COMP90054 Group Project Report
Yinsh Game

Qingyang Feng 980940 Marco Liu 1339143

Oral Presentation Link

https://youtube.com/playlist?list=PLdPVbMYb0R8b5aXRY3qpmnm0l64PMF-yu

1 Introduction
This report describes the development of an autonomous agent that can play and compete in a tournament for the game Yinsh.

2 Techqiues
2.1 Deep Q-learning
2.1.1 Methodology
Deep Q-learning is an offline model-free reinforcement learning technique. It uses deep neural networks for function approximation. This technique is selected as Yinsh has a large action space and state space. The use of function approximation will avoid keeping a large Q-table. Additionally, since reachable states in the large state space may be sparse, compared to Q-table, Q-function is expected to provide estimates when unreached states are encountered.

2.1.2 Algorithm
A framework similar to the Multi-agent Q-learning algorithm discussed in the lecture is implemented. The algorithm is repeated over a large number of episodes of the game and actions are selected using the ϵ-greedy algorithm and the Q-function.

2.1.3 Q-function representation
Q-functions are represented using a deep neural network. The architecture of the neural network:

1. The first layer takes the features of the state. In our case, the state is encoded as an array of integers with a length of 85. Each element of the array represents the state of a legal position on the board, denoted by: EMPTY=0, RING_SELF=1, CNTR_SELF=2, RING_OPPO= -1, CNTR_OPPO= -2
2. The second layer is a hidden layer, with a number of 64 hidden dimensions
3. The final layer is the output layer, with the same dimensionality as the action space. As there are four types of actions and each action is related to the place/move/remove positions, the action space of the game is large. To deal with large dimensionality and make the algorithm learn faster, the following method of mapping is used to reduce the dimensionality of the action space: 0 ~ 84: actions with type 'place ring', there are 85 possible actions of this type; 85+0 ~ 85+53: actions with type 'place and move', there are three possible lines for this type of action to move along (vertical, horizontal, and diagonal) and depending on the moving direction of action, there are 18 possible positions to move to; 85+54+0 ~ 85+54+53: actions with type 'place, move, remove'. Similar to 'place and move'; 85+54+54: actions with type 'pass'. There are 194 different actions in total.
4. Non-linear ReLU is used between layers

2.1.4 Q-function update and Reward Shaping
The parameters of the neural network are updated as follows

1) $\theta \leftarrow \theta + \alpha \times \delta \times \nabla_{\theta} Q(s, a; \theta)$
2) $Q(s, a) \leftarrow Q(s, a) + \alpha \times \delta$
3) $\delta \leftarrow r + \gamma[\max_{a'} Q(s', a') + \Phi(s')] - [Q(s, a) + \Phi(s)]$

Where r is the reward, γ is the discount factor, α is the learning rate and $\Phi(s)$ is the potential of state s. Instead of using the immediate state after our action is executed, the next state we used to update our Q-function with is the state we observe in our next turn after the actions from the other agents are applied. The reward we used is the number of additional rings the agent wins by the action. However, this reward is sparse, as only a small amount of actions will lead to this reward. To help the algorithm converge faster, we used potential-based reward shaping (Φ). The potential function is defined as follows:$\Phi(s) = w_0 \times cntrNum + w_1 \times avgSeqLen + w_2 \times avilActionNum$

Where $cntrNum$ is the number of counters left, $avgSeqLen$ is the average length of the sequences formed by the self agent, $avilActionNum$ is the number of available action for the self agent at state s and w_0, w_1, w_2 are the weights associated with them.These features are believed to provide intermediate reward information leading to the actual reward state. States with larger $cntrNum$ and $avgSeqLen$ will have a larger chance of forming a row and larger $avilActionNum$ will provide more mobility for the next action.

2.1.5 Experiment Setup

The player we used to train the Q-function is an agent employing the Alpha-Beta Pruning Algorithm. The discount factor is set to 0.9 and ϵ in the ϵ-greedy policy is set to 0.1. Different values for the learning rate and the weights of the potential function are tested.

2.1.5 Result and Analysis

The policies produced by different parameter settings are competed against a random agent as a baseline. The competing results of running 100 games are provided in Table 1 in the appendix Learning rates 0.1 and 0.01 are experimented. Considering a smaller learning rate may take longer to converge, numbers of 100 episodes and 1000 episodes were experimented. However, it is noticed that increasing the number of training episodes for $\alpha = 0.01$ does not improve the winning rate and $\alpha = 0.1$ provided a better performance. This suggested that a learning rate of 0.01 is too small and may cause the solution to be stuck at the local optima. Different sets of feature weights were also experimented and the parameters that provide the best performance is $\alpha = 0.1$, $(w_0, w_1, w_2) = (0.05, 0.15, 0.05)$ and the resulted policy is used for evaluation against other techniques in the following sections.

