Reversi Agent based on MiniMax Dueling-DQN network

Pengchao Tian
76719970
tianpch@shanghaitech.edu.cn

Peishan Cong
52511592
congpsh@shanghaitech.edu.cn

Zesong Qiu
99090887
qiuzs@shanghaitech.edu.cn

Wenhui Qiao
57425238
qiaowh@shanghaitech.edu.cn

Abstract

Board Games always push the player not only consider how to do the local optimal decision at current state, but also ask the plaer to guess how will his opponent will perform in the next state. MiniMax tree is one of the basic idea to consider the states in the future. However, MiniMax tree has high time complexity and hard to define the reward and its parameters, only use MiniMax tree in an agent is hard to perform well.

1. Introduction

People have played board games for a long time, which are usually adversarial games. Started from IBM's Deep Blue, people were trying to let computers to learn to become smarter enough to play chess with masters. Recently, Alpha Go, developed by DeepMind, has beat Ke Jie, proving that computers could learn the skills in the games.

Adversarial games, usually opponents have to speculate the other's strategy to try to get more reward from them. In the broad games, the reward is mainly determined by the number of chess on the chess board and the positions of the chess on the chess broad. And thus, the policy to take the high reward position and avoid the dangerous position is the main task to learn a board game.

In the general board games, it has initial states s_0, and other states s_t at *t* time. Two players denoted as p_1 and p_2. its next action set depends on the current state $action\,(s_t, p_i) = \{PointPosition\}$. With specific rules, update function, *Update*$(s_t, p_i, action)$ to update the chess board. s_t = *Update*$(s_{t-1}, p_i, action)$

In our project, we decide to utilize the Reversi, which has a different way of attacking, scrambling the opponent in the other chess games, , to implement an agent that could learn the specific strategy to try to beat other agents online or even people.

2. Reversi Game

2.1. Rules

2.1.1 States

Reversi is a game with 8×8 chess board. Two players will be adversarial on it, one with black chess and the other with the white chess.

The initial state, as s_0, shown in the figure 1. The black picker goes first and then the white goes next.

Then it may have $3^{8\times8}$ states in all.

2.1.2 Actions

The action depends on the chessman on the board. This game aims to *flip* the opponent's chess on the chess board.

Any legal position has to be able to catch the opponent's chess in the middle with the legal position and the our chess on the board, to change the opponent's chess into ours.

The algorithm below shows how we decide the positions in the current state s_t are available for a player, denoted simplied by 1 and -1, 1 means the black and the other means the white. Also the current state, as *board* is sent into the algorithm. Returning the available position, actions to take and the points we get in this action.

To identify whether this position P is legal, only to consider the row, column, two diagnal lines crossing P, and learn whether the opponent's points are caught to flip.

As shown in the figure 2, the crossing point is the legal position for the black chess is because the oringe Diagnal line has another black chess on the board, so the black chess can catch the white chess in the middle, flipping the white into the black chess.

2.2. Strategy

Players in the Reversi aim to get more positions in the limited 8×8 space chess board, so the basic strategy for

Algorithm 1 Get available actions and Flipping points

function FLIPPING($id, board$)
  $AvailAct \leftarrow \{\}$
  $FlipP \leftarrow \{\}$
  for $Points \leftarrow (x, y), x, y \leftarrow range\,(0, 8)$ **do**
    $Row = boards\,[x, :]$
    $Column = boards\,[:, y]$
    $Diag_1 = diag_1\,(boards\,[x, y])$
    $Diag_2 = diag_2\,(boards\,[x, y])$
    $LineSet \leftarrow \{Row, Column, Diag_1, Diag_2\}$
    for $lines \in LineSet$ **do**
      for $Ps = lines\,(Points :)$ **do**
        If $\exists t, Ps[1 : t-1] == -id \& Ps[0] == id \& Ps[t] == id$ **then**
          $AvailAct \leftarrow AvailAct \cup \{Points\}$
          $FlipP = FlipP \cup \{Ps[1 : t-1]\}$
        Endif
      end for
      for $Ps \in lines\,(: Points)$ **do**
        If $\exists t, Ps[1 : t-1] == -id \& Ps[0] == id \& Ps[t] == id$ **then**
          $AvailAct \leftarrow AvailAct \cup \{Points\}$
          $FlipP = FlipP \cup \{Ps[1 : t-1]\}$
        Endif
      end for
    end for
  end for
  return $AvailAct, FlipP$
end function

the beginer of this game is to flip more opponent's chess in each transaction t.

However, the most important thing is that the space is limited for the player, the weight of position can be strongly different from one to another. Take the corner as an example, the four corners could not be flipped by the opponent because their neighbors are only three. So that we can learn that who get more corners in the game, he has higher probability to win this game.

