AIML402: Assignment 2 report

Junyi Shen (8386129)
September 15, 2024

The assignment is to develop a reinforcement learning agent to play the game of Raj and compete with other agents, including the random agent, value agent, and valueplus agent. In this report, I will introduce the strategies I use, explain how I trained and improved my agent, and illustrate the performance of my agent under different test cases.

1 Strategy

The approach to developing the reinforcement learning agent is to train the agent and update the Q-table, then use the Q-table values to determine which actions to take when competing with other agents. In the training process, There are many things to consider when developing the reinforcement learning agent, including how to determine the state, how to design the reward function, what opponents should be used to train my agent, and how to determine parameters T(how explorative it is), γ(discount on future reward), and α(learning rate). In this report, I will discuss each of these issues and explain why I chose my methods to implement the agent.

In the submitted training file, the agent was trained under 200,000 games(approximately 218 seconds on my Macbook Pro) to make it more competitive. But in the developing phrase, each single test is based on 10,000 games training in order to compare different approaches and parameters in a short time(about 10 seconds), so that I can quickly test and compare lots of different approaches and parameters. As a result, we can see that in this report, the agent acts even better in the test phase than in the development phase, since the final agent was trained under 200,000 games instead of 10,000 games.

1.1 How to determine the state

The state of my reinforcement learning agent consists of three elements:

(1) The item bidding on

(2) The items left

(3) My cards

The reason I chose these three elements as the state is as follows:

The item bidding on and my cards are the most crucial parts of the bidding game and are necessary elements of the state. In addition, knowing the items left will help the agent consider future items to bid on and is helpful for the agent to develop strategies (such as not using too large cards for small items).

I excluded the following elements from the state:

(1) My bank

(2) Opponents' banks

(3) Opponents' cards

State	my_rl_agent	random_agent	random_agent	State in Q-table percentage
bidding_on/ items_left	2.46	2.25	2.26	99%
bidding_on/ items_left/ my_cards	2.86	2.06	2.05	99%
bidding_on/ items_left/ my_cards/ bank	2.65	2.12	2.2	93%
bidding_on/ items_left/ my_cards/ opponents_cards	2.47	2.23	2.26	45%

Table 1: Comparison of difference states, under 10,000 games training and 5,000 games testing

The reason I didn't include these elements is that my bank and the opponent's banks are not important for which action to take in a particular state. When we are making a decision in a bidding game, we will focus more on the items and cards instead of banks.

Although opponents' cards can be considered as a useful factor, I still excluded it because the state space will become too large and we cannot update most values in the Q-table in a reasonable training time.

I also printed the percentage of states that we can find in the Q-table when testing, so that we can know if most of the states are covered in the training process. If the percentage is too low, it means there are too many states in the state space and we should simplify the state.

As Table 1 shows, the state consists of bidding_on, items_left, and my_cards has the best performance.

1.2 How to design the reward function

To design the reward function, I considered several factors. Firstly, the terminal states should provide a large reward(or large penalty) based on the banks of players. A larger bank of the player and lower banks of opponents will result in a larger reward and vice versa. Secondly, for the non-terminal states, we should still give the agent a small reward(or a small penalty) based on the current banks and current cards. If the agent gets a larger bank but only uses small values of cards, it should be rewarded. In contrast, if the agent uses big value cards but only gets a small bank, it should be penalized.

Since the percepts in the AgentFunction don't provide the opponents' banks, I calculate the opponents' banks by comparing the current percepts and the previous percepts, which allows me to calculate the bank differences between players.

State	my_rl_agent	random_agent	random_agent
(a)	2.52	2.2	2.24
(b)	2.42	2.25	2.3
(c)	2.57	2.16	2.24
(d)	2.66	2.17	2.14
(e)	2.65	2.14	2.17
(f)	2.94	2.03	1.99

Table 2: Comparison of difference reward functions, under 10,000 games training and 5,000 games testing

To evaluate how good the cards in hand are, I calculate the card score by subtracting the average hand card values from the average initial hand card values. Then multiply the result with the number of hand cards left:

card_score = (avg_hand_card_value - avg_init_hand_card_value) * len(my_cards)

I compared the performance of the following different reward functions under a 10,000 training and 5,000 testing game setting:

(a) Only get *reward = mybank* in the terminal state

(b) Only get *reward = (mybank * 2 - opponents_bank1 - opponents_bank2)* in the terminal state

(c) Uses (a) as the terminal state reward, and give 1/10 of the reward to the non-terminal states

(d) Uses (b) as the terminal state reward, and give 1/10 of the reward to the non-terminal states

(e) Uses (a) as the terminal state reward, and give 1/10 of the sum of reward and card_score to the non-terminal states

(f) Uses (b) as the terminal state reward, and give 1/10 of the sum of reward and card_score to the non-terminal states

The result shows that (f) performs the best, since it not only considers my bank and the opponents' bank, but also gives a small amount of reward related to cards and banks to the non-terminal states (Table 2).

1.3 How to determine the parameters T, γ, and α

There are three parameters that can influence the behavior of the reinforcement learning agent:

(1). T: (T > 0) Temperature parameter. It is used in the softmax function. The larger T is, the more explorative the agent is and less willing it is to choose the action with a larger Q-value. In contrast, the smaller T is, the more exploitative the agent will be.

(2). γ: $(0 < \gamma \leq 1)$ Future reward discount. It is the percentage of rewards of the next state pass to the current state. The larger γ is, the more the agent will care about long-term rewards.

(3). α: $(\alpha > 0)$ Learning rate. It is the percentage of new information updates to old information. The larger α is, the faster the agent updates its Q-table but it would be more prone to overfitting. In contrast, the smaller α is, the more conservative the agent will be.

To determine the parameters T, γ, and α, I tried multiple combinations of these three parameters. In each test, I adjusted one parameter slightly and tried to compare the performance with previous tests.

Learning rate α performs differently under different training times. Lower α needs more training to perform better. Therefore, I tested many different combinations of three parameters, and for the combinations with good performance, I tried more training games(100,000 times) and decreased α to see if the performance could be improved.

Eventually, I found that T = 0.3, $\gamma = 0.7$, $\alpha = 0.1$ or 0.2 under 100,000 games of training can perform well and got an average score of 3.42 competing with two random agents. I tried to adjust T to 0.2 and 0.4, γ to 0.6 and 0.8, but I did not get a better result. