DATA130008

Introduction to Artificial Intelligence

Final Report

BoYuan Yao 19307110202
LeiRu Long 19307130350
2021-12-19

Abstract In the last two checkpoints, we have already implemented Alpha-Beta Pruning algorithm and MCTS algorithm on Gomoku AI. After learning some basic concept of reinforcement learning, we constructed a prototype AI using the idea of QLearning. We also improved our Alpha-Beta Pruning AI for better performance.

1 Introduction

Reinforcement learning has become a hot topic in recent years. Agent could learn the policy of given environment by interacting with the given environment. With the aids of neural network, reinforcement learning could achieve better performance (e.g. AlphaGo Zero). We have learned some Model-Based and Model-Free learning policies on class. So we implemented a QLearning-based AI to enhance our understanding of the lectures.
Moreover, as we learned after class, there are also some advanced methods of tranditional searching policy. The PVS (Principal Variation Search) is a good way to promote the pruning ability of Alpha-Beta Pruning. We also find several ways to shorten the searching time.
The organization of the report is as follows. In the next section we will give a brief description of QLearning and our implementation of reinforcement learning checkpoint. In the third section, we will introduce the improvement of our Alpha-Beta Pruning Algorithm. We will plot our experimental result in the following section and in the last one, we will conclude what we learned in the Gomoku project and some more future work.

2 Q-Learning AI

2.1 Q-Learning

The key idea of Q-Learning is, maintaining the state-action Q estimates and use the value of the best future action for bootstrap update. It's a famous off policy startegy, with the following iteratiom

$$\begin{aligned} &Q(s_t, a_t) = Q(s_t, a_t) + \alpha\delta_t \\ &\delta_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \end{aligned}$$

Where δ_t is known as TD error. This off-policy learning strategy really fit our target, as training the Gomoku AI is costly (though mush easier than Go) and, we need historical data from old polices (i.e. the information of former training derive by the old agent).

2.2 Sigmoid

Sigmoid function is a famous function used in neural network. As we know, without sigmoid function, neural network just contain several matrix operations, which means it could only represent the linear model. But with the sigmoid function, neural network gains the ability to represent nonlinear information, which is important beacuase most of the real-life models involve a lot of nonlinear relations. Moreover, sigmoid function is a good transformation of value in interval $(-\infty, +\infty)$ to interval $(0, 1)$, which provides a good way for evaluation-to-win-rate transform. It has the following form

$$S(t) = \frac{1}{1 + e^{-t}}$$

We could attain a simple property, that $S'(t) = S(t)(1-S(t))$, meaning that we could easily attain the gradient of function S at t just using $S(t)$.

2.3 Q-value

With the idea of sigmoid function, we could transform our evaluation to a winning rate. So in our RL AI, we compute the Q-value with the following equation

$$\text{Q-value} = \frac{1}{1 + e^{-\beta s(f)}}$$

Where $s(f)$ stands for the score of the given features, and β stands for a shrink coefficient to avoid the extremely small difference of the value of sigmoid function with efficiently large (w.r.t absolute value) of x.

2.4 Feature Extractor

We inherent the feature of the last Alpha-Beta Pruning AI, i.e. scanning the four directions of a empty point and extract the pattern inside it. With all these features, our AI could return a Q-value for each action.

2.5 Root Search

We inherent the tree search process in MCTS, using the following tree policy

```
Algorithm 1 Tree Policy
Input: root state $s_0$
Output: selected son of root state candidate
1: Return candidate with biggest Q-value
```

Notice that if a *candidate* hasn't been explored, the Q-value of this son node is 1, so that every active son of the current root state will be explored.

2.6 Feedback

Adopting the feedback structure in Project-3, the feedback module deliverying the impact of reward at leaf node (as we attain the outcome of the game) all the way back to root node. As we using sigmoid, the iteration is as follows

$$weight(f_{t+1}) = weight(f_t) + \alpha\delta_t v\beta(1-Q)Q$$

Where the δ_t is the TD error in 2.1, α is the learning rate, β is a shrink coefficient to avoid the extremely small difference of the value of sigmoid function with efficiently large x, and v is the number of the given feature f.

2.7 Simulation

Inherenting the simulation in our MCTS AI, the simulation policy involves ϵ-greedy to balance the exploration and exploitation and generate enough data for learning.

2.8 Training Policy

We apply the following training policy

```
Algorithm 2 Training policy
Input: initial policy
Output: new policy
1: new policy ← initial policy
2: while Doesn't meet stopping criteria do
3:     while new policy could not beat initial policy do
4:         generate data using self-play method with initial policy
5:         update new policy with the data
6:     end while
7:     initial policy ← new policy
8:     Check stopping criteria
9: end while
```

The idea is natural selection, killing the old AI with old policy ,updating the fighting policy of data generation group and pushing the AI to find the stronger policy.

3 Improvement of Alpha-Beta Pruning AI

The original Alpha-Beta pruning stratgy could not meet the standard of high peformance searching, with the help of PVS, iterating deepening search, moves pruning, mapping technics we could achieve better performance. The basic idea of our work is based on the AI WINE.

3.1 PVS

PVS (Principal Variation Search) is an improved version of origin Alpha-Beta algorithm. As we learned in Alpha-Beta Pruning, we know that the meaning of "Alpha-Beta" is, in fact, a window for searching. As the searching proceeding, the window will shrink and we will get an exact value for root node. In fact we could divide searching nodes into the following three kinds:

- Alpha node. In alpha node, every branch will return a value lower than current Alpha.
- Beta node. In beta node, at least one branch will return a value bigger than Beta.
- PV node. In exact node, every branch will return a value between interval $(Alpha, Beta)$

We could see that Alpha-Beta Pruning will prune the subtree with root falls into the first two kind. But for PV node, it takes more time for searching beacuse we can not apply any kinds of pruning. So our wish is, we could have a window with zero width, i.e. we have the "midpoint" of all the son nodes, so that all the nodes belongs to Alpha node or Beta node, given an exact value for root node with high efficiency.
So in implementation, we suppose we have already found that sweat point, and deploy an Alpha-Beta search of zero-width window to prove our assumption. If we get a value satisfies $val \leq alpha$, then we have found the best evaluation $alpha$, otherwise our assumption fail, we need to do one step Alpha-Beta Pruning with normal-width searching window.
Though we might waste sometime on the attempt of zero-width window searching, we gain bigger edge when the assumption works, easily promoting the pruning ability of the AI.

Algorithm 3 Alpha-Beta pruning with PVS

```
Input: $\alpha, \beta, root$
Output: Exact value for $root\ val$
 1: while $root$ has kids do
 2:     if not the first kid and $\alpha + 1 < \beta$ then
 3:         $val \leftarrow -AlphaBeta(-\alpha - 1, -\alpha)$
 4:         if $val$ in interval $(\alpha, \beta)$ then
 5:             $val \leftarrow -AlphaBeta(-\beta, -\alpha)$
 6:         end if
 7:     end if
 8:     Update $alpha$ if needed
 9: end while
10: Return $val$
```

3.2 Iterating deepening search

Considering the limited time we have on the Gomoku game, iterating deepening search is a great method for time management. As we can not find the whole story of one game just for one Alpha-Beta search (explonential work !!), we just iteratively deepening the searching depth so that we could find better answer if we have time for more searching layers, and find a proximal solution without violating the time limits if we don't have that much time. To maximize the power of iterating deepening search, we need the help of zobrist hash.

