RL Training Gomoku Agent Based on Artificial Environment

Final report for Gomoku

Xuetian Chen 19307110189 Yifan Li 19307110499

December 7th 2021

Abstract

In this RL implementation, we first build a game training environment based on Gomoku. Based on this virtual environment, we have completed tabular reinforcement learning and VFA reinforcement learning respectively. In the tabular RL part, the specific methods are divided into MC control and TD control concluding Q-learning and Sarsa. Finally, we compare and analyze the effects of these four agents, and put forward their shortcomings and improvement methods. In addition, the advantages and disadvantages of the training environment we built are also analyzed.

1 Introduction

In the previous two implementations, we tried to use Alpha-Beta pruning algorithm and MCTs to build Gomoku AI agents. This time, we will use another well-known algorithm in go game, which named as reinforcement learning.

Reinforcement learning is a very general algorithm, which has been studied in many disciplines, such as game theory, cybernetics and so on. In the literature of operations research and control, reinforcement learning is also called approximate dynamic programming or neural dynamic programming. The ultimate goal of reinforcement learning is to form policy. In Gomoku, reinforcement learning algorithm collects sample information by interacting with the environment, and uses sample information and function approximation to optimize performance.

Reinforcement learning includes several specific implementation methods. For Gomoku problem, we hope to find the feasible good enough algorithm.

2 Reinforcement Learning in Gomoku

First, build Gomoku's MDP environment. Think about an interaction between reinforcement learning and Gomoku game. We play chess alternately with our opponent *(actions)*, and each time a chessboard is detected *(state)*. Each action (play a chess) has no penalty or reward. Only after the last drop, that is, when the game result came out, will a reward be generated (it is specified that if the agent win then it get 10 points, else -10 points. Because Gomoku is a zero-sum game, a draw state rarely occurs in Gomoku. Instead, the final result usually turns out to be win or lose.[2]), then an episode has been formed.

In the implementation of RL, we must first optimize or converge the value of our agent's

action-return network through multiple training. Therefore, we artificially build an training environment for it and use the previously implemented minimax agent as its virtual opponent. The specific implementation methods are as follows:

Given a chessboard, when it is the RL agent to be trained to play chess, it selects an *action* according to a certain *policy* and transmits the action to the environment. The chessboard moves on a chess according to the action to form a temporary state, and the minimax agent in the environment selects its optimal action according to the temporary state to form a *new state*. The state and the reward generated during the environment response finally returned to the RL agent.

To sum up, when RL agent takes an action, it receives a state ***after two moves*** (unless the game ends halfway), and the reward is 0 unless the game has been decided.

This virtual training system performs well in the later practical use, but it also has certain limitations. We will analyze its advantages and disadvantages in detail in the last part.

Next, choose policy evaluation method. Policy evaluation can be divided into two approaches: Monte Carlo method and Temporary Difference method. These two methods have their own advantages and disadvantages in Gomoku. According to the fact that Gomoku environment can only return the reward of both players at the end, most of the values of Gomoku action-reward network are 0 at the beginning. For each interaction with the environment (i.e. simulation), only the state utility values of the last step will be updated, The huge state space of Gomoku determines that we have only a small probability to meet the same end state for the second time, which may lead to the slow convergence of TD Policy Iteration. However, the advantage of TD is that after every step, reward can respond in time and update state values, while MC can only update when whole episode is completed. It is possible that episodes will be very long, resulting in information not being used in time. Therefore,we decided to implement both assessments and compare their performance.

Thirdly, choose actions to generate samples. Now, choose a behavior policy to generate experience which trade off exploration and exploitation. Generally, we use ϵ-greedy policy.

