown in 5. The pieces are concentrated, and are also easier to notice simple strategies mastered by network. Though such good selfplay data does not appear very frequently since actions are sampled from stochastic policy returned by MCTS, it indeed shows that with mmRoll,

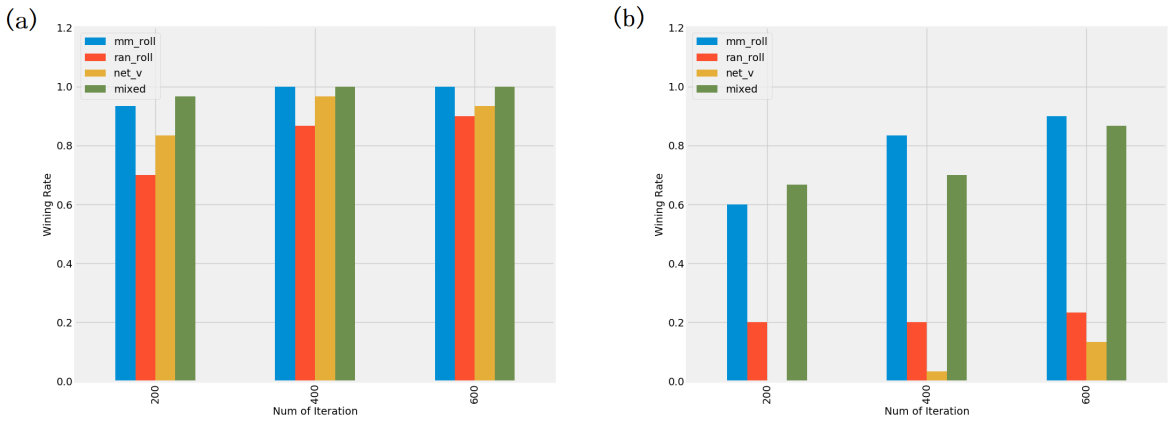


Figure 4: Comparison For Different Rollout Function in Training and Evaluation Step with Baseline. In (a) the network is trained by mmRoll while in (b) is trained by pure network return. Both are evaluated by 4 rollout functions in legend.

selfplay data tends to be more meaningful and educative.

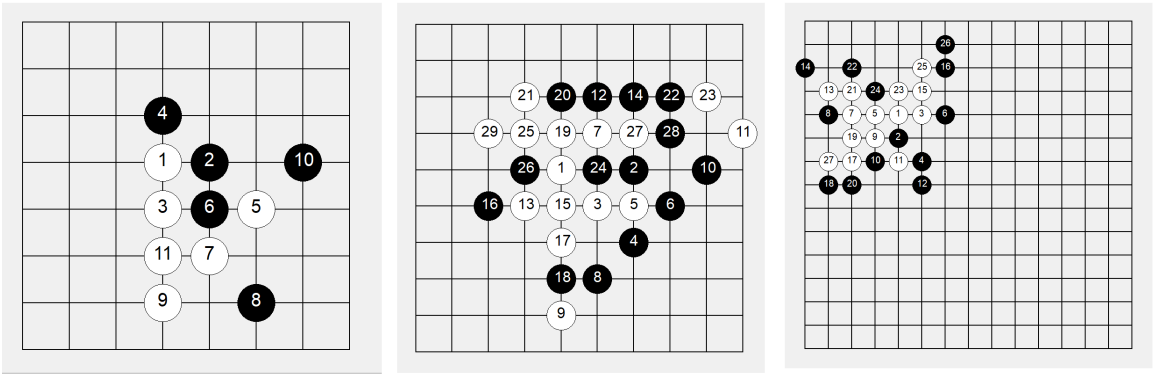


Figure 5: Good selfplay data with mmRoll

In addition, we also focus on training loss in whole process. Compared with pure reinforcement learning, loss for network with mmRoll decreases rapidly at first. At around 15 rounds, it almost converges to a stable loss (non-zero). So a natural question is why network works well even though the training loss converge to a non-zero value.

