AI for Draught

Shouchen Zhou, Kecheng Ye, Zhirui Zhang, Penghao Wang
School of Information Science and Technology
ShanghaiTech University
{zhoushch, yekch, zhangzhr4, wangph1}@shanghaitech.edu.cn

Abstract

In this project, we implement several draught game agents using the technique learned in the CS181 course. We also design an evaluation function for the draught game and evaluate its strength in each agent. By bridging theory and implementation, the project enhances problem-solving skills and demonstrates the real-world relevance of AI concepts. It serves as an interdisciplinary example of AI's applicability, illustrating how AI techniques can be employed in gaming and beyond.

Introduction

This project focuses on the game of Draughts, a strategic board game with profound combinatorial complexity. The Draughts, with its simple rules yet intricate strategic depth, offers an ideal testbed for AI algorithms. Our project aims to design and implement a set of diverse AI algorithms, each unique in its approach to decision-making and strategy formulation. These include the Random algorithm, offering baseline performance metrics; the Minimax Search algorithm, a classic approach in game AI emphasizing strategic depth; Reinforcement Learning, which seeks to optimize strategies through trial and error in a dynamic environment; and the Monte-Carlo Tree Search, a probabilistic model known for its effectiveness in dealing with the uncertainty and complexity inherent in Draughts.

State Space

The board is shown in Figure 1, there are a total of $8 \times 8 = 64$ lattices on the board, but only 32 of them are accessible for pieces. Each accessible lattice may be put on a black piece, a white piece, a white king, a black king, or nothing on it. So each accessible lattice may have 5 status. So the total number of state spaces is $5^{32} \approx 10^{22}$.

Motivation

So far, we have learned various intelligence agents in courses, including search, adversarial search, bayesian network, Markov decision model, reinforcement learning, and machine learning. Draught, with simple rules but various states, is suitable for applying these methods to an intelligent agent. With this motivation, we implement random agents, minimax agents, MCTS agents, Q-Learning agents, and approximate Q-Learning agents. Also, we train a Neural Network to evaluate the final score for a state to improve the evaluation function. We compare the performance of each method against nondeterministic methods including random and MCTS and evaluate the strength of our evaluation function of the game state.

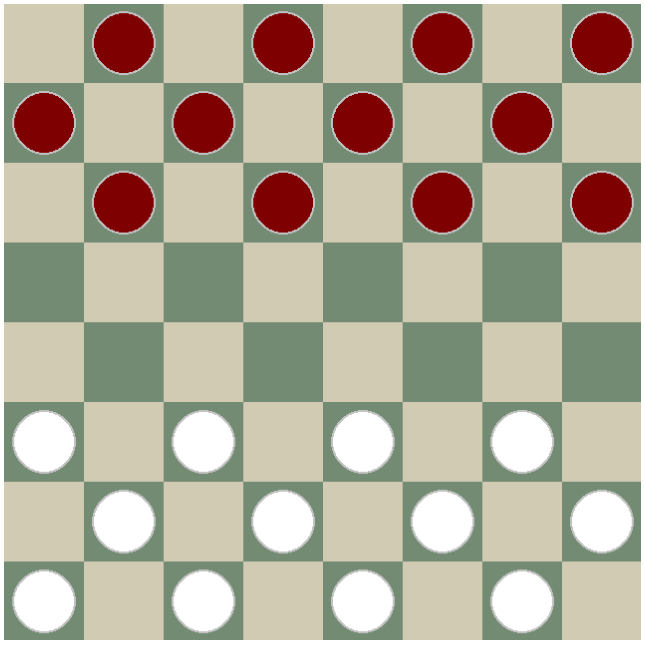
Figure 1: Visulization of draught game board

Methods

In this part, we will make the notion that $transition = (s, a, s')$, which means that the state s will transition to s' if the action is a.

Score Function

In our methods mentioned in the introduction, most of them need a score function to evaluate whether the taken action is preferable or not. As a result, how to design a reasonable and effective score function is significant. During all the different methods, the factors that affect the score function are similar. The influencing factors are:

1. **The number of pieces that survived** Since the condition that we don't lose is that we still have pieces that survive on the board, so we should try to make more of our pieces survive. Meanwhile, our movements aim to eat all the pieces of the opponent, so the fewer of the opponent's pieces are, the better the state is. So the number of pieces that survived, both our pieces and the opponent's pieces, matters.
2. **The number of kings that exist** Since the king can go backward rather than the non-king piece which can only go forward, the number of kings should be considered in the score function. We should try to make the number of our kings bigger and the number of the opponent's kings smaller.
3. **The sum of our all non-king pieces' distances to the opponent's bottom line of the board** Since we need to make more of our kings, the sum of our all non-king pieces' distances to the opponent's bottom line of the board reflects the potential to have more kings. The smaller the sum of distances is, the bigger the potential will be.
4. **The sum of our all pieces' distances to the left or right line of the board** Considering the piece which is next to the left or right line cannot be eaten, we can try to move more pieces to the positions that are next to the left or right line of the board. We can measure this feature by the sum of all pieces' distances to the left or right line of the board, while the distance of pieces next to the left or right line is zero. The smaller the sum of distances is, the better the situation is.