Note that in MC control, each simulation will only affect the utility value of the state on that episode. This inspired us to think of UCTs. According to the experimental results of Xu Cao and Yang Lin[1], we will also find that the winning rate of AI agents constructed using the composite algorithm of UCT-ADP is higher than that constructed using only ADP algorithm. So it may be a feasible scheme to combine UCB policy with reinforcement learning. Therefore, We combine the structure of UCB to get our own behavior policy used in MC control:

$$\pi^{UCB}(s) = \underset{S' \in states}{\operatorname{argmax}} Q^{\pi}(s') + c\sqrt{\frac{lnN_s}{N_{s'}}}$$

Last, represent the value of state. According to the summary of reinforcement learning in class, there are both tabular RL and approximate value functions (VFA) RL. Table RL can calculate the specific value for detected state, which is more accurate, but it needs more memory and computation. At the same time, Gomoku's huge state space determines that it is almost impossible for tabular RL to obtain the value for each state. Using VFA to estimate the true value of the state can map the original huge state space into a smaller parameter space, reducing the computation, memory and experience. However, VFA is limited by function form and feature selection, so it is not easy to get well effect. We decided to implement both tabular RL and VFA RL respectively and choose the better one.

Based on the above, the pseudo code is as follows:

Algorithm 1 Monte Carlo On-Policy UCB Policy Iteration

Initial Q(s) = 0, N(s) = 0, k = 0, set γ.
loop
    Sample k-th episode $(s_{k,1}, ak, 1, r_{k,1}, s_{k,2}, \cdots, s_{k,T})$ given π^{UCB}
    $G_{k,t} = r_{k,t} + \gamma r_{k,t+1} + \cdots + \gamma^{T_i-1} r_{k,T_i}$
    for $t = 1, \cdots, T$ **do**
        $N(s_{k,t}) = N(s_{k,t}) + 1$
        Evaluate $Q^{\pi}(s_{k,t})$
    k = k+1

3 Tabular Reinforcement Learning

3.1 Off-Policy Monte Carlo Control

In the last task, we implemented the Monte Carlo algorithm. In fact, similar idea is also used in reinforcement learning. Apply Monte Carlo method to the update of value function. In MC, for state s visited at time step t in episode k, we can update its estimate value:

$$Q_k^{\pi}(s) = Q_{k-1}^{\pi}(s) + \frac{1}{N_k(s)}(G_{k,t} - V_{i-1}^{\pi}(s))$$

When calculating $G_{k,t}$, it is necessary to set an appropriate discount γ value. According to previous implementations, experience tells us that γ will not be too small. Set $\gamma = 0.80, 0.85, 0.90, 0.95, 1.00$ respectively and conduct several comparative experiments.

Game results are as follows: Note that the last 2 numbers in the AIs' name means their γ value. For example, $DMC90$ means this AI agent used discount $\gamma = 0.90$, the same is true for others. In particular, $DMC00$ represents $\gamma = 1.00$.

-	DMC90	DMC85	DMC80	DMC00	DMC95
DMC90	-	2 : 4	3 : 3	3 : 3	2 : 4
DMC85	4 : 2	-	1 : 5	2 : 4	3 : 3
DMC80	3 : 3	5 : 1	-	1 : 5	3 : 3
DMC00	3 : 3	4 : 2	5 : 1	-	2 : 4
DMC95	4 : 2	3 : 3	3 : 3	4 : 2	-
总比分	14 : 10	14 : 10	12 : 12	10 : 14	10 : 14
胜负比	1.400	1.400	1.000	0.714	0.714
得分	8	7	5	4	2

Figure 1: contests result between different gamma

Note that the difference between the two highest score AI agent with $\gamma = 0.85$ and $\gamma = 0.90$ respectively is not very big. Which may be due to too few innings. Therefor, increase the number of battles between the two alone. Result is DMC90 won with rate 2.

Now we get a rough best value of discount $\gamma = 0.90$. This value can also provide a reference for us to select γ later.

3.2 Temporary Different Learning

In the previous section, we consider transitions from state to state and learn the value of states. Now we consider transitions from state-action pair to state-action pair, and learn the value of state-action pairs.

