COMP30027 Artificial Intelligence Project Part B Report

Xinyao Niu(900721), Maoting Zuo(901116)

1 Project Structure

We divided our project into two main parts to achieve high cohesion and low coupling code design. One (mainly took care by Maoting) take care of running and response to referee and collecting useful information from the game which might be useful for the other part. The other one (mainly took care by Xinyao) are responsible for making decision based on given information. Overall, we implemented 6 different version of this project. After merging the similar package, we have up to 4 unique package of work which one of them is based on deep reinforcement learning and the other three are mainly based on classical MaxN and hand-tuning approach. In this section, I will only discuss the final version that we are going to submit as an example of all classical approach (we just call it HardCode)[1] and only important files in deep reinforcement learning to illustrate the idea we implemented.

1.1 HardCode

This section will discuss important files which implement in classical approach. Our final file named VanGame will be a example. All packages implementing classical strategy have similar structures.

File Name	Usage
__init__	python package required
compatNode.py	node for search
config.py	constants for the project
maxn.py	maxn search
player.py	handling different player
strategy.py	manage other functions
utils.py	frequently used function

We use to have one extra file call logger, which log every game state and write in a file for training purpose. Since we found out that using DRL in this project could not achieve our goal, we removed all related components when we are polishing our code.

[1] notice that not all file are as polish as the submitted version

1.2 Deep Reinforcement Learning (DRL)

Only two extra part needs to mention, which is keras_model.py. In this file, I implement a approach which could read the feedforward neural network weights from a file and build up a model to give prediction based on given input. The is the essential part related to decision making, which represent our evaluation function. All auto-training procedures are build up in jupyter-notebook in outer file. The other important part is logger, which is the other essential part of the whole auto-training process. It could record all the information required for training turn by turn.

2 Decision Making

In this section, I will briefly discuss how we are going to handle 3-player situation and how we are going to decide which action we are taking. Also, an intuitive explanation of our evaluation function won't be neglected for sure.

2.1 3-Player Problem

To deal with 3 player player problem, we decided to upgrade min-max tree to MaxN tree. It could handle arbitrary number of player, but the cost to explore the tree growth exponentially. Another bad news is that we could not use the original pruning method which works really efficient in min-max as it is a binary case. I will talk about the pruning method we used in later section.

2.2 Strategy For Choosing action

The strategy for choosing the best action are same in both classical method and DRL. There are a lot possible action for each stage. We will use our evaluation function to evaluate how

good is a action. Then, will pick the one with best score.

2.3 Evaluation Function

Using evaluation function just like using mathematical approach to represent a game, and based on the result to make decisions. In this project, we use 7 parameters in total and some sophisticated combination to come up with a piece-wise function which trying to describe the whole game based on that.

The following parameters are what we used in the game, to describe the game when we look at our one step further action:

Define:

- T = number of pieces that already exit + number of piece on the board still under our control

1. **pieces_difference**
   Are we going to gain or lose any piece? This figure can be only one of [-1, 0, 1].
2. **reduced_heuristic**
   Reduced in heuristic value. In part B the heuristic is much complicate compare to part A. It is abstract measurement of how many steps are we going to take in order to win the game. If we have $T < 4$, for each 1 smaller, we add 15 to the total score. For $T > 4$, we only calculated the closed 4.
3. **danger_piece** How many pieces could be eaten by others after one turn. Notice that we subtract 1 if we gain a new piece from others to prevent the bot are too afraid of eating other players pieces. Even though, the value could only be 0 or positive.
4. **player_exit** If it equals 4, that means we win the game. Therefore, we assign it with a extremely large score to make sure that we are definitely going to take that action.
5. **other_reduced_hueristic** Similar as reduced heuristic, but it becomes a measurement of other players. Intuitively, it is a measurement of how many problems we give our opponent when we are going this way.
6. **turn** Number of turn so far for the game.
7. **close** How close are our pieces to each other. If we have $T >= 4$, we only consider the $4-$ number of already exit piece. Otherwise, all the pieces will be considered.

When tuning the weights, we look at each stage ourselves and think what value we expect our model to predict and what decision will we make in that circumstance.
We divided the evaluation function into three major parts, they are $T < 4$, $T == 4$ and $T > 4$. For each case, we further divided into two parts which are before 15 turns and after 15 turns. The basic rules works as follow:

1. group up and move together at early stage. Empirically, group up are less likely to be eaten and can move fast at early stage. But it may become a bad constrain in later stage. Thus, this constrain become loss after few turns by assign it a smaller weights with turn > 15.
2. Don't be eaten if $T \leq 4$. Try your best to eat other if $T < 4$. Only eat other piece only if you could create them a lot problems.

For more details of the weights, refer to the comment in the code file.

3 The Use of Machine Learning

We put a lot effort on implementing and fine-tuning the DRL method. We use logger to record the game state, and assign -100 for losing of a game, 100 for winning. Then, use $\lambda = 0.8$ as decay rate to update the reward for previous states. However, after about more than 20k matches based on self-playing, the ability of the bot still couldn't match the player based on greedy search.
We primarily trying to feed the map to feedforward NN, whereas, it convergence really slow. Consequently, we have to make some "handmade" features, which are quit similar as the previous selection one. Our aim turn to find a sophisticated weight for those features. However, the game still convergence really slow. After that, we think of adding some extra reward to the game to guide our model converge faster. If on of our a pieces got eaten and $T < 4$, we give it -20 (from $\frac{100}{4}$) reward. Additionally, at the end of the game, we don't give it 100, instead, we punish it with -20 number of piece that haven't exit. After another 10k game against greedy bot, it could win 40% of the match. Unfortunately, due to time limited

for this project, we don't have chance to dig further.

4 Comparison Between Different Strategies

All our model could satisfy the time and memory limit for even 256 game turns. The fastest one is greedy, which only take less than 0.1s to make decision. The other two HardCode with MaxN version could make decision within 0.5 sec on average. The one use deep learning could make decision around 0.1s which is just a little bit longer than greedy strategy.

5 Optimization for MaxN

This is actually the core part of our strategy, but our strategy are relatively simple. In terms of saving memory, we create nodes only during the time of MaxN. For MaxN pruning, we use our evaluation function to chop off the worst half of action, as they wouldn't be even considered anyway. By doing this, we could simply half the branching factors. For each layer in MaxN, we use the greedy evaluation function to evaluate how good is each action in terms of who is going to take action at that turn and then propagate back the final choice to its parents. Notice that we don't really have a "tree" structure, we use recursive call to deal with parents and children hierarchy which could save a lot of memory.

6 Things we learnt from this project

In conclusion, what we learn from this project is that classical method could still perform really well if you only have limited time and resources, and reinforcement learning method take really long time to see some feedbacks, you must be patience for that. The reason for that is reinforcement learning has too many hyper-parameters that we need to tune. Last but not least, hope you enjoy reading this report, have a nice day.