# Deep Q-learning Method for Gomoku

***Abstract*—In the last two checkpoints, we have already implemented Min-Max Search algorithm with Alpha-Beta Pruning and MCTS Search algorithm on GOMOKU AI. This time, except for improving our Min-Max Search AI with some more threating algorithm, like VCT and VCF, we have also implemented Double Deep Q-Learning algorithm with Experience Replay, which is an upgraded version of Deep Q-Learning.**

***Index Terms*—MinMax Search, VCT, VCF, DQN, DDQN, Experience Replay**

## I. Introduction

GOMOKU is a popular board game played in many countries in the world. In this game, players alternatively place pieces of their color on the board. 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. There are many algorithms that can build up an GOMOKU AI, and we have already use **Min-Max Search** with Alpha-Beta Pruning, **HMCTS** in the previous two checkpoints. This time, our work mainly contains two part: improve our MinMax-Search AI by some useful tricks to make them performs better, and build an AI by Reinforcement Learning, Double Deep Q-Leaening (**DDQN**) exactly.

**Reinforcement Learning** can broadly be separated into two groups: **model free** and **model based** RL algorithms. Model free RL algorithm don't learn a model of their environment's transition function to make predictions of future states and rewards. **Q-Learning**, **Deep Q-Learning Networks** (**DQN**), and **Policy Gradient** methods are model-free algorithms because they don't create a model of the environment's transition function.

Reinforcement Learning involves managing state-action pairs and keeping a track of value (reward) attached to an action to determine the optimum policy. However, this method of maintaining a state-action-value table is not possible in real life scenarios when there are larger number of possibilities. Therefore, instead of utilizing a table, we can make use of Neural Networks to predict values for actions in a given, which we called **Deep Reinforcement Learning**.

**Q-Learning** is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards without requiring adaptations. However, when the Q-table is much bigger, the naive Q-Learning algorithm works poorly. To save time and memory, we can use Neural Networks to approximate these Q-values instead of storing all the accurate (action, value) pairs, which is called **Deep Q-Learning Network** (**DQN**). Besides, to make the algorithms converges quickly and use the samples more efficiently, we can use some other tricks such as **Experience Replay** or **Double Network**, both of which are implemented in our experiment.

The total report can be divided into these 4 sections (except the first section):

- Section 2 Improvement of Min-Max Search
- Section 3 Deep Q-Learning Network
- Section 4 Experimental Results
- Section 5 Conclusion and Future Work

## II. Improvement of Min-Max Search

To perform better in the elo rating, we decided to use Alpha-Beta Search method in the final competition for its efficiency and flexibility. Our strategies to gain higher performance can be concluded into to parts: First, we implemented fast pattern detection functions to store the cross patterns on the board; Second, before doing min-max search, the agent will do Prior search first, which consists of **VCF** and **VCT**, two classical idea in the Gomoku world.

### *A. Detailed Representation of Patterns*

In the previous implementation, we've already considered some basic cases of cross patterns. However, to get larger search depth while searching fast, we find it neccessary to update our representation of pattern.

In this implementation, the class $Board$ contains more information of the current state. To be specific, we added a new property, $Cross$, to save the positions where can form special combinations of chess pattern in four directions once we put a chess. Here we divide the cross patterns into 3 big categories and each categories contains several classes according to there priorities. It is worth noting that the priority is important for the later implementation of Prior Search.

For example, sometimes the next move we take will form a double "C4" or a "H4", in which case the opponent will have to block one of them (or one side of it) and we can win with the other, or vice versa. While if a "C4" and a "H3" formed at the same time will not guarantee the victory in 1 step, but we will win within 3 steps. Then these two different cases are

TABLE I
SPECIAL PATTERNS

| | | |
|---|---|---|
| VCF | VCF6 | oooo_ |
| | VCF5 | oooo_ |
| | VCF4 | xooo_*_oo or xooo_ * _ooox |
| | VCF3 | xooo_ * _oox or xooo_ * _o |
| | VCF2 | xooo_ * _ox |
| | VCF1 | xooo_ |
| VCT | VCF4 | oo_ * _oo |
| | VCT3 | oo_ * _oox or oo_ * _o |
| | VCT2 | oo_ * _ox |
| | VCT1 | oo_ |
| TWO | TWO4 | o_ * _oox |
| | TWO3 | xoo_ |
| | TWO2 | o_ |
| | TWO1 | xo_ |

set to two different classes in one category. More details can be seen in the following table.