As shown by experiment, we believe there are following two significant reasons:

- **Equivalently Good Actions.** At certain states, networks is confident in some equivalently good actions rather than only one. But policy given by MCTS tends to converge to one of them, which leads to non-zero loss. In fig 6, some “equivalently good actions" learned by network are shown.

- **Short-Sighted Supervisor z.** Since we use a game exactly result z to be supervisor of all steps in whole game (flipping sign for each move), it is very short-sighted and with large variance. Because of randomness in selfplay, it is very likely to give different z for same state s. For example, in the state s with no piece at all, the average z may gives network a pretty good gradient direction, but loss will never converge to zero.

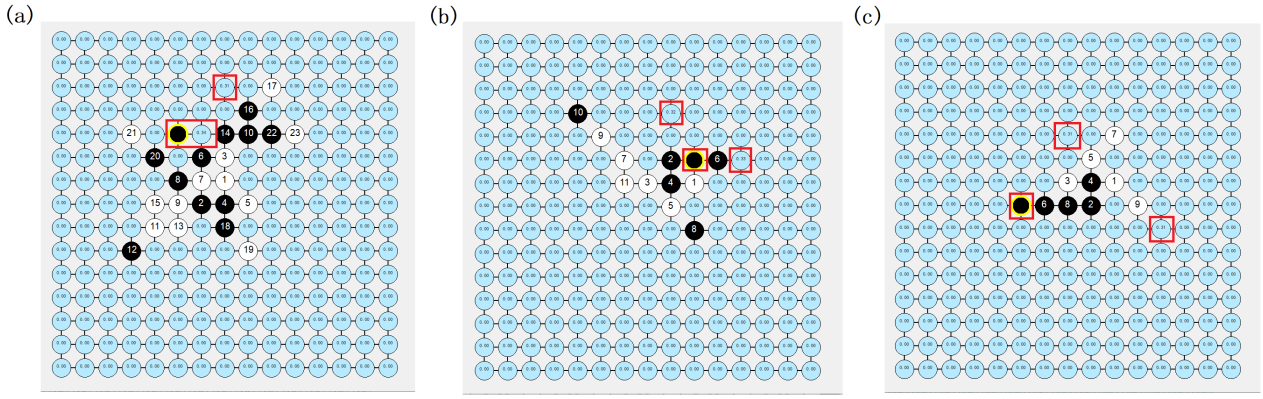


Figure 6: Equivalently Good Actions. The positions marked in red rectangle are equivalently good actions predicted by network.

5 Conclusion

5.1 Our Contribution

Proposed a Efficient Way to Train Intelligent Gomoku To solve Gomoku problem and barriers encountered in preliminary experiment, we proposed a new way to trained intelligence Gomoku agent. That is "Reinforcement Learning with Advisor" that combines traditional method and deep reinforcement learning method to solve Gomoku problem, and speed up the training process at the same time.

Complete Architecture and Training Pipeline. Instead of just focusing Gomoku problem, the entire architecture and training pipeline is scalable for different board games by substituting the game simulator part, approximated rollout function and first layer of network. In another word, the entire architecture and training pipeline is versatile.

5.2 Limitation

Despite the trained model has already learned amazing strategies, there are still some limitations of our methods.

Insufficient Sample by MCTS. Since limitation of computation power, 400 samples in MCTS is the maximal number we can afford, which is insufficient compared with large state space of Gomoku. Thus policy given by MCTS during training process is not strong and powerful enough. We only tried 800 samples in 8×8 board and it converged much faster than 400 samples version. So we believe if 800 or more samples are used, network is capable to learn much more fancy strategies.

Biased Rollout Function. Despite compared with pure network return, mmRoll provides much more information and better approximates the optimal rollout function R^*. However, it is still a biased rollout function that restrict the upper bound of our model.