Although different methods differ in the process of getting the best action, as they share the same influencing factors, they can use the same score function to evaluate one state. In most of our methods, the score function we use is:

$$\begin{aligned} f(s) &= \omega_1 * (N_{\text{our-survived}} - N_{\text{opponent-survived}}) \\ &+ \omega_2 * (N_{\text{our-kings}} - N_{\text{opponent-kings}}) \\ &+ \omega_3 * \sum_{\text{our non-king pieces}} \frac{1}{L_{\text{dis-to-bottom}}} \\ &+ \omega_4 * \sum_{\text{our pieces}} \frac{1}{min(L_{\text{dis-to-left}}, L_{\text{dis-to-right}}) + 1} \end{aligned} \tag{1}$$

where N means the number of pieces, and L means the distance between the selected piece and the target line. What's more, since every non-king piece's distance to the bottom line is not zero, the denominator of the third feature is not zero. But the sum of the minimal distance of each piece to the left or right line may be zero. So we can add 1 to the denominator of the fourth feature to avoid the infinity value of the score function. ω_1 to ω_4 is the weights of each feature. In our implementation, the parameters we use are $\omega_1 = 1, \omega_2 = 2, \omega_3 = 1, \omega_4 = 0.5$.

Random

A naive but fast agent for a draught game is a random agent, for a given game board and given turns, we could get all possible moves, then we randomly choose one and perform the move. Note that for other games, random agents will perform extremely poorly. But for draught, due to the special rule, that piece must eat the opponent piece if available and small game board, the random agent's performance is not so bad, and it is an ideal agent for fast testing with other types of agents.

Greedy

A simple improvement of the search agent is a greedy method. For the current state, firstly we get all the valid actions. For each action, we can get the state after movement. Then we can use the score function in section **Score Function** to get the reward of each state. Then we choose the action with the highest score. Thus, we achieve the best local solution of depth one.

Minimax Search

As a two-player game, a straightforward and efficient method to solve the draught is adversarial search. Here we deploy minimax search to draught, each time the agent finds the optimal search with the assumption that the opponent will move optimally. We implement this algorithm recursively, and to limit the time of each search, we limit the maximum search steps. When the algorithm stops the search, we use the score function to evaluate the current game board score to choose the optimal move.

We also apply the alpha-beta pruning algorithm to decrease the nodes of the search tree, by pruning nodes with bad performance, we could increase the search depth in a limited time, thus improving the performance.

Reinforcement Learning

Monte-Carlo Tree Search MCTS (Monte-Carlo Tree Search), a well-known algorithm used in Alpha-GO, could handle large state space problems and complex decision trees. Here we apply MCTS to classical draught games. As shown in figure 2, the MCTS algorithm builds the search tree during the iterations of four steps: Selection, Expansion, Simulation, and Backpropagation. To build the tree, we define our tree structure with tree nodes. For each node, we store the visit time, win time, and parent and children. For the current node, we could use the rule of draught to find all possible movements as the children of the node. Then we explain the details of the four steps.

1. **Selection** We start from the root node of the search tree, representing the current state of the game, where we use board class to store the current game state. Then we use the UCT (Upper Confidence Bound for Tree) selection policy to select child nodes. The UCT is an application of UCB, and the value can be obtained by

$$\operatorname*{arg\,max}_{v' \in \text{children of} v} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2lnN(v)}{N(v')}}$$

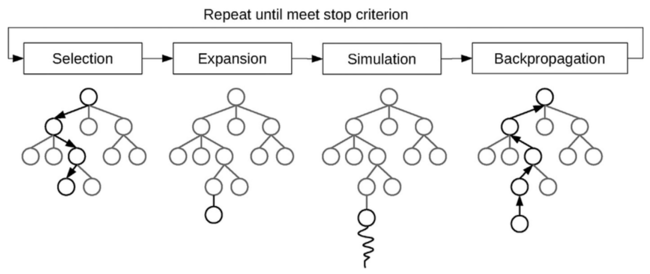

Figure 2: Illustration of the MCTS algorithm steps

where v represents for the parent node, v' is one of the child node of v, N is the nodes' visit time, and Q is the quality value of the current node. Note that c is chosen as $\frac{1}{\sqrt{2}}$ as an empirical constant. Each time we select the node with maximum UCT value, by selecting iteratively, we find the leaf node of the search tree that hasn't been fully expanded as a result of the selection step.