3.3 Zobrist Hash

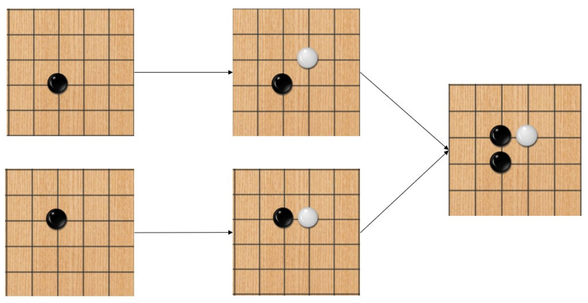


Figure 1: Why using zobrist hash

As we conclude in the first report, zobrist hash is a good way to store all the expanded move. But due to the bad data structure in our last AI, it is risky to use zobrist hash and it consume a lot of time. This time we reform our AI and apply the zobrist hash to our program. Not only did we apply the zobrist hash on move update, but we also deployed it on the Alpha-Beta search process, storing the some key information of each node, which could be a great boosting for the performance of iterating deepening search.
In our implementation, when the AI expand next move, it will first searching the $pvsTable$, to search for the best move recorded during the Alpha-Beta process, otherwise it will generate all the moves possible. We will record the node information in the *hashtable*, and whenever the Alpha-Beta process or iterating deepening search use the recorded node information (with depth not less than current depth), it could reuse it to avoid searching. Which means the cost of iterating deepening search will be minimized as it reuse the information of former searched layers.

3.4 Moves Pruning

When generating the moves in our last Alpha-Beta AI, we simply generate all the moves and sort them in descending order. And for the move which is realy important, we just using the function *seek_must* at the very begining to find the winning moves or defending moves to avoid losing the game. But it is really short-sighted. Therefore, in this AI, we applyed moves pruning on each moves generation, it works as follows

Algorithm 4 Moves pruning

```
Input: moves list cand
Output: pruned move list cand
1: if best move could form at least FLEX4 then
2:     Return the very first move of cand
3: else if The opponents has FLEX3 then
4:     Find the oppnents move which could form FLEX4
5:     Find the move for both sides which could form BLOCK4
6:     Return the above moves
7: end if
8: Return origin cand
```

Where the renewed pattern table is as follows:

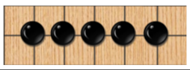
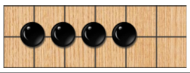
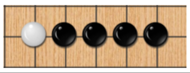
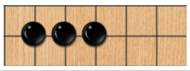
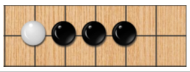
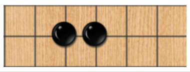
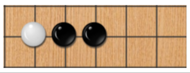
Pattern Type	Definition	Value
WIN		1200
FLEX4		800
BLOCK4		144
FLEX3		96
BLOCK3		18
FLEX2		12
BLOCK2		2

Table 1: Score For Each Pattern

3.5 Mapping technics

In our first Alpha-Beta Pruning AI, the scanning process occurs each time we update the score of the table. With a lot of optimization, the updating process did cost less time than before. However, there are better solutions for the evaluating and updating process. When initialization the basic structure, we simply construct a table for each pattern, and to seek for pattern for a certain point and in a certain direction, all we need to do is derive a key using the information of the neighbors. Here is a simple example of how we manage to do it.

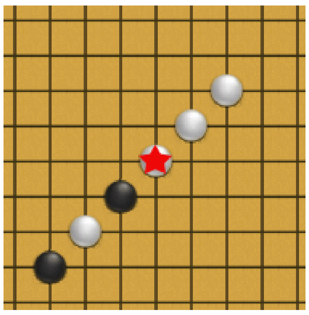


Figure 2: Example for mapping technics

To evaluate the point with red star in the reverse diagonal direction, we could derive a key using XOR operation. In this case, assuming in the AI module we have 0 stands for white piece ,1 stands for black piece and 2 stands for empty place, then the key (in one direction) is

$$key = 2\hat{}(1 << 2)\hat{}(0 << 4)\hat{}(1 << 6)\hat{}(0 << 8)\hat{}(0 << 10)\hat{}(2 << 12)\hat{}(2 << 14)$$

Where ^ stands for bit XOR and << stands for the shift operation.
And with the given key, we could check the pattern in the initialized structure *pattern_table* in my AI. Which is much faster than before, avoiding all the scanning operation in between the searching process, but finishing all of them at a table at initialization, saving tons of time. Moreover, we even stores the pattern value for various of cases in the structure *pval_table*, which furture optimize the performance of the AI.