To update the patterns quickly, we carefully considered our representation of the board, which reduces the $20 \times 20 = 400$ positions into $20 + 20 + 39 = 79$ different lines (actually only part of the lines need to be considered). Finally, the property $Cross$ is a list nested with a sequence of sets: two list indicates two different players, and the sets contains the positions that will form different cross patterns.

$$Cross = [\ [Set(), \ldots, Set()], [Set(), \ldots, Set()]\ ]$$

Once we want to find the threat positions, we can find them easily by traversing the sets.

## B. Victory of Continuous Threat

VCT(Victory of Continuous Threat), which also can be divided into VCT(Victory of Continuous Three) and VCF(Victory of Continuous Four), is a classical strategy for human players to defeat the opponents. In VCF part, the player put chesses continuously in "H4" or "C4", to force the opponent to block the threat pieces in next step until the five in a row is obtained. And in VCT, the player use three attack methods, namely, "H3", "C4", and VCF continuously, and finally win the game.

In VCF, we will first find a position which will form "H3" or better patterns, then check whether it will offer us a sequence of four patterns, that is, "H4" or "C4" and so on. This process includes checking whether the opponents can block us with their threat positions. Finally we will return the positive number of steps to win or negative number to lose. And in VCT, a similar pattern is followed with positions will form "H2" are considered as starters. Note that to avoid the exceeding of the time limits, the searching is restricted by $MAXFDEPTH$ and $MAXTDEPTH$ respectively and those funtions will return 0 if search depth is bigger than the limitations.

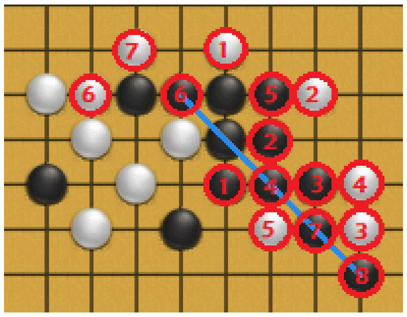
Fig. 1. VCT

## III. DEEP Q-LEARNING NETWORK

Reinforcement learning is an exciting field that has attracting a lot of attention and popularity. One of the core concepts in Reinforcement Learning is the Deep Q-Learning algorithm. Q-Learning (QL) is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards without requiring adaptations. In Deep Q-learning Network (DQN), instead of using a table mapping each state-action pair to its corresponding Q-values, we use a neural network to map the input state to (action, Q-value) pairs.

### A. *Introduction of Q-Learning*

The key idea of Q-Learning is maintaining state-action Q estimates and use the value of the best future action for bootstrap update. At each time step, given ($s_t$, $a_t$, $r_t$, $s_{t+1}$), the update equation is

$$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha \left(Q_t - Q(s_t, a_t)\right), \qquad (1)$$

where

$$Q_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a'). \qquad (2)$$

Q-Learning is a simple yet quite powerful algorithm to create a cheat sheet for our agent, and this helps the agent figure out which action to perform. However, the naive A-Learning algorithm does exists some problems:

- First, the amount of memory required to save and update the table would increase as the number of states increases.
- Second, the amount of time required to explore each state to create the required Q-table would be unrealistic.

Here is a thought - what if we approximate these Q-values with machine learning models such as a neural network?