5.3 Future Possible Work

Simulated Annealing Rollout Function. The crux of using approximated rollout function is to enable our model to explore less useless actions and converge faster. As network learns to play, we can decrease the its reliance to biased rollout function, say mmRoll, by decreasing combination weight with time. Then bias will decrease with respect to time and finally we will only adopt unbiased value estimation returned by network.

References

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

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

[3] Yuandong Tian, Jerry Ma, Qucheng Gong, Shubho Sengupta, Zhuoyuan Chen, James Pinkerton, and C Lawrence Zitnick. Elf opengo: An analysis and open reimplementation of alphazero. *arXiv preprint arXiv:1902.04522*, 2019.

[4] Louis Victor Allis et al. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.

[5] Rijul Nasa, Rishabh Didwania, Shubhranil Maji, and Vipul Kumar. Alpha-beta pruning in mini-max algorithm–an optimized approach for a connect-4 game. *Int. Res. J. Eng. Technol*, pages 1637–1641, 2018.

[6] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[7] George C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2):179–196, 1979.

[8] Han Liao. New heuristic algorithm to improve the minimax for gomoku artificial intelligence. 2019.

[9] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[10] Ingo Althöfer. Game self-play with pure monte carlo: the basin structure. *Friedrich-Schiller Univ., Jena, Tech. Rep*, 2010.

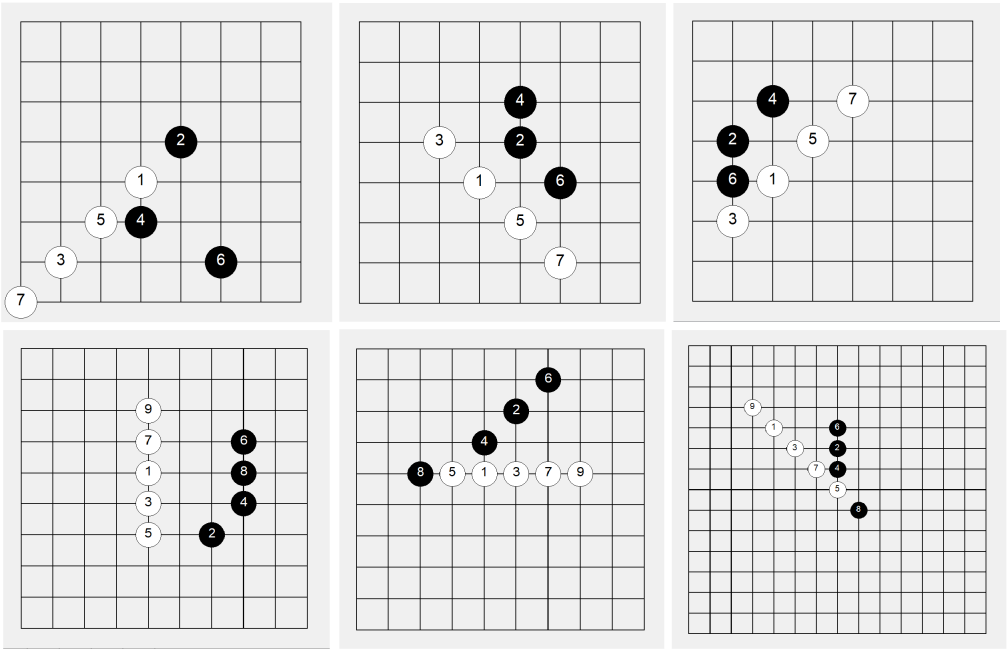
[11] Zheng Xie, XingYu Fu, and JinYuan Yu. Alphagomoku: An alphago-based gomoku artificial intelligence using curriculum learning. *CoRR*, abs/1809.10595, 2018.

[12] Rongxiao Zhang. Convolutional and recurrent neural network for gomoku, 2016.

[13] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

Appendix

5.4 Collapsed Selfplay



5.5 Redundant Selfplay