4 Experimental Result

4.1 Q-Learning Result

The initial weights and the learned whights (4000 rounds of learning) are as follows:

–	initial	learned
WIN	600	599.9297
FLEX4	400	399.5267
BLOCK4	150	146.6131
FLEX3	70	64.4071
BLOCK3	20	16.7302
FLEX2	10	1.1991
BLOCK2	2	−4.4021
FLEX1	10	−22.9923
BLOCK1	0	−17.2641

With the learned policy, we test the agent on our computer with some AIs, and here is the result

–	abpruning
pisq7	2 : 10
valkyrie	8 : 4
mushroom	10 : 2
fiverow	12 : 0

The learned policy works pretty well.

4.2 Alpha-Beta Pruning Result

Setting the maximum searching depth 8 and maximum branch of each layer 8, we attain the following result on ftp

–	abpruning		abpruning
pisq7	12 : 0	Yixin2018	1 : 11
Noesis	9 : 3	Eulring	3 : 9
Wine18	1 : 11	valkyrie	12 : 0
Sparkle	9 : 3	fiverow	12 : 0
zetor2017	7 : 5	PureRocky	12 : 0
pela	1 : 11	mushroom	12 : 0

The final Elo rating for our AI is 1666.

5 Conclusion and Future Work

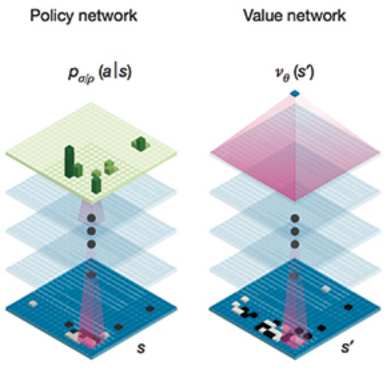


Figure 3: AlphaGo Networks

With weeks of hard work, our AI really improved a lot. But with the limitation of background knowledge and time (also the performance of python is not that good), there are still a lot of improvements could be done. For tranditional method, we are lack of the time to implement a powerful checkmate module, as we found that the last checkmate in our AI has some problems that could even make the result worse. And the combination of check-

mate and Alpha-Beta pruning is also a field worth trying, as checkmate is much faster than Alpha-Beta pruning (also more accurate).
For RL, as we searching online, we found a lot of AI using the deep learning technics to enhance there ability to express the variation on board. There are usually plenty of networks for different phases, e.g in AlphaGo, they have a policy network and value network to give policy and evaluate the win rate. With limited time and space constrain (we could not use numpy due to the 20M constraint), it is really hard and inefficient to construct the neural network and training them until they could work.

6 Acknowledgement

Thanks a lot for TA's patient tutorials, and in time reply when we met some problem runnning AI on the ftp folder.

References

[1] Zhentao Tang, Zhao, D., Kun Shao, Le Lv. (2016). ADP with MCTS algorithm for Gomoku. 2016 IEEE Symposium Series on Computational Intelligence (SSCI).

[2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, pp. 1-43, 2012.

[3] Chang, H.S., M.C. Fu, J. Hu, and S.I. Marcus. 2016. "Google Deep Mind's AlphaGo". OR/MS Today.

[4] L. Allis, H. Herik, and M. Huntjens. Go-moku and threat-space search. Computational Intelligence, 12, 10 1994.

[5] Pearl, Judea. "The solution for the branching factor of the alpha-beta pruning algorithm and its optimality." Communications of the ACM 25.8 (1982): 559-564.

[6] Albert Zobrist (1970). A New Hashing Method with Application for Game Playing. Technical Report, Computer Science Department, The University of Wisconsin, Madison, WI, USA.

[7] J. Lawrence Carter, Mark N. Wegman (1977). Universal classes of hash functions. STOC'77

[8] Tony Marsland, Murray Campbell. Parallel Search of Strongly Ordered Game Trees. ACM Comput. Surv. 1982, 14(4): 533-551.

[9] Lorenzo Piazzo ,Michele Scarpiniti ,Enzo Baccarelli. Gomoku: analysis of the game and of the player Wine. CoRR. 2021, abs/2111.01016.