**Algorithm 1:** ***Deep Q-Learning Algorithm***

```
Initialize action-value function $Q$ with random weights $\theta$
For episode = $1, \cdots, M$ do
    Initialize state $s_t$
    For $t = 1, \cdots, T$ do
        Select $a_t = \begin{cases} \text{argmax}_a Q(s_t, a; \theta) & 1 - \epsilon \\ \text{a random action} & \epsilon \end{cases}$
        Execute action $a_t$ and observe reward $r_t$ and next state $s_{t+1}$
        Set $s_t = s_{t+1}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) & \text{for non-terminal } s_{t+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(s_t, a_j; \theta))^2$
    End For
End For
```

## B. *Introduction of Deep Q-Learning Network*

In DQN, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. The comparison of the naive Q-Learning and the Deep Q-Learning algorithm can be seen in Figure 2.

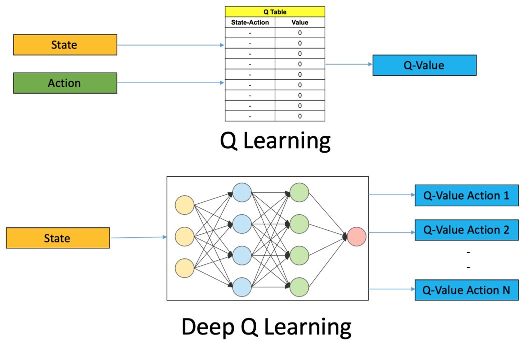

Fig. 2. The Comparison between QL and DQN

## C. *Improvement of Deep Q-Learning*

As we can see in the above pseudo-code, the target is continuously changing in each iteration, which causes the unstable learning process. Therefore, an intuitive solution is to keep the target unchanged for a few iterations. One that algorithm uses two networks instead of just one network in the training process, which is called *Double Deep Q-Learning Network* (**DDQN**). Furthermore, to make more efficient use of the samples, we can store them in our capacity and reuse them later on, which is called *Experience Replay*.

The pseudo-code of the DDQN with Experience Replay is shown below (the blue part are the difference with DQN):

**Algorithm 2:** ***DDQN with Experience Replay***

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\hat{\theta} = \theta$
For episode = $1, \cdots, M$ do
    Initialize state $s_t$
    For $t = 1, \cdots, T$ do
        Select $a_t = \begin{cases} \text{argmax}_a Q(s_t, a; \theta) & 1 - \epsilon \\ \text{a random action} & \epsilon \end{cases}$
        Execute action $a_t$ and observe reward $r_t$ and state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$
        Set $s_t = s_{t+1}$
        Sample random minibatch of transitions $(s_t, a_t, r_t, s_{t+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \hat{\theta}) & \text{for non-terminal } s_{t+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(s_t, a_j; \theta))^2$
    Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

*1) **DDQN**:* Since the networks in naive DQN that calculating the predicted value and the target value are the same, there could be a lot of divergence between these two. Therefore, we can use two neural networks that calculating the target value (**Target Network**) and the predicted value (**Prediction Network**) separately (shown in Figure **??**).

The Target Network has the same architecture as the Prediction Network but with frozen parameters. For every $C$ iterations (a hyperparameter), the parameters from the Prediction Network are copied to the Target Network. This leads to more stable training because it keeps the target function fixed (for a while).

*2) **Experience Replay**:* Experience Replay in Deep Q-Learning has two functions: make more efficient use of the experiences during the training, and avoid forgetting previous experiences and reduce the correlation between experiences.

Experience replay helps us make more efficient use of the experiences during the training. Usually, in online reinforcement learning, we interact in the environment, get experiences (state, action, reward, and next state), learn from them (update the neural network) and discard them. But with experience replay, we create a replay buffer that saves experience samples that we can reuse during the training. This allows us to learn from individual experiences multiple times.

The problem we get if we give sequential samples of experiences to our neural network is that it tends to forget the previous experiences as it overwrites new experiences. For instance, if we are in the first level and then the second, which is different, our agent can forget how to behave and play in the first level. The solution is to create a Replay Buffer that stores experience tuples while interacting with the environment and then sample a small batch of tuples. This prevents the network from learning about what it has immediately done.