TD learning is a combination of Monte Carlo ideas and dynamic programming ideas. Like Monte Carlo methods, TD can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other other learned estimates, without waiting for a final outcome.[3]

In TD, approaches fall into two parts: on-policy and off-policy. On-policy means that the policy for interacting with the environment and the policy for the actual update iteration are the same policy; On the contrary, off-policy means that the policy interacting

with the environment is different with the policy actually updated. Both approaches have representative algorithm. We will implement Q-learning (off-policy) and Sarsa (on-Policy) respectively. The pseudo-code of the two algorithms can be summarized as follow:

Algorithm 2 Temporary Difference Algorithm

1: Initial Q(s,a) $\forall s \in S, a \in A$. t=0,initial state $s_t = s_0$
2: Set π^e to be ϵ-greedy w.r.t Q
3: **loop**
4: Take $a_t \sim \pi^e(s_t)$
5: Observe $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$
6: Update $Q(s_t, a_t)$
7: Update policy $\pi^e(s_t) = \arg\max_a Q(s_t, a)$ with probability $1 - \epsilon$, else random
8: $t = t + 1$

3.2.1 Q-learning

One of the most important breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins,1989). In Q-learning, the agent's experience consists of a sequence of distinct episodes, In the n^{th} episode, our agent:

1. Observe its current state s_n
2. Selects and performs an action a_n according to ϵ-greedy policy
3. Observes the subsequent state s_{n+1} from environment
4. receive an immediate payoff r_n, and
5. adjusts its Q_{n-1} values using a learning factor α, according to

$$Q_n(s_n, a_n) = Q_{n-1}(s_n, a_n) + \alpha[r_n + \gamma \max_{a_{n+1}} Q_{n-1}(s_{n+1}, a_{n+1}) - Q_{n-1}(s_n, a_n)]$$

According to Watkins, $Q_n(s_n, a_n) \to Q^*(s, a)$ as $n \to \infty$ with probability 1.
Fortunately, our minimax runs very fast, making the training time completely acceptable.

3.2.2 Sarsa

The Sarsa agent behavior is basically the same as that of Q-learning, except that the update after each transition changes into:

$$Q_n(s_n, a_n) = Q_{n-1}(s_n, a_n) + \alpha[r_n + \gamma Q_{n-1}(s_{n+1}, a_{n+1}) - Q_{n-1}(s_n, a_n)]$$

And a_{n+1} comes from observed transition.
Different from Q-learning, the convergence of Sarsa depends on the nature of the policy's dependence on Q. We can set $\epsilon = \frac{1}{t}$ to make sure Sarsa converges with probability one to optimal policy and action-value function.

4 Approximate Value Functions

The problem for previous approaches is the memory needed for large tables and the time and data needed to fill them. The key issue is generalization.[3]. The kind of generalization we use is called function approximation, it take examples from a value function and attempts to generalize from them to construct an approximation of the entire function.
In this implementation, the function approximator we choose is linear combinations of features. It is no easy to obtain feasible features that can capture important properties of the state.
After many attempts, we finally selected 14 features to form the feature vector of the chessboard. In Gomoku, some widely recognized important chess types determine the trend of the game. Chess types are mainly classified according to the number of pieces and whether the two ends of the chess type are blocked. In the initial idea, we included all chess types into the feature vector, but we found that the final result of the weight of the chess type only contains one chess piece is very large, which makes the performance of AI agent very poor. We analyze the reasons and found that the chess type containing

just one chess piece accounts for too much of all chess types in the whole chessboard, and both players will form many chess type containing one chess at the beginning of training, resulting in a much faster weight update of the chess type of just one chess piece than other chess types.
Finally, we count the number of chess type composed of two or more pieces on the chessboard, and take them as feature vectors.
Combined with the above analysis and experiments, we finally adopt linear, gradient-descent Sarsa and ϵ-greedy policy. The pseudo-code is as follows.

Algorithm 3 linear, gradient-descent Sarsa and ϵ-greedy policy

Initial α, ϵ, γ, w,k=0
loop
    Sample tuple (s_k, a_k, r_k, s_{k+1}) generated by $\epsilon-greedy$ policy and environment
    Update weight:
    $w = w + \alpha * ((r_k + \gamma \hat{V}^{\pi}(s_{k+1}, w) - f(s_t)^T w) * f(s_t))$
    $\alpha = \alpha *$ step-size discount
    k=k+1

Finally, the parameters after debugging are $\alpha = 0.01, \epsilon = 0.10, \gamma = 0.9$. Note that after each update, step size α must be decreasing. This is because when the number of training times increases, the weight is very close to the optimal solution. We need to limit the step size of each update to ensure better convergence. In the implementation, we set this discount value to 0.9.
In addition, we find that the setting of the initial value of the weight has a great influence on the convergence of the Gomoku problem. For example, if you simply set the initial value of the weight to all 0, the weight will diverge in the subsequent iterative update and cannot converge. After many parameter adjustment attempts, we set the initial weight to $[0, 0.5, 1, 2, 4, 8, 10, 0, -0.5, -1, -2, -4, -8, -10]^T$, which came from our Gomoku experience: the chess type with more pieces is more important than the chess type with less pieces, and the chess type without blocked on both sides is more important than the blocked chess type. This initial value has proved its convergence in practice.