2. **Expansion** After selecting a leaf node, we generate child nodes for the current node of possible future states, and then we add these nodes as the child nodes of the leaf node.
3. **Simulation** After Expansion, we perform a simulation of one of the added child nodes. Here we randomly choose actions in the simulation step to estimate the possible final state of the current node. Note that for the speed of the agent, we will limit the max actions step, so if the node needs large steps to get the final state, we will end this step ahead and regard the player that has more pieces as the winner.
4. **Backpropagation** Lastly, we perform backpropagation through the tree to update all parent nodes of the search path. We update the visit times and the reward of win or lose to each node.

As for an MCTS agent, we first initialize a state class to store the current state of the game, then we perform the four steps for fixed iterations. After building the tree for search, we choose the best child of the current node by finding the maximum layouts and return the corresponding best action. With a larger number of iterations, the tree is built more richly, thus the agent will be more intelligent. However increasing the number of iterations will also result in longer inference time, so a balance between iterations and intelligence is important.

Q-Learning Q-learning is a reinforcement learning approach, we hope we can learn a better strategy while taking action. We define the Q value($Q(s,a)$) as the current state which is represented by the current board, and the action which is represented by the next board because we are dealing with a determinate action. Besides, we use the evaluate function mentioned in section **Score Function** to define the reward in two different ways.

$$r1(s,a,W) = f(s,W) - f(s',W)$$
$$r2(s,a,W) = [f(s,W) - f(s',W)] - [f(s',B) - f(s'',B)]$$

where f is the evaluate function, s is the current board, s' is the next board, s'' is the board after the opponent's action, and we are WHITE as an example. Specifically, we get a positive reward if we eat pieces, move toward the opponent, or win. The two ways differ mainly in whether we consider the opponents' reward. Experiments show that they are almost the same and the second one is a bit better.

Considering the training, we train our agents against random agents. As we take an action, we have a new sample and we use the sample to update our Q-value, the procedure is as follows:

$$sample = r1(s,a,W) + \gamma \max_{a'} Q(s',a')$$
$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(sample)$$

As to the parameters, we also have ideas and experiments. γ is the discounting factor that decides whether we prefer early reward. In Draught, there is a special rule which is "must eat when you can eat", so a strategy that we can sacrifice a piece to control the opponent's action, thus benefiting long-term situation may exist, which makes the discounting factor $\gamma > 1$ possible. Based on this we try the discounting factor from 0.8 to 1.1. The results show that 0.8 works better and we think the main reason lies in that the agent's intelligence now didn't reach the level that may consider such a complex strategy. Moreover, we set ϵ to 0.4 to enable our agent to explore more states. However, when testing, we still have problems that we possibly meet an unvisited Q-state and we can only regard its Q-value as 0.

Approximate Q-Learning Since there are a total of $5^{32} \approx 10^{22}$ states, it is not possible to visit all as long as store all states during training. So we introduce the approximate Q-learning to avoid the side effects of unvisited states.

Our experience is summed up in a few powerful numbers. We take the following experimental value functions for approximate Q-values.

$$f_1(s,a) = \begin{cases} 10, \text{if the WHITE side wins} \\ -10, \text{if the BLACK side wins} \\ 0, \text{otherwise} \end{cases} \tag{2}$$

$$f_2(s,a) = N_{\text{our-survived}} + 2 * N_{\text{our-kings}}$$

all N are numbers for state s

$$f_3(s,a) = \sum_{\text{our non-king pieces} \in s} \frac{1}{L_{\text{dis-to-bottom}}} \tag{3}$$

all L are depth for state s

$$f_4(s,a) = \sum_{\text{our pieces} \in s} \frac{1}{min(L_{\text{dis-to-left}}, L_{\text{dis-to-right}}) + 1}$$

Where N means the number of pieces, and L means the distance between the selected piece and the target line. All former formulas are only considered the state s.

We also need to take the state after the transition into consideration, by setting

$$f_5(s,a) = -f_1(s,a)$$

$$f_6(s,a) = N_{\text{opponent-survived}} + 2 * N_{\text{opponent-kings}}$$

all N are numbers for state s'

$$f_7(s,a) = \sum_{\text{our non-king pieces} \in s'} \frac{1}{L_{\text{dis-to-bottom}}} \tag{4}$$

all L are depth for state s'

$$f_8(s,a) = \sum_{\text{our pieces} \in s'} \frac{1}{min(L_{\text{dis-to-left}}, L_{\text{dis-to-right}}) + 1}$$

With the constructed value functions for approximate Q-values above, we could train with the method of approximate Q-learning:

$$sample = \sum_{i=1}^{8} w_i f_i(s,a)$$
$$difference = sample - Q(s,a)$$
$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(sample)$$
$$w_i \leftarrow w_i + \alpha(difference) f_i(s,a)$$

For the training details, we set the discount factor $\gamma = 0.8$, and all weights w_i are initially set to be 10. We firstly pre-train the model, we totally trained 3000 epochs, with a learning rate to be 0.01, and after the 1000-th epoch, the learning rate times 0.99 after each epoch. The opponent to be the minimax search with depth 3.