The main challenge we encountered when implementing this technique was to come up with a good design of the potential function therefore it is hard to get a good approximation. If there is more time available, more feature engineering will be implemented to capture intermediate rewards.

Strength:

- Low on the fly thinking time: the policy is pre-trained and loaded when playing games
- No feature engineering for state representation
- Estimate available for unreached states during training
- Efficient memory: don't need to store all state-action pairs in a Q-table

Weakness:

- Convergence to the optimal solution is not guaranteed
- A large amount of data and computation is involved during training, thus is slow to train

2.2 Monte Carlo Tree Search

2.2.1 Methodology

Monte Carlo Tree Search (MCTS) is an online decision approach, which is invoked each time a new state is visited by the agent during the search. MCTS has some features that make it suitable for a chess game. Firstly, MCTS is efficient in solving problems with huge state space, where calculating the value of all subtrees is impossible. By combining with Upper Confidence Bounds (UCB), it will take into account both exploration and exploitation, so as to avoid falling into the local optimal solution. Moreover, it is an anytime algorithm which can be terminated at any time and give the best answer found so far, and it can operate effectively without any knowledge in the particular domain.

Based on the above advantages of MCTS, we regard it as a hopeful method for Yinsh game and decide to implement it. Firstly, Yinsh is a zero-sum, pefect information, determinism, sequential, and discrete game where MCTS is appliable. Also, Yinsh has a large state space, especially in early states, and MCTS could be efficient dealing with large state spaces. What's more, our game has a strict time limit of 1 second per round, and MCTS can give us the best action it has met before reaching the time limit.

2.2.2 Algorithm

The MCTS algorithm has four steps which are iterated over within the time limit to incrementally build up the tree of evaluated states.
-Selection: For selection, we return the root node itself if it is not fully expanded or it is the end state of the game. Otherwise, we use UCB to select an action and the corresponding child node and repeat the selection on the the corresponding child node.
-Expansion: After selecting a node, we should choose a legal and unexpanded action of the node to proceed. Through experiments, we find that due to the exsitence of the time limit, the number of nodes we expand each round can be very few in the early stage of a game. If we randomly select a descendant to expand, we are unlikely to select the most promising one. To improve the performance of the algorithom, we use heuristic functions and reward shaping to help with selection.
-Simulation: As the main step of MCTS, simulation takes up most of the thinking time in each round. Normally, a random rollout is played till the game terminates. Due to the time limit, we have to set up the max depth (the max number of rounds) of simulation. We have tried different value for the max depth, from 10 to 35, through experiments, and we found that a simulation depth of 20 is the most suitable for the game. Apart from the randomised simulation, we also tried to exploit the heuristic function to help select actions. However, it will significantly increase the simulation time and hence affect the performance of the agent. Therefore, we choose to apply the randomised simulation.
-Back-propagation: In this step, we backpropagate the reward of simulation to the ancestors recursively using a Q table. We have tried different discount factors, snd we finally decide to use 0.8 as the discount factor.
-Heuristic Function and Reward Shaping:
A reasonable heuristic funtion can guide us towards good actions. As we mentioned above, we use a heuristic funtions and reward shaping to help with nodes expansion. In this game, the rewards are sparse since the players only get scores when the the rings are removed. To help the algorithm converge more quickly, we augment our reward function to reward behaviors that we think may help us win the game and give punishments when unflavorable patterns are found. The reward for a particular game state consists of three parts:

1. Successive rings and counters: We traverse the game state and found all the successive rings and counters (at least 3) in a row belonging to a single player. For example, the ring and counters in Fig.1 become a row of 5. A longer row represents a higher reward. Also, in each row, a counter represents a higher reward than a ring, which can tell the agent to replace the ring in a row with a counter.
2. The place of rings and counters on the board: Since each state can have many actions to proceed, we need to prune some actions that we think are most promising. We think it may be better to occupy the places close to the center of the board. Hence, we distribute reward according to the place of rings and counters of the player. We divide the board into 4 areas, and the area closer to the center represents the higher reward.
3. Patterns: We want to find if there exists any particular patterns that is favorable to the opponent and give punishment if we find any.

A heuristic function is also used to help place rings. At the start of the game, since there is no good information to exploit, the agent may randomly place the ring. We think it is a better move to place the ring around the center of the board, and we use a heuristic function to guide the agent.

