Final Report For Gomoku Project

1st Keyu Mao
School of Data Science
Fudan University
220 Handan Road, Shanghai, China 200433
20307130241@fudan.edu.cn

2nd Jianzhi Shen
School of Data Science
Fudan University
220 Handan Road, Shanghai, China 200433
20307130030@fudan.edu.cn

***Abstract*—In this project, we build a DQN model which links the state and the Q value in Q-learning with a deep neural network. Besides, the effect of experience replay is tested. We improve the Minimax agent with Threat Space Search and Principal Variation Search, which will be elaborated later. We try a comprehensive methods named HMCTS. The performance of each agent is recorded, and we choose the best, Minimax with TSS, as the final upload.**

***Index Terms*—Gomoku, Alpha-Beta pruning, VCF VCT, PVC, Q-learning, Experience Replay**

I. Introduction

Gomoku, which is also called Five in a Row, is an abstract strategy board game. It is traditionally played with Go pieces (black and white chess piece) on a certain board. In the game, 2 players will place chess piece alternatively until the game reach a terminate state, which is typically formed by 5(sometimes more than 5) continuous chess pieces in a same color.

Given gomoku's rather simple rule but complex situation in real games, many algorithms have been developed to solve gomoku problem. Nowadayw, artificial intelligence agent can beat the top human player easily. In the last 2 reports, we managed to use Minimax search with Alpha-Beta pruning and Monte Carlo Tree search(which is based on UCT) to create AI agent and test the agent by making it fight against other different agents. In this final part, we will try to do some optimization to make Minimax search work more effiently and help agent to be more aggressive. Besides, we will try another kind of MCTS with heuristic application, which was taught by TA in class, and compare it with MCTS with UCT.

Ever since 2017, the advent of AlphaGo leads more people to focus on combining reinforcement learning with chess game. Reinforcement learning is a field of machine learning concerned with how intelligent agents should take actions in an environment in order to maximize the notion of cumulative reward. Adaptive dynamic programming is the problem of finding an optimal (or nearly optimal) control policy for a (discrete time) valuated stochastic process whose local rewards and transitions depend on unknown parameters(1). In the following section, we will introduce our trials on reinforcement learning, including failure final model based on ADP.

The origanization of this report can be decribed as:

- Reinforcement learning theory and trials
- Monte Carlo Tree search with heuristic application
- Optimization of Minimax search
- Experiment and results
- Conclusion and future work

II. Reinforcement learning

A. Basic Idea of Q-learning

In this section, we will try to solve the gomoku problem based on some basic ideas in reinforcement learning. As it's obvious that gomoku game doesn't have an explicit probability transition matrix ,which can be used to construct Markov decision process, and an explicit reward function, we will apply some model-free methods to solve this problem.

We mainly learned 2 kinds of methods regarding model-free reinforcement in class: Temporal difference and Monte Carlo method. Temporal difference (TD) learning refers to a class of model-free reinforcement learning methods which learn by bootstrapping from the current estimate of the value function. These methods sample from the environment, like Monte Carlo methods, and perform updates based on current estimates, like dynamic programming methods.(2)

While Monte Carlo methods only adjust their estimates once the final outcome is known, TD methods adjust predictions to match later, more accurate, predictions about the future before the final outcome is known.(3)

There are many algorithms use the idea of TD to solve the relative reinforcement problems, among them SARSA and Q-learning turn to be the most well-known ones. In our experiment, we mainly consider Q-learning and will try some Value Function Approximation methods to make Q more efficient and help the agent to generalize in some degree.

The key equation in Q-learning explains how q value updates through each sample(or each transition) obtained by agent:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha((r_t + \gamma \max_{a'} Q(s_{t+1}, a'))) \tag{1}$$

B. Feature extractor

As we mentioned in last subsection, we will use some Value Function Approximation to generalize the state and help the agent work more efficiently. In that case, it's essential for us to develop a method to extract features from the state so we can use those features to evaluate the state.

In gomoku, patterns are of vital importance to players. The number of different patterns can help in some degree reflect the utility of each player and complex patterns can be more likely to lead the agent to winning state. In middle term report, we made use of the number of all types of pattern to create utility function and that strategy made sense in Minimax search. In this section, we decide to apply that idea to reinforcement learning.

We first choose 7 kinds of ”important” and ”typical” patterns and record the number of them in a list(both our and opponent’s patterns, so there are 14 in total). Considering the fact that in gomoku who’s turn is also a significant factor, we should also add the order of roles to the features. For each pattern, we use 4 number to describe. For instance, the first position denotes the number of pattern 1 of our chess, the second denotes the number of pattern 2 of opponent, then the third denotes present role, 4th denotes the role of next turn. After all the patterns, we then add both roles again to strengthen the importance of order. Finally the length of features should be 4*7+2 = 30.

C. *Simple trial of neural network*

After we decide the algorithm and choose a reasonable feature extractor, we can have a simple trial to solve gomoku based on Q-learning. In class, we learn to use linear combination of features to do evaluation, but we don’t think linear combination can solve gomoku considering the complexity of gomoku. On the other hand, Neural Network may deserve a try.

As we all know, neural network can be explained as a combination of non-linear functions, so it can be applied to many varieties of problems. The most simple idea is to do some modification from linear one. To be specific, we can construct a 2-layer Neural Network, then the network can predict the win rate of present state based on input features. The loss function of network just stays the same as linear version, namely MSE(if we can view the problem as a kind of binary-classification, we can also use BCE loss function):

$$J(w) = E_\pi[(r_{t+1} + \gamma \max_{a'} V(s_{t+1}, a')) - V(s_t, a)] \qquad (2)$$

What we need to do is to recompute the gradients and do backpropagation to update weights and bias. The directory ’try’ display example code to do such job. In the experiment, we set input size as (1, 30), which is same as the dimension of feature. The size of hidden layer is (16, 16). The whole network takes after figure 1 below.

Nevertheless, the effect of the network is terrible, and we realize that simple structure cannot make sense when confronted complex problems like gomoku.

D. *Self-teaching ADP for Gomoku based on existing idea*

Given the conclusion we mentioned in last subsection, we decide to look for some relative work concerning reinforcement learning and gomoku. Due to the limitation of size of agent file, we cannot use some deep learning package like

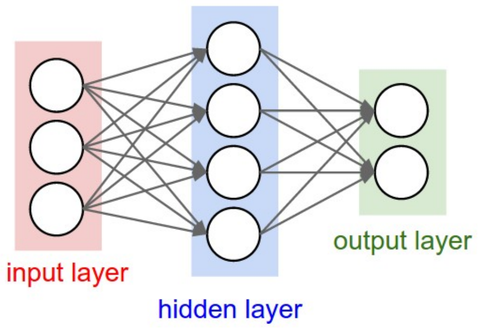

Fig. 1. Neural Network structure

pytorch. So we focus on the work published before 2014 and Zhao’s work attracts my attention(4).

According to Zhao’s work, 2 networks will work together to solve gomoku problem. To be specific, the critic network is a neural network which is used to evaluate board situations. The action network is not a neural network, but it works together with the critic network to determine an action. Likewise, Zhao also try to use patterns and role to create features from present board but they carefully choose 20 patterns. Given that they create their neural network with input size as 274, hidden layer size as 100 and output size as 1. The output of the neural network means the winning probability of player 1 starting from a board situation. The mechanics of the combination of 2 networks can be described like figure 2.The topological structure of their critical network is similar to ours which has been shown in 1.

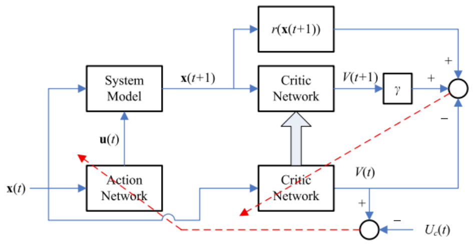

Fig. 2. Network Mechanics

In the experiment, we won’t abandon our feature extractor and choose to cover Zhao’s work. On the contrary, we would like to make use of their idea and net work structure, especially the combination of critic network and action network. It’s worth to be mentioned that they didn’y use the idea of Q-learning like what we’ve in last subsection, which means that their error function will be slightly different. To be exactly, when it comes to the computation of error, they won’t consider the best value given different actions and the specific form of error in each iteration can denoted as:

$$e(t) = \alpha[(r_{t+1} + \gamma V(s_{t+1}, a')) - V(s_t, a)] \qquad (3)$$

$$E(t) = \frac{1}{2}e^2(t) \qquad (4)$$

Where α is the learning rate of critical network and γ is the discount factor to decrease the influence of the value in the future state. For Gomoku, a reasonable move is often near a position which has been occupied, which has been explained in our previous report. Given that, we can carefully change our candidate action on the present board. Besides positions near occupied grids, we may also add some heuristic function to help judge some promising actions. Some previous heuristic functions have been used to help agent make wiser choice in the game. During the training process, when there are several alternative actions which have equally high evaluation, we can simply choose the one that is last found, helping reduce the space to store some extra states.

To handle the exploration–exploitation dilemma, we may apply e-greedy policy to train the agent. Namely we can let both players select moves according to following e-greedy policy, which is defined by:

$$a(t) = \begin{cases} \arg\max\limits_{a} V(t+1) & \text{with probability} \quad 1-\epsilon \\ \text{random action} & \text{with probability} \quad \epsilon \end{cases}$$

e-greedy policy ensure that we can mostly follow the presently best policy, and meantime we can also explore some new action in case that many states may not be visited until the convergence of the algorithm.

E. Experience Replay

In sequential Q-learning update, the transition samples (s_t, a_t, s_{t+1}, r_t) are dependent, which is a handicap to the convergence. Therefore, the DQN(5) maintains an experience buffer to store the transition samples while updating, and after a batch (5000 traces for example) is learned, the model will go through a random permutation of the stored transitions, in order to improve independence and accelerate convergence. The effect is shown in the Experiment and Results Section.

III. Heuristic MCTS

A. Heuristic Application

MCTS introduces randomness into tree search. The heuristic utility of a node is related to the empirical frequency of winning playouts of its successors. In the second report of midterm, we manage to carry out MCTS for gomoku. In our previous work, we mainly use the fundamental principle of MCTS and use UCT as the criteria to choose the best nodes. Such simulation can perform very well if we can generate virtual games for enough times and it has been proven that MCTS is an estimate of Minimax(6), so it can handle huge branching factor without the help of an explicit utility evaluation.

However, in this gomoku project, time is strictly limited and the efficiency of original MCTS won't make much sense during the test. That's because due to the lack of generated times, the whole tree may not expand very well and cannot get a reasonable move in the end. We've been applying some heuristic functions to MCTS in the midterm project, and those functions just remind us of a simpler algorithm: Heuristic MCTS, the origin of UCT. Heuristic MCTS treat every possible actions equally, and simulate them for same times. On the contrary , UCT treat different possible actions unfairly, so it can simulate different actions giving different time. UCT can do so well at the trade-off between exploration and exploitation that in most cases UCT perform much better than Heuristic MCTS. Nevertheless, in the project, Heuristic MCTS also deserve a try due some special factors like time-limitation.

B. Performance evaluation

The following Pseudo code will show the basic structure of Heuristic MCTS. In our cases, Heuristic MCTS have better ability to reach a terminate state than original MCTS, that help Heuristic MCTS to deal with some simple situation more efficiently. Additionally, since we also eqip the agent with some heuristic knowledge concerning threatening patterns, the agent can reduce the risk which is brought by the randomness of MCTS. However, the effect is not expected to be great due to the simple structure of the algorithm. From our perspective, we suppose Heuristic MCTS can perform slightly better than MCTS when confronting some weak agents but worse when opponent is well-performed agent.

IV. Optimization in Minimax search

A. Threat Space Search

1) Definition of threat: As we all know, in gomoku patterns play an essential role can determine the result of game. The number of different patterns and the complexity of patterns will make a difference on the board. In that case, both players should concentrate on patterns throughout the game. Some patterns with extremely threatening effect on the board can be recognized as threats, which should be dealt with immediately and can force the opponent to play a specific set of limited options. A winning strategy consists of a series of threats leading to a forced win.

Given facts mentioned above, it's really important to spare some time to focus on threats. And we suppose that if we can use Minimax search to do threat space search, we can easily find whether we can construct some complex patterns in finite future rounds. To achieve that goal, we have to give the strict definition of threats first.

We introduce the threats to develop an intuition and to help us understand the concept of threat space search more intuitively in the following figure 3.

Then let's talk with some details concerning threats in figure 3:

- The four: a line of 5 squares, where 4 are occupied by the attacker, and the fifth square is empty(Also named sleep four in our past report)
- The straight four : a line of 6 squares where the attacker occupies the 4 grids in the center, and the rest of squares are empty(Also named live four in our past report)
- The three:

Algorithm 1 Heuristic MCTS

```
 Input: current state [1]
Initiate actions,flag←heuristic_actions(state)
Initiate scores←0 for all actions
while within time and iteration limit do
   if flag is True then return first action in actions
   else
      for action in actions do
         state←MAKE_MOVE(state,action)
         reward←Simulate(state)
         scores[action]←scores[action]+reward
         state←DRAW_MOVE(state,action)
Output: action with max(scores)

function SIMULATE(state)
   actions, flag←heuristic_actions(state)
   if flag is True then
      action ← first action in actions
      state←MAKE_MOVE(state,action)
   while state is not terminal do
      action←RANDOM_CHOOSE(actions)
      state←MAKE_MOVE(state,action)
   if Win then
      return 1
   else if Draw then
      return 0.5
   else
      return 0

function HEURISTIC_ACTIONS(state)
   actions ← possible actions that have neighbors
   if Threaten patterns are included in actions then
      return most threatening actions, True
   else
      return actions,False
```

1) a pattern which is composed of a line of seven squares where attacker occupies 3 grids in the center with the remaining 4 squares empty.
2) a line of 6 squares with any 3 of the center 4 squares occupied by the attacker.(Also named live three)

- The broken three: a line of 6 squares where the attacker has occupied 3 non-consecutive squares of the center 4 with the remaining squares remaining empty(Also a kind of live three)

Threats mentioned above is really of vital importance since the next move will turn to the terminate state and determine the winner. Besides, The Three should be considered since it can be easily formed and lead to a straight four in the next turn. Those threats will make sense in our threat space search.

2) Threat space search: In this subsection, a substantial reduction comes from the determination of the defensive moves connected to a threat. Such reduction is made based on some expertise in realistic gomoku game. First we consider a commonsense in gomoku: after a four, one defensive move is possible; after a three, two or three are possible. If we consider, for instance, after a three all possible countermoves, the number of variations rapidly increases. Nevertheless, it appears that human experts mostly find winning sequences in which the choices by the opponent are irrelevant, which was concluded in many previous games that human expert s fight against AI agents. This idea can help us to reduce searching space during our Minimax search.

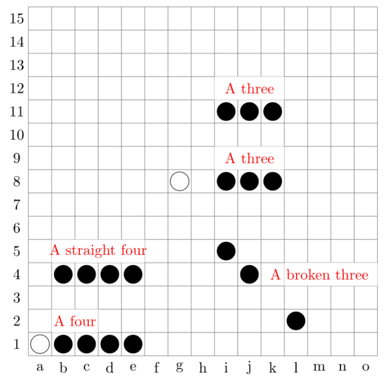

Fig. 3. Threat types

In TSS, we focus only on attacking play and try to ignore all defensive moves. If more than one defensive move is possible, then we treat each move as if it had been played simultaneously. When we manage to find a reasonable way to construct threat sequence, we then should get a deep insight into it and check whether the sequence can survive no matter what defensive moves will be made by opponent.

The main idea of TSS is to mimic human experts. In reality, human experts will efficiently discard the move which can construct threat but can be blocked in the halfway simply. By considering this expertise, our agent choose the strategy to identify threat sequences that are likely to lead to an absolute victory and then further verify those ”victories”.

3) Example of TSS and space reduction analysis: Define that Move means the movement taken by attacker, and Against means the movement list of opponent, which will be used to block the threat. A typical example is figure 4.

The following table I will show the exact procedure of TSS in the above example.

TABLE I
TSS PROCESS IN EXAMPLE

Depth	Threat Type	Move	Against
1	Four	l15	k15
1	Four	k15	l15
1	Four	o12	o11
1	Four	o11	o12
1	Three	j5	j4,j8,j9
2	Four	f15	i4
2	Four	e15	f15
3	Five	e15	None

The corner on the top right of figure 3 help to construct 4

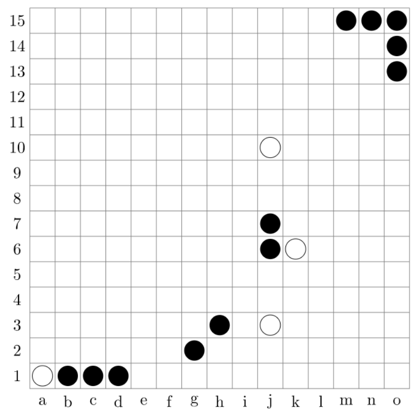

Fig. 4. An example board for TSS

threat trees with depth 1. Those choices are not promising and should be abandon as early as possible.

Besides, the first 2 rows show that there are many symmetric cases during TSS process. And Trees in those cases are in conflict since our Move turns to be opponent's Against in another tree.

In the bottom of the table I, we get to a winning state with searching depth as 3. This informs us of the key factor that we don't need to consider exact opponent responses to our moves in TSS. And this heuristic thinking also helps us to reduce searching space a great deal during TSS process.

B. *Principal Variation Search*

PVS, or Scout(7), is an optimization which further exploits the accelerating effect of successor sorting. After the first successor, as most locally rewarding, is checked, we use the null window $(\alpha, \alpha+1)$ to check its siblings, where α is now the value of the first successor, (or for min layer, $(\beta-1, \beta)$) . If a sibling turns out to be more rewarding, then the returned value will be $\alpha+1$, and then we should recheck it with the new window $(\alpha+1, \beta)$, in case it has a higher outcome.

Since in most cases, the most locally rewarding successor is the best choice, PVS will improve the pruning. Even if it is not the case, the extra searching time in window $(\alpha, \alpha+1)$ is relatively small. So PVS will accelerate the search, as the result shows in the Experiment and Results Section.

V. Experiment and results

A. *The performance of Heuristic MCTS*

In the third part of the report, we introduce the principle of Heuristic MCTS, and discuss the probable performance of it in the test. In this subsection, we will test the algorithm and compare it with MCTS with UCT and Minimax search without optimization, which have been realized in past, and evaluate the algorithm.

The results have been shown in table II. We can simply conclude that the overall performance of Heuristic MCTS is much worse than both UCT and Minimax search without optimization. That's beacause the whole idea of Heuristic

TABLE II
Performance of HMCTS

Algorithm	5row	mushR	pisq7	pureR	sparkle	valkyR	rate
Minimax	7:5	12:0	5:7	3:9	2:10	7:5	0.5
UCT	9:3	9:3	1:11	8:4	0:12	3:9	0.42
HMCTS	4:8	12:0	0:12	2:10	0:12	1:11	0.26

MCTS is too simple to figure out reasonable solutions in some complex situations when there are plenty of patterns on the board. However, for simpler circumstance, HMCTS does can make choice more rapidly and end the game more likely than UCT. And that's why HMCTS can win more games when facing rather weaker agents like mushroom.

B. *The experiment of RL and TSS*

In the following tabIII, we will show all the results of algorithm which have been carried out in this report and compare them with each other. The list of algorithms consist of ADP, ADP with the idea of experience replay, Heuristic MCTS, and Minimax with TSS.

TABLE III
All competing results

Algorithm	5row	mushR	pisq7	pureR	sparkle	valkyR	rate
ADP	3:9	6:6	0:12	3:9	1:11	4:8	0.24
HMCTS	4:8	12:0	0:12	2:10	0:12	1:11	0.26
ADP-ER	5:7	6:6	1:11	5:7	0:12	4:8	0.29
Minimax	7:5	12:0	5:7	3:9	2:10	7:5	0.5
TSS	8:4	12:0	8:4	9:3	3:9	9:3	0.68

As we can see in the above table, reinforcement learning doesn't show very great results. The reasons can be divided into several parts:

1) First, we didn't establish great features which can splendidly describe the situation of present board. Only role and patterns are not sufficient.
2) Second, training times is limited so that the weights may not converge to best.
3) Third, since we initiate weights and bias randomly and set number for hyper-parameters personally, the traning process may not go smoothly. It a better chioce to use a pre-trained weights and use cross validation to find best hyper-parameters.

On the other hand, experience replay indeed helps the model to gain better weights and biases. In other words, previous experience, randomly sampled, can help the model to converge faster and perform better in the competing test.

For the part of TSS, we simply find out that TSS(based on Minimax and using alpha-beta pruning) does best job among all the algorithms. TSS can help the agent to find threatening patterns much earlier than simple Minimax and then make a strategy to win the game. It shows that when facing some excellent agents like PureRocky, Minimax with TSS can have high chance to take over the game. But on the contrary, since TSS will search for extra threats, it will surely take more time than Minimax without TSS.

C. *Effect of PVS*

This experiment tests the accelerating effect of PVS. In the experiment we build two agents, one using regular Alpha-Beta pruning, and the other using PVS. Without TSS, the two agents are set the same truncate depth 6 (to make the distinction clearer), and battle each other with on Piskvork. The result are depicted in figure 5 and 6:

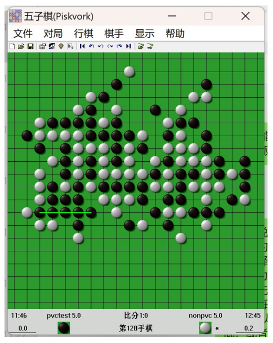
五子棋(Piskvork)
文件 对局 行棋 棋手 显示 帮助
11:46 pvctest 5.0 比分1:0 nonpvc 5.0 12:45
0.0 第128手棋 * 0.2

Fig. 5. Alpha-Beta, depth=6

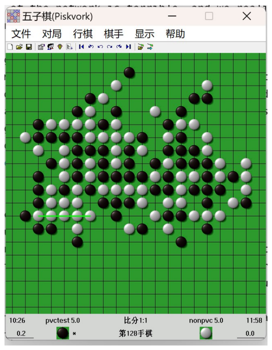
五子棋(Piskvork)
文件 对局 行棋 棋手 显示 帮助
10:26 pvctest 5.0 比分1:1 nonpvc 5.0 11:58
0.2 * 第128手棋 0.0

Fig. 6. PVS offensive, depth=6

As can be seen in the figures, the result action of both agents are identical, and no matter who is offensive, PVS runs faster than regular Alpha-Beta pruning. Although the improvement is small, the slight decrease of running time will increase the probability of winning a time-restricted game.

D. *Time consumption of finding threat*

In TSS, we mainly derive 2 strategies when searching for threats on board. For the first strategy, we will consider using Sleep Four and Live Three to attack when it's our turn, and choose to consider whether opponent will choose to attack instead of defending or do a pre-defensive move to get rid of potential threats in the future. For the second strategy, we only consider aggressive movements on both sides. Clearly, the first strategy consider the situation more carefully but will surely take more time.

Furthermore, In our middle report we discuss the use of Zobrist hashing. It occurs to us that such thinking can also be applied when searching threats since the idea of Zobrist hashing can help us to use XOR to compute board state and draw move in a rapid speed.

We test 2 strategies and the idea of Zobirst hashing by fighting against our previous agent based on Mimimax. And the results are shown in figure 7. In conclusion, the depth that we set to search for threats determint the time consumption linearly. And just as our discussion, strategy 1 is slower than strategy 2 regradless of the application of Zobrist hashing. The Zobrist hashing can in some degree accelerate the searching process but the effect is not significant enough. The red line in the plot denotes the final version we choose to use.

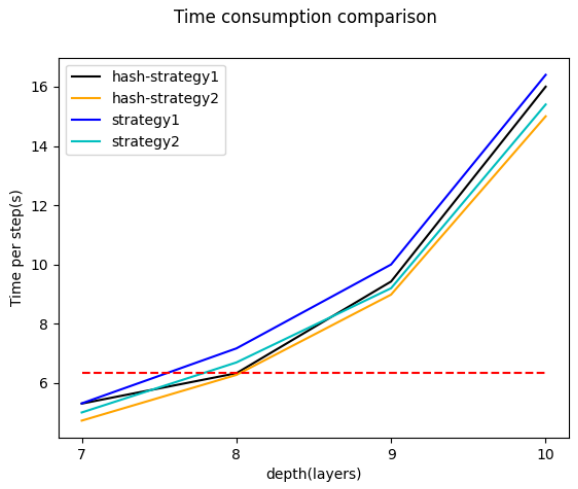

Fig. 7. Time consumption test

VI. Conclusion and Future Work

In this project, we successfully carry out DQN model which links the state and the Q value in Q-learning with a deep neural network, try heuristic application in MCTS, and finally optimize our Minimax search algorithm with TSS idea. We find that our agent based on optimized Minimax perform best in the test. Besides, we also do a series of experiment and examine the performance and time consumption of several algorithms.

For future work, after fully understanding the principle of neural network, we can construct a better DQN model, using CNN to efficiently fit the value considering move history. AlphaZero uses MCTS during the training process to replace reward as winning rate in each state, so we can also try comprehensive application of different methods. In fact, we have already tried several combinations like HMCTS, and further exploration is expected.

References

[1] G. Hübner, "Adaptive dynamic programming," *OPTIMIZATION AND OPERATIONS RESEARCH–Volume IV*, p. 119, 2009.

[2] R. Sutton and A. Barto, "Reinforcement learning: An introduction (adaptive computation and machine learning)," *ieee transactions on neural networks*, 1998.

[3] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[4] D. Zhao, Z. Zhang, and Y. Dai, "Self-teaching adaptive dynamic programming for gomoku," *Neurocomputing*, vol. 78, no. 1, pp. 23–29, 2012.

[5] M. Fang, Y. Li, and T. Cohn, "Learning how to active learn: A deep reinforcement learning approach," pp. 595–605, 01 2017.

[6] F. Chelaru and L. Ciortuz, "Combining old-fashioned computer go with monte carlo go," pp. 216 – 222, 10 2008.

[7] J. Pearl, "Scout: A simple game-searching algorithm with proven optimal properties," in *Proceedings of the First AAAI Conference on Artificial Intelligence*, AAAI'80, p. 143–145, AAAI Press, 1980.