Then we do finetune for the pre-trained model, similar to the pre-training, but we set the learning rate to be 0.001 at initial, and the opponent to be the minimax search with depth 5.

After finetuning, we tested the winning rate for our trained approximate Q-learning model, the winning rate has a great increase compared to the pretraining model. The winning rate of Approximate Q-learning and random increases from 89% increase to 91%.

Neural Network Since testing the Minimax algorithm could generate a large amount of battle information, we want to make good use of it. By using the previous battle information to train a Neural Network, and generate a new feature for the **Approximate Q-learning**.

We collected the training data by storing the match information during testing the winning rate between Minimax with depth 7 and another Minimax with depth 7. We store each board's information and the turn of the player during the matching. After matching, the board that was generated by the winner is set to have the label $+1$, and the board that was generated by the loser is set to have the label -1. Also, the board's generator was also recorded.

With the generated data, we could do training and inference to generate the new feature for approximate Q-learning.

1. **Training** Throw our observation during the matchings, almost all capture operations happened within 2 steps. This means that in nearly all situations if a piece eats the opponent's pieces, it will not eat more than 2 pieces. So if we consider the board as an image, a convolution kernel with 5×5 receptive field is enough.

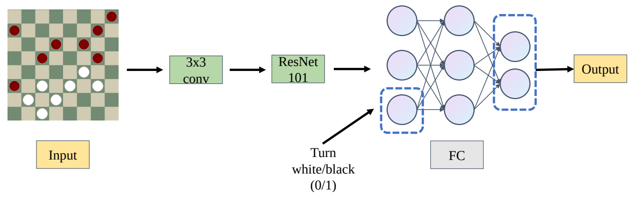

Figure 3: The pipeline of the Neural Network to generate new features.

After the convolution layer, we set ResNet101 and a 3-layer fully connected MLP to do the 2 classification task. The output layer includes two numbers since we are doing the 2 classification, so the final output $+1$ or -1 is the prediction of the new state's winning rate. And we set this to be the new feature for approximate Q-learning.

The total pipeline is shown in 3.

2. **Inference** To generate the new feature, we put the origin board s, and the board s' after action a respectively into the network, and infer the winning rate. The winning rates are set to be the new feature $f_{new}(s,a)$.

After generating the new feature, we put it into the original **Approximate Q-learning**, all settings are the same instead of adding a feature. And we could see that the winning rate has increased. The details are shown in Section **Result**.

Results

Below are the results of our experiments, including our main methods: minimax, MCTS, Q-learning, approximate Q-learning against our MCTS, and the baseline random agent. In the final test, we play 100 games and count the number of winning games. The first discovery we made is that there exists a large advantage to being the Sente(start first), so for every pair of comparisons, we switch sides to have a deep assessment of the algorithm's performance.

As is shown in the tables, all our implementations perform better than the random baseline. Combing the performance of Sente and the performance of Gote, we can rank these methods: Minimax $>$MCTS $>$approximate Q-Learning $>$Q-Learning $>$random, which proves that our idea that Q-Learning is not able to visit enough states is right. Approximate Q-Learning achieves a better performance while using less memory.

In Table 3 and Table 4, we show the result after we add a neural prior as a feature in approximate Q-Learning, which demonstrates improvements especially when being Gote.

External Resources

We use Python to implement this project, using PyGame for GUI and game logic, Numpy for fast computation, and PyTorch for the Neural Network including convolution network, ResNet, and Linear network used in the improved Q-Learning.

Gote \ Sente	Random	Search	Monte-Carlo Tree Search	Q-Learning	Approximate Q-Learning
Random	52%	100%	93%	85%	89%
Monte-Carlo Tree Search	19%	87%	81%	59%	63%

Table 1: The comparison between different methods, note that the winning rate is for the Sente agent

Sente \ Gote	Random	Search	Monte-Carlo Tree Search	Q-Learning	Approximate Q-Learning
Random	48%	87%	82%	74%	68%
Monte-Carlo Tree Search	7%	39%	19%	16%	19%

Table 2: The comparison between different methods, note that the winning rate is for the Gote agent

Gote \ Sente	Approximate Q-Learning w/o neural prior	Approximate Q-Learning with neural prior
Random	89%	91%
MCTS	63%	67%

Table 3: The comparison of Q-Learning with and without neural prior, note that the winning rate is for the Sente agent

Sente \ Gote	Approximate Q-Learning w/o neural prior	Approximate Q-Learning with neural prior
Random	68%	76%
MCTS	19%	18%

Table 4: The comparison of Q-Learning with and without neural prior, note that the winning rate is for the Gote agent