Final Report for Gomoku Project

Zepeng Ding
School of Data Science
18307110088@fudan.edu.cn

Yuze Ge
School of Data Science
19307130176@fudan.edu.cn

Abstract

In this project, we first implemented the **Reinforcement Learning(RL)** for Gomoku AI, espacially use self-teaching Adaptive Dynamic Programming method(ADP). As to ADP, limited by the machine and training time, it doesn't perform well as expected, but we believe that more training time can make a smarter agent. Futhermore, we improved our agent with $\alpha - \beta$ pruning by adding some new technologies, such as **Threat-Space Search**. The agent performs much more smarter and quicker, and as a result, it has become our best-performing agent, with around 1620 rating score under Bayesian Elo.

1 Introduction

Gomoku, also known as five-in-a-row, is a popular board game. In this game, players alternate placing pieces of their color, either black or white, at the intersections of the board. The black player goes first.The winner is the first who forms a line of at least five adjacent pieces of his color, in horizontal, vertical or diagonal directions on the board. Such winning line is called five-in-a-row which is also referred to the name of the game Gomoku.

Reinforcement Learning(RL), which is used to describe and solve the problem of learning strategies by an intelligence (agent) during its interaction with the environment in order to maximize the reward or achieve a specific goal. A common model of reinforcement learning is the standard Markov decision process, which views learning as a trial-and-error evaluation process in which the agent selects an action to be used in the environment, and the environment accepts the action with a change in state.

Adaptive Dynamic Programming method (ADP), a temporal difference learning method, learns how to map an action to a state to get the maximal reward from interacting with the environment. Zhao[1] developed a self-teaching adaptive dynamic programming for Gomoku in 2012, This method use a critical network to evaluate the board situation and use the pre-trained network to choose the best move. It's so-called **deep reinforcement learning**.

It is well known that we generally use deep learning methods as a complex function mapping, and the fitting from states to Q and V values in reinforcement learning can be performed by deep learning networks. Similarly, in deep reinforcement learning, we also divide into value iteration and policy iteration, and converge to the best-fit function for each state-value pair through deep learning networks.

Organization for the following report:

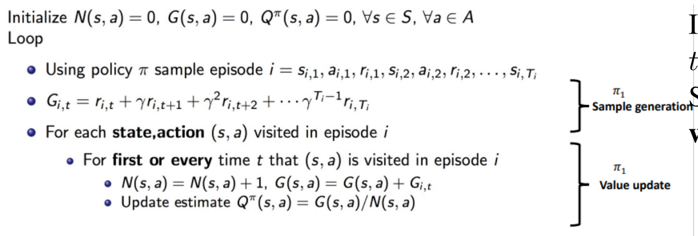
- **Section 2** Basic Ideas of Reinforcement Learning
- **Section 3** ADP for Gomoku
- **Section 4** Improved Minimax with Alpha Beta Pruning
- **Section 5** Experimental Results
- **Section 6** Conclusion and Future work

2 Basic Ideas of Reinforcement Learning

We mainly focus on the model-free RL methods. *Model-free* method attempts to directly estimate the q-values of states, without ever using any memory to construct a model of the rewards and transitions in the MDP.

In the class we learned model-free RL with Monte Carlo method and with Time Difference Method. We can obtain an evaluation of the policy from the results obtained from multiple trials, as follows:

Based on policy-evaluation algorithm, we could use policy iteration to learn the best action (policy) for each single state, and after multiple iterations, The results of policy learning will become stable and

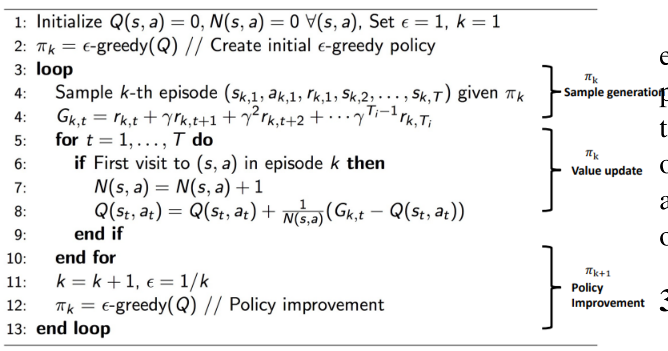


```
Initialize $N(s,a)=0$, $G(s,a)=0$, $Q^\pi(s,a)=0$, $\forall s \in S$, $\forall a \in A$
Loop
  • Using policy $\pi$ sample episode $i = s_{i,1}, a_{i,1}, r_{i,1}, s_{i,2}, a_{i,2}, r_{i,2}, \dots, s_{i,T_i}$
  • $G_{i,t} = r_{i,t} + \gamma r_{i,t+1} + \gamma^2 r_{i,t+2} + \cdots \gamma^{T_i-1} r_{i,T_i}$
  • For each state,action $(s,a)$ visited in episode $i$
      • For first or every time $t$ that $(s,a)$ is visited in episode $i$
          • $N(s,a) = N(s,a)+1$, $G(s,a) = G(s,a) + G_{i,t}$
          • Update estimate $Q^\pi(s,a) = G(s,a)/N(s,a)$
```