2.2.3 Analysis:
-Challenges Experienced: For MCTS in Yinsh, two major challenges I have met are time limit and reward shaping. Firstly, the thinking time is strictly limited within 1 second, which can greatly decrease the performance of the agent since we don't have enough simulations to converge to the best solution. In order to deal with this problem, we try to reduce the simulation time by using max depth and random simulation. Also, we use the heuristic to find the most promising node to expand first.
Reward shaping is the most critical and difficult part in Yinsh, which also greatly affects the performance of the agent. Finding a good and useful reward or heuristic fuction is never easy, and a bad heuristic function may even decrease the performance. Improving the reward function needs a lot of effort and exploration. We need to observation the results of the experiments, extract the patterns

and information we want, think about the reason it work or not, and modify our reward function accordingly.
-Strengths and weaknesses: We have already discussed the strength of MCTS in 2.2.1. MCTS can converge to optimal providing infinite memory and computation time. However, in this game, we have a strict time limit of 1 second. We may require a large number of iterations to converge to a good solution. Key nodes are not visited for enough times to promote the algorithm to give a reasonable solution. We have found that, given more thinking time, MCTS outperforms other methods.
Also, MCTS may not be able to detect shallow traps, where opponents can win within a few moves, as well as minimax search. Thus, minimax search performs better than MCTS in games like Chess, which can end instantly (king is captured). In addition, MCTS may not be able to detect shallow traps where the opponent may win in a few steps, and it may fall into local optimization easily.
-Improvements: Since reward shaping is critical to improve the performance of the algorithm, the future improvements mainly lie on getting a better reward and heuristic function. We may need to extract more usful patterns and justify our reward function, especially the proportion of different parts.

2.3 Alpha-beta Pruning Minimax Algorithm

2.3.1 Methodology

Minimax is a backtracking algorithm used in game theory to find the best movement, assuming that the opponent also plays optimally. Minimax algorithm is powerful for simple games, but may be computationally infeasible for more complex games. Alpha-beta pruning is a process that reduces the amount of computation and search during minimax, which is conducted in a depth first manner. Comparing with minimax, the alpha beta pruning deletes all nodes that will not affect the final decision and thus speed up the search. We also implement Alpha-beta Pruning Minimax Algorithm in Yinsh.

2.3.2 Analysis:

-Challenges Experienced: The time limit is also a big challenge for Alpha-beta algorithm. For Alpha-beta, we need to determine the max depth we want to search. Normally, the deeper the algorithm reaches, the better it performs. However, since the algorithm is performed in a depth-first fashion, a lager depth may lead to less children of the root node visited. Hence, we need to trade off between the depth and the width.
Through experiments, we find that it's suitable to set max depth as 5.
-Strengths and weaknesses: Alpha-beta minimax is fast and suitable for simple games like Tic-Tac-Toe. However, it may not very powerful for game with bigger state spaces like Yinsh. Also, we have to limit the depth due to the time limit, which has also weaken the algorithm.

2.4 Greedy BFS Algorithm

2.4.1 Methodology

Through experiments, we have found that BFS is very powerful for this game. Hence, we are thinking of combining BFS with the heuristic or reward we write. That is, the algorithm will make use of the heuristic to decide whether to do the action or not.

2.3.2 Analysis:

The Greedy BFS performs well, since it combines both the advantages of BFS and heuristic. It is fast and make use of the heuristic function we write.

3 Results and Evaluation

In order to compare the performance among the techniques we implemented. Each agent we developed is competed with the provided BFS agent. 20 games was ran for each competition and the results are displayed in Table 2 in the appendix. Finally, we decide to use Greedy BFS as our agent for final tournament.

Appendix

Table 1:

α	w_0	w_1	w_2	episode	Average Score	Winning Rate
0.1	**0.05**	**0.15**	**0.05**	**100**	**1.92**	**94%**
0.01	0.05	0.15	0.05	100	1.57	86%
0.01	0.05	0.15	0.05	1000	1.41	87%
0.1	0.1	0.1	0.1	100	1.44	87%
0.1	0.1	0	0.1	100	2.01	93%
0.1	0.05	0.3	0.1	100	1.75	88%

Table 2:

Teal	Magenta	Teal (average score /winning rate)	Magenta (average score /winning rate)
DeepQ	BFS	1.25 / 25%	2.45 / 85%
AlphaBeta	BFS	0.70 / 20%	2.50 / 90%
AlphaBeta (with Heuristic)	BFS	1.3 / 35%	2.55 / 80%
MCTS	BFS	1.45 / 25%	2.55 / 80%
BFS	Greedy BFS	1.50 / 40%	2.0 / 60%