The whole process of DDQN with Experience Replay can be seen below:

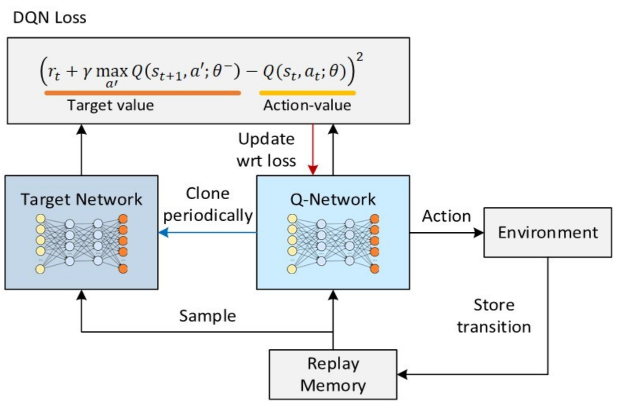

Fig. 3. DDQN with Experience Replay

## D. *Implementation details*

In our implementation, we used two layer network, the first layer uses $Relu$ activation funciton with 218 input nodes and the sectond layer uses $Sigmoid$ activation function with 64 hidden nodes. Given the input values, which are the features we extracting from the current board, the network will finaly output a value as the evaluation of the winning probability of current board state, which will be used in $\epsilon - greedy$ when we select the next optimal policy. In the code, 2 input nodes denote the current player, 6 nodes represent the number of the patterns one player have, and we consider both side at the same time.

## IV. Experiment Results

In the matching, our AIs will compete with 12 AIs from Gomoku website, under 3 fixed openings with exchanges of hands. All players should limit the time of each step within 15 seconds and the entire game time should not exceed 90 seconds. The highest score our agent can get up to around 1425. The results are listed as follow.

TABLE II
Gomoku Competition Results

| **Opponent** | **Minimax Agent** |
|---|---|
| YIXIN17 | 0:12 |
| WINE | 1:11 |
| PELA17 | 0:12 |
| ZETOR17 | 2:10 |
| EULRING | 1:11 |
| SPARKLE | 8:4 |
| NOESIS | 7:5 |
| PISQ7 | 9:3 |
| PUREROCKY | 10:2 |
| VALKYRIE | 11:1 |
| FIVEROW | 9:3 |
| MUSHROOM | 11:1 |

## V. Conclusions and Future Work

So far, we have implemented several GOMOKU AIs by using MinMax-Search with Alpha-Beta Pruning, MCTS, HM-CTS, DDQN, and improved our MinMax-Search Agent with some new tricks. And finally we get at most 1425 rating under the Bayesian Elo.

However, there still exist a lot of works we can do to improve our GOMOKU AI. If we still choose MinMax-Search as our searching algorithm, we may need to construct more computationally efficient representations of the states, or more efficient transitions method from state to state. For example, pela, they use a novel representation, which maintains a $256\times256$ matrix-like hash table to evaluate the point in the chess board. If we want to use some other algorithms, such Reinforcement Learning, we may need to come up with more powerful neural networks and need to use some deep-learning coding language, such as Pytorch or Tensorflow to help us build the algorihtm architecture.

## References

[1] 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 Ai in Games, 4(1):1–43.

[2] L.V.Allis, H.J.Herik, and M.P.H.Huntjens. Go-moku and threat-space search. 1993.

[3] Zhentao Tang, Dongbin Zhao, Kun Shao, and Le Lv. Adp with mcts algorithm for gomoku. 2016 IEEE Symposium Series on Computational Intelligence (SSCI), IEEE, 2016:1–7

[4] H.J.K.Jun Hwan Kang, Effective montecarlo tree search strategies for gomoku ai, 2016 International Science Press. IEEE, 2016:1–9.

[5] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. Nature, 2016, 529(7587):484–489.

[6] Hasselt, Hado. Double Q-learning. 2010.