π_1 Sample generation
π_1 Value update

Figure 1: MC On Policy Evaluation.

converge to the optimal solution. Here is an example of policy iteration with MC (we have learned in class):



```
Initialize $Q(s,a)=0$, $N(s,a)=0$ $\forall(s,a)$, Set $\epsilon = 1$, $k = 1$
$\pi_k = \epsilon$-greedy($Q$) // Create initial $\epsilon$-greedy policy
loop
  Sample $k$-th episode $(s_{k,1}, a_{k,1}, r_{k,1}, s_{k,2}, \dots, s_{k,T})$ given $\pi_k$
  $G_{k,t} = r_{k,t} + \gamma r_{k,t+1} + \gamma^2 r_{k,t+2} + \cdots \gamma^{T_i-1} r_{k,T_i}$
  for $t = 1, \dots, T$ do
    if First visit to $(s,a)$ in episode $k$ then
      $N(s,a) = N(s,a)+1$
      $Q(s_t,a_t) = Q(s_t,a_t) + \frac{1}{N(s,a)}(G_{k,t} - Q(s_t,a_t))$
    end if
  end for
  $k = k+1$, $\epsilon = 1/k$
  $\pi_k = \epsilon$-greedy($Q$) // Policy improvement
end loop
```

Figure 2: Monte Carlo for On-policy Policy Iteration.

One of the most popular algorithm in RL is Q learning. Q-Learning is a value-based algorithm in reinforcement learning algorithm, Q that is $Q(s,a)$, is the state of the state at a certain moment, take action a can gain the expectation of the environment will be based on the agent's action feedback corresponding reward reward, so the main idea of the algorithm is to build the state and action into a table to store the Q value, and then according to the Q value to select the action that can get the maximum benefit.

The main advantage of Q-learning is that it uses the time difference method (a combination of Monte Carlo and dynamic programming) to learn off-policy and the Bellman equation to solve the optimal policy for Markov processes. The update function is shown below:

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

Algorithm 1 Q-learning Pseudo code

```
Input: $Q(s,a), s0, actionsetA$
Output: the approximate optimal Q-value we learned
Initialize $Q(s,a)$ for $\forall s \in S, a \in A$
$t \leftarrow 0, s_t \leftarrow s_0$
Set $\pi_b$ to be $\epsilon$-greedy w.r.t. $Q$
while Iteration termination condition not met do
    Take $a_t \sim \pi_b(s_t)$
    Observe $(r_t, s_{t+1})$
    $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t))$
    with prob $1-\epsilon$, $\pi(s_t) = argmax_a Q(s_t,a)$
    with prob $\epsilon$, random choose $\pi(s_t)$
    $t \leftarrow t+1$
end while
return Q-value for each state and action $(s,a)$
```

In short, the Q-learning process is learning from experience in multiple attempts at a Markov decision problem. Each attempt forms data at a new point in time, and the Q learning algorithm uses the newly observed data to update the previously learned model and eventually estimate the empirically meaningful optimal action choice for each state.

3 ADP for Gomoku

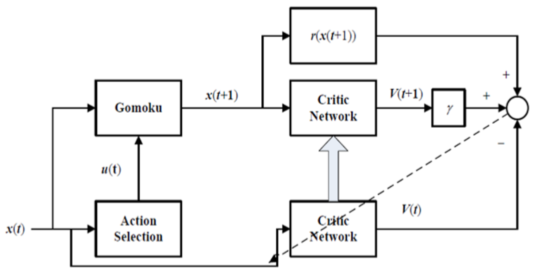
An adaptive dynamic programming (or ADP) agent takes advantage of the constraints among the utilities of states by learning the transition model that connects them and solving the corresponding Markov decision process using a dynamic programming method. The framework of ADP algorithm is showed as follow:

Algorithm 2 ADP Pseudo Code

```
Input: $percept$, a percept indicating the current state $s'$ and reward signal $r'$
persistent: $\pi$, a fixed policy;
$mdp$, an MDP with model $P$, rewards $R$, discount$\gamma$;
$U$, a table of utilities, initially empty;
$N_{sa}$, a table of frequencies for state-action pairs, initially zero;
$N_{s'|sa}$, a table of outcome frequencies given state-action pairs, initially zero;
$s, a$, the previous state and action, initally null.
if $s'$ is new then
    $U[s'] \leftarrow r', R[s'] \leftarrow r'$
end if
if $s$ is not null then
    Increment $N_{sa}[s,a]$ and $N_{s'|sa}[s',s,a]$
    for each $t$ such that $N_{s'|sa}[t,s,a]$ nonzero do
        $P(t|s,a) \leftarrow N_{s'|sa}[t,s,a]/N_{sa}[s,a]$
    end for
end if
```

```
$U \leftarrow$ POLICY-EVALUATION($\pi, U, mdp$)
**if** $s'$ is TERMINAL **then**
    $s, a \leftarrow null$
**else** $s, a \leftarrow s', \pi[s']$
**end if**
**return** action $a$
```

Our version of ADP based on the framework of Zhao[1], consists of three parts: Critical Network, Action Network and System Model. The process of ADP for Gomoku is presented as follow:



- x(t): current board state;
- u(t): action;
- x(t+1): next step state;
- r(x(t+1)): reward;
- The critic network is used to eatimate value function V(t)

Figure 3: The ADP Structure.

3.1 The state to describe a board situation

To evaluate a board situation, the important issue is how to choose the proper features. Zhao's work carefully choose 20 patterns for each of the two players(as figure 4 shows). However, such method need to deepcopy the board state for every move and scan the whole board to extract features.

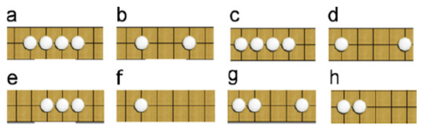


Figure 4: Examples of different patterns.

Zhao[1] use five input nodes indicate the number of every pattern except for five-in-a-row (n denotes the number of a pattern), which is shown in table 1. The number of the special pattern five-in-a-row, is represented by 1 input node; If this pattern shows up, then its input is 1, otherwise 0. For each pattern we assign two input nodes to represent the turn, and add two input nodes to indicate which player is the first to move. So totally we need **274** input nodes to discribe a board state.

3.2 Reward Model

The reward model presents that how the state of board changes and given rewards to the AI players. The reward is set to 0 during the game, and after a game, if player 1 wins, the final reward is 1, if he loses, the reward is 0, and if he draws, the reward is 0.5.In practice, draws are rarely happen.

3.3 Critical Network

A feed-forward three-layered fully connected neural network is adopted to evaluate board situations:

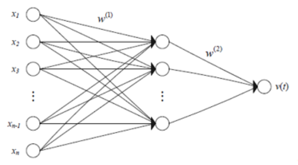


Figure 5: Examples of different patterns.

In our work, for each chess shape, two input nodes to present the number of point for two player respectively. Other two input nodes are assign to represent the turn for each chess shape as well. If AI itself moves first, then the first input is 1 and the second input is 0. If opponent player moves first, then the inputs for these two nodes are reverted. Also, two nodes presenting the turn for the whole board is needed. Therefore, the number of input nodes adds up to $100 \times (2 + 2) + 2 = 402$.

In our design, there are 402 nodes in the input layer, 64 nodes in the hidden layer and 1 node in the output layer. The output of the neural network is the winning probability of AI player starting from a board situation.

The key step is to optimize the Q function in each iteration to reduce the gap in realistic $r + Q(s', a')$ and the expected $Q(s, a)$. We define the prediction error as $e(t) = \alpha[r(t+1)+\gamma V(t+1)-V(t)]$, and the goal is to minimize objective error $E(t) = \frac{1}{2}e^2(t)$.

3.4 Actions

There are many accurate and approximate reductions to choose the next move. To make the AI player implied by pure ADP, those reductions based on heuristic functions are abandoned. The only reduction taken is that the points are considered only if which is near a position which has been occupied. During the training process, when there are several alternative actions which have equally high evaluation, we simply choose the one that is last found.