What's more, when one player is trying to flip his opponent's chess, the opponent is also trying to flip his. So even though one player has occupied lots of position in the board, his opponent can also win if he could reverse more chess in several actions at last. The local optimal for a single action is not enough.

The players have to design a strategy to help them to occupy more edges and corners in the game, also to defend themselves from being flipped.

2.3. Challenge

It seems to good for MiniMax taught from class, but MiniMax cannot consider the deeper level of the state in the chess board also how to define the reward for each transaction is a problem. What's more the performance of MiniMax is restricted by value functions given by human, which may not be optimal.

If only depends on the number of points flipped by each transaction and the weight of each position, the reward is only local, which is not a long term strategy with *h* level of tree. Also how to adjust the parameter of each weight is a problem.

Then we come up a strategy to dynamically to represent the state.

3. Model and Improvement

3.1. Minimax

Basic idea of MiniMax is to build a tree at the current state s_t, as the root node, and from the current state to do legal actions to the adversarial states $s_{t+1} = Update\,(s_t, p_i, action)$. And then recursivly to do the state transactions in the as Uniform Cost Search. Though the tree grows exponentially, the purning method can be utilized to aviod the strength of exponential growth, time complexity is $O\,(N \times b^m)$, where *N* is number of states, *b* is the max number of child node, and *m* is tree height.

However, the main thing is that the reward of the state is hard to measure. Such as the standard of the state, the number of chess to be flipped, the position of own chess is. In spite of exploiting these feature to measure the reward, the parameter of them have to be adjust manually and the height of tree is limited. Its hard and cost much time to design a MiniMax agent to perform well.

To prove this, we have written a MiniMax agent to compete with our MiniMax Dueling DQN agent, the result comes out that the latter performs much better.

3.2. Deep Q-learning Network

Since MiniMax has to adjust the parameters manually, its easy to think of a way to train the agent to adjust the parameter spontaneously, which is Reinforcement Learning.

Enlightened by q-learning, which is capable of learning the state's Q Value online, learn from playing the Reversi.

Nevertheless, it is difficult to use q-learning directly to learn the best path since a Q chart to record the state of each action would make the chart unimaginably large. DQN does not record the Q value but use the neural network to predict the Q value, and learns the optimal action path by constantly updating the neural network. [1] It is composed by two networks,target_net, which gets the value of the q-target, and eval_net, which gets the value of the q-eval.

The training data is randomly extracted from the memory bank, which records the actions, rewards, and results of the next state (s, a, r, s').s represents the chess state and

action is the position where to put chess,s' represents the state after chess putting and chessboard fliping.

The loss is a loss for L2 regression.

$$L = (q\_eval - q\_target)^2$$

where q_eval is the value of $Q_{eval}(s_t, a_t, s'_t)$ to be predicted by the Evaluation Network, Target_net only does forward propagation to get Qvalues, and the $q\_targetl$ is equal to $r_t + \gamma \max\left(Q_{target}\left(s_t, a_t, s'_t\right)\right)^2$.

$$\text{Loss } = \left(Q_{eval}(s_t, a_t, s'_t) - \left(r_t + \gamma \max Q_{target}\left(s_t, a_t, s'_t\right)\right)\right)^2$$

In the parameter update cycle of the algorithm, samples in the memory pool are randomly sampled or sampled in batches, and parameters of the model are updated through q-learning.

3.2.1 Change from convolution to FC layer

Convolution layer learn the local information while in reversi game, position information is more significant than local area information.The corner should be more important than other position.If use the 3×3 fliter core,the value of corner will be diluted and the edge position near corner will be strengthened, which leads to inaccuracy. We reckon that the concept local receptive field of conv net is not suitable for reversi game, since its state do not have to be represented as an image. So we change the convolution layer to FC layer. The input of the $8 * 8$ chess state will be view into $1 * 64$. Then it turns out the q_value on the corner predicted by the network show its significant as we thought.

3.2.2 Adding resnet layer

The Residual Unit is shown in figure 3,the formulation of $F(x) + x$ can be realized by feedforward neural networks with "shortcut connections",which pass the previous output to the back. Identity shortcut connections add neither extra parameter nor computational complexity and it can prevent overfitting when training. Our network is shown in figure 4

3.2.3 The same network training for two rivals

We tried to use two different network with distinct parameters for players while it turn out that it is hard to become converge so we use the same network training for two rivals.In practice,we multiply -1 by the corrent chess state s,so that the original place for black(1) is white(-1).On the perspective of the opponent, the state stored in memory is corresponding with its identity.For each Agent Net, they will see their own identity as 'virtual', always assume they are 1, their opponent is -1. But acctually, in physical chess board, black is 1 and white is -1.

3.2.4 Backword for the whole game

Our original thought about reversi is that this game is a rare-reward model, it can only get one value at the final of the whole game. Thus, we need to transmit the final value to all mediate state with backward algorithm. Then train the agent with state and relative value.
The experiments proved this method does not work. The performance is terrible, and AI learned nothing. The backward value of mediate state should multiply a discount, so the value is too small to represent some information.

3.2.5 Reward standard

To figure out the problem with backward. We add some reward on each position according to our experience on playing to replace the backward method. The corner should have the highest reward.It can be imagined that to guarantee it won't loss corner, the point on the position (0,1) and (1,0) (which is next to corner) will be lower compared with other edge position.The reward on each position is shown in figure 5 (since the chess board is symmetry,the reward is shown on top 4×4).And we define the reward for putting chess on position p as:

$$R(p) = V(p) + V(\text{Flip}(p, \text{identity}))$$

where $V(\text{Flip}(p, \text{identity}))$ is the sum of reversed chess value caused by p.

3.3. Dueling DQN

In Dueling DQN [2] considers, The latter layer is not a separate sequence, but is divided into two separate sequences (control flow).The first part is only related to the state S and has nothing to do with the specific action to be adopted.The second part called the Advantage Function is related to both state and action,it focuses on the importance of the action in this state.The Q-value is decided by Advantage value and state value.

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha)$$

In the related paper,it replace the optimal value with the average value of the dominant function to reduce the range of Q and remove the extra degrees of freedom,Q-value is defined as :

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A\left(s, a'; \theta, \alpha\right)\right)$$

The dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. In our network, we apply the dueling network and residul block, advantage value has the global

chess board information and determinate whether or not the state is good for final success or not and the 64×1 value is similar with previous in DQN network representing the value of each position.The network is shown in figure 6

3.4. Minimax Dueling DQN

Dueling DQN has already have great performance on current state, but it still can not consider the future state. Thus, we combine the concept of MiniMax with Dueling DQN, and we name it as MiniMax Dueling DQN. MiniMax Dueling DQN will consider one more step about the opponent by using MiniMax method. The reward of current state is:

$$R_{current}\left(s, a, s'\right) = R\left(p_{current}\right) - R\left(p_{next}\right)$$

where $p_{current}$ is the current action a, and the p_{next} is the next action chosen by opponent.

4. Experiments

4.1. DQN v.s. Minimax

Firstly, we tried to make the DQN agent play against minimax, and the result is shown in the figure 7. The DQN won the game in a large scale, with a winning rate of 90 %. Such result also proved that minimax is not a good way to deal with our game.

4.2. DQN v.s. Dueling DQN

Then, we try to let DQN and Dueling DQN compete against each other, and the result turned out to be surprisingly impressive as shown in figure 8. The Dueling DQN overwhelmingly wins the game, which is shown in the figure. Such result pointed out that the necessity and correctness of our advantage function.

4.3. MiniMax Dueling DQN

We improve Dueling DQN with the concept of MiniMax into MiniMax Dueling DQN. Showing that with depth-one MiniMax tree, the model can consider the opponent how to perform in the next state, and such consideration would help it to avoid being flipped more position in the next state. With the improvement of the one-step future consideration, it will defeat the original model, both play as white and black, as shown in figure 9

No matter in which role, MiniMax Dueling DQN can both keep a higher winning rate as shown in figure 10. Here comes an interesting phenomenon that MiniMax Dueling DQN keeps a further higher winning rate when it is on white side.

The reversi game is actually a asymmetric game. The difference is negligible for two human players or less smart AI, but become increasingly significant when both players are *smart enough*. The black players, who take the first action, has a inherent disadvantage. In the near ly final state or some special state of the game, when the only remaining valid actions are all the *terrible* actions (i.e. taking action here would make it much more likely to lose the game), the black player will have to take an action here, even if it knows that it is a bad choice. Such limitation and asymmetry account for difference.

5. Conclusion

After several algorithm updates, MiniMax Dueling DQN is the most powerful algorithm at present, and the interval time of each drop is very short, so there is almost no waiting time for people. This algorithm not only effectively approximates the q-value through Dueling DQN, but also can consider the advantages of MiniMax in the future step, while avoiding the disadvantages of long calculation time of MiniMax algorithm. There is no battle with the MCTS algorithm in chess, but it is certain that MiniMax Dueling DQN is superior to MCTS in time. The MCTS algorithm will be completed later and the two will be compared.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[2] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.