5 Analysis and Future Improvements

Finally, we test and compare the four agents one by one. The result is that VFA agent is much better than other three agents. So we choose VFA agent as our final RL agent. And Final Elo ratings of our VFA agent is 1202.0. We are glad to see that the chess power of VFA agent is good. The reason is that Gomoku's state space is too large for tabular RL method, so it is impossible to traverse each chessboard state and calculate its convergent utility.
But for the VFA method, because it uses the idea of generalization, even if the state has never been encountered, we can estimate its value according to previous experience, and some changes in the MDP will not make a large fluctuation on its effect. This is a model with high generalization ability.
Of course, the current VFA agent still needs to be improved.
First, the current feature extraction is too rough, only the sum of chess types of each point are considered. But the chess type interaction between different points is also not considered. Therefore, the current feature vector blurs too much chessboard information.

5.1 Analysis of Minimax training Environment

5.1.1 Stochastic/deterministic model.

Our Minimax agent is a deterministic agent. If our Minimax agent is globally optimal, the

tablar RL agent trained by the environment will also play optimally. However it is impossible. We must consider the case that the true environment (the true opponent in a given match) acts differently from training environment. With incorrect information of the environment, its decision may be suboptimally. We can modeling the training environment as stochastic: in probability $1-e$, take the same action as Minimax agent does, in probability e, take another action, maybe a Minimax action with different depth or action taked by other AI. With a stochastic model of the environment, RL agent can perform better when encountering different types of opponents.

5.1.2 Upper Limit of VFA agent.

If we use a Minimax agent with depth 1 as the training environment, the trained VFA is exactly the depth 2 Minimax agent: it maximize the minimized evaluated vuale of board-after-2-steps (S2), because our Minimax agent also use linear evaluation function.
However, if trained by depth 2 or more, say, 5, the case is more complex. Our VFA agent will maximized the values of S2 boards, which are minimized values of S6 board chosen by depth 5 environment. It is like a hybrid of depth 2 and depth 6. Certainly it will play better than depth 2 Minimax, but worse than depth 6 one, because it doesnot know the minimized S6 values computed by the environment, only the S2 values of current chosen board.
An optional method is feedback the true evaluated S6 value, then the agent will play almost the same as depth 6 Minimax. But it is inefficient and meaningless, because the training process is much longer, and the action selection is a like a totally computed Minimax with no pruning.

5.1.3 Linearity of VFA.

A linear function of chess shapes is not a good representation of the board. Consider a lose-match with a lot of chess. The reward is negative. The update will consider the current features as much worse ones, because the feature value is large. This results in divergence of weights in our first trials of VFA.

5.1.4 Four-Direction Shape VFA.

Now our VFA agent extract only one-direction features. Use four-direction shapes as features are more precise. Fortunately, in our final version of AB agent, the information about four-direction shapes of each chess is recored and updated in tiny extra time cost. Therefore it is available and we have tried this idea. However with number of features increasing to 32, the tuning process is much harder than before. We don't have extra time to try tuning the initial parameters in order to guarantee its convergence.

References

[1] Cao, X., & Lin, Y. (2019). UCT-ADP Progressive Bias Algorithm for Solving Gomoku. 2019 IEEE Symposium Series on Computational Intelligence (SSCI), 50-56.

[2] Tang, Z., Zhao, D., Shao, K., & Lv, L. (2016). ADP with MCTS algorithm for Gomoku. 2016 IEEE Symposium Series on Computational Intelligence (SSCI), 1-7.

[3] Richard S. Sutton, & Andrew G. Barto. (1998). Reinforcement Learning: An Introduction. A Bradford Book.

[4] Watkins, C.J., & Dayan, P. (2004). Q-learning. Machine Learning, 8, 279-292. .