Value of n	input 1	input 2	input 3	input 4	input 5
0	0	0	0	0	0
1	1	0	0	0	0
2	1	1	0	0	0
3	1	1	1	0	0
4	1	1	1	1	0
> 4	1	1	1	1	$(n-4)/2$

Table 1: patterns encoding of board situation

Pattern	**Shape**	**Priority**
win	oooo_	A
flex4	_ooo_	B
block4 more than 2	xooo_*_ooox	B
block4 and flex3	xooo_*_oo	C
block4 and block3	xooo_*_oox	D
block4 and flex2	xooo_*_o	D
block4 and block2	xooo_*_ox	E
blocck4	xooo_	F
flex3 more than 2	oo_*_oo	G
flex3 and block3	oo_*_oox	H
flex3 and flex2	oo_*_o	H
flex3 and block2	oo_*_ox	I
flex3	oo_	J
block3 and flex2	o_ * _oox	K
block3	_oox	L
flex2	_o	M
block 2	_ox	N

Table 2: Full Pattern

To cope with the exploration-exploitation dilemma, the state space of Gomoku are explored in the ways below. For the first move, the AI player randomly choose a position to start. For the rest moves, a $\epsilon-$greedy policy is applied for both players.

With the number of training games increasing, the value of e should decrease gradually.

4 Improved Minimax with Alpha Beta Pruning

4.1 Introduction

In the final project, we improved the previous $\alpha - \beta$ AI and decided to use it in the competition. We have added many new chess patterns, which allows our strategy to focus on more details. We have implemented threat space search, including Victory of Continuous Four(VCF) and Victory of Continuous Three(VCT). In addition, we divide the search function into rootsearch and minimax (non-root nodes), which is conducive to the realization of the above functions.

4.2 Full Pattern Recognition

We added a new feature to each position on the board, called full pattern. It comprehensively considers the patterns in the four directions of the position. Its values are shown in Table 2.

When the pattern of a position is updated (such as moving a stone or undo a move), we will immediately update the full pattern of that position. In our strategy, these detailed full patterns will help us a lot.

4.3 Victory of Continuous Four

Victory of Continuous Four(VCF) is a technique that uses the absolute first move of continuously block4 or flex4 until the win5 is obtained. Details of VCF are as follows:

- **Startpoint and Searcher**: We set the searcher as the **first player** to perform VCF. We use the point where the fullpattern is at least **blcok3** in the occupied positions as the startpoint. If we can't find that point, we don't do VCF.
- **win5**: We have a priority A point, we win.
- **lose5**: Opp have more than one points with priority A, we lose.
- **block opp 5**: Opp have one priority A point, we put a stone at that point and recursive.
- **win4**: We have priority B points, we win.
- **try block4 and flex3**: We are the searcher and we have points with priority C. If opp have no point to form a 4(no points with priority B,C,D,E,F), we win. Or we put a stone at the C point and recursive.
- **try other4**: We are the searcher and we have priority D or E points. We only try the points

with D or E near the startpoint in case it takes too long and recursive.

- **try flex3**: We are the searcher and we have priority G points. If opp have no point that can form a 4, we win.

- **other**: We don't have patterns above, can't judge.

In the code, when our search result is victory, the AI will save the victory point. When the search result is confirmed, return the number of steps p needed to reach the end. p¿0 means victory, p¡0 means defeat. (These are not shown in the above information). For example, if we have point with priority B and opp have no point to form a five, we need 3 steps to win. We set max VCF depth to 20 and return 0 if search depth is bigger than 20.

4.4 Victory of Continuous Three

Victory of Continuous Three(VCT) refers to the tactical skills that use three attack methods, namely, live three, block four, and VCF continuously, and finally win. Details of VCF are as follows:

- **Startpoint and Searcher**: We set the searcher as the **first player** to perform VCT. We use the point where the fullpattern is at least **flex2** in the occupied positions as the startpoint. If we can't find that point, we don't do VCT.

- **win5**: We have a priority A point, we win.

- **lose5**: Opp have more than one points with priority A, we lose.

- **block opp 5**: Opp have one priority A point, we put a stone at that point and recursive.

- **win4**: We have priority B points, we win.

- **defend**: Opp are the searcher, and opp have a priority B point, we need to defend. Call function **getCandidates()** to get candidates in such situation. And we put stones at the candidate point and recursive.

- **try block4 and flex3**: We are the searcher and we have points with priority C. If opp have no point to form a 4(no points with priority B,C,D,E,F), we win. Or we put a stone at the C point and recursive.

- **try other4**: We are the searcher and we have priority D or E points. We only try the points with D or E near the startpoint in case it takes too long and recursive.

- **try double flex3**: We are the searcher and we have a point with priority G. If opp have no point to form a 4(no points with priority B,C,D,E,F), we win. Or we put a stone at the G point and recursive.

- **try other3**: We are the searcher and we have a point with priority H or I. We only try the points with H or I near the startpoint in case it takes too long and recursive.

- **other**: We don't have patterns above, can't judge.

As in VCF, in our code we will return steps to end the end and win point(if exsists). We set max VCT depth to 16 and return 0 if search depth is bigger than 16.

4.5 implementation

- **Pattern and Evaluation**

  As in the previous project, we use a 16-bit binary number to represent the chess pattern in one direction. The affected position will be dynamically updated every time we put a stone or undo a move. These can be achieved by simple bit operations.

  Each pattern have a score, and the score of a point is determined by the pattern in its four directions. The board evaluation is obtained by subtracting the scores of all the opponent's pieces from the scores of all my pieces.

  In order to speed up the search speed of VCF and VCT, we use a 2×15 list nShape to store the number of positions of full patterns that two sides can form in the next step. Priority judgement in the VCT and VCF can be quickly determined by looking up the list.

- **Minimax**

  Before running minimax(or rootsearch), we first run the VCF. If the result of the VCF is that our side wins, we directly return the best point. Otherwise, we run the VCT. If the result of the VCT is that our side wins, we return the best point. Finally run minimax (rootsearch).

Minimax part is the same as our previous version. We adjusted the search depth and branching factor. We set max depth = 10 and branching factor = 16.

- **RootSearch**

  In rootsearch, candidate node is the node that is feasible in the next step. We first call function **getCandidates()** as in minimax. There are two differences next:

  1) If the length of candidates is one, we simply return the one candidate.

  2) For each candidate, we make a move and run **VCT** from the opponent's perspective. We delete the candidates that the opponent can win through VCT. If all candidates are deleted, we select the candidate that needs the most steps to make us lose.

  The rest part of rootsearch is the same as minimax.

5 Experimental results

We used HMCTS,UCT and minimax search with α-β pruning AI to compete with 6 AIs on the GomoCup website. Each competition uses 3 kinds of fixed openings, alternate starting order, 6 games in total.We set the thinking time for each step not to exceed 15s, and the entire game time not to exceed 90s. The result is as follows:

Agent	Minimax	RL
YIXIN17	0	0
WINE17	0	0
SPARKLE	5	0
NOESIS	6	0
FIVEROW	6	2
MUSHROOM	6	4

Table 3: Competition outcome

6 Conclusion and Future work

In the final project, we first implemented the RL for Gomoku AI, espacially use self-teaching ADP. Limited by the machine and training time and the the lack of enough knowledge of neutral network, it doesn't perform well as expected, but we believe that more training time can make a smarter agent.

Then, we improved out $\alpha - \beta$AI by adding more detailed patterns and implementing threat space search. And we finally get around 1620 scores under Bayesian Elo.

In the project, we also tried several methods like iterative Deepening search and Zobrist Hashing but the score is not as high as out current agent. In order to achieve iteratively deep search, we have added a lot of new structures to the code, which makes the search time longer even when the depth is shallow. In addition, our agent with Zobrist Hashing takes a lot of time to initialize the hash table, and the hit rate of the hash table is low, about 0.06. So we finally gave up these two methods. We learned that many powerful AI programs are written by C++ not Python. We believe that out agent will perform better if written by C++.

In the end, we found that $\alpha - \beta$agent performed best among the three types of AI.The rules of Gomoku are relatively simple, and the time of each round and each move is strictly limited in this game. Minimax can directly select successor nodes based on the evaluation function, while other methods are limited by computing power and time and cannot produce the best results. Therefore, under the current rules, minimax can exert the best effect.*

References

Dongbin Zhao, Zhen Zhang, and Yujie Dai, "Self-teaching adaptive dynamic programming for Gomoku.", Neurocomputing, 2012, pp. 23-29

Z. Tang, D. Zhao, K. Shao, and L. Lv, "Adp with mcts algorithm for gomoku," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI).* IEEE, 2016, pp. 1–7.

H. J. K. Jun Hwan Kang, "Effective monte-carlo tree search strategies for gomoku ai," in *2016 International Science Press.* IEEE, 2016, pp.1–9.

J. Wang and L. Huang, "Evolving gomoku solver by genetic algorithm," in *2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA).* IEEE, 2014, pp. 1064–1067

L. V. Allis, H. J. Herik, and M. Huntjens, Go-moku and threat-space search, in *University of Limburg, Department of Computer Science.* Department of Computer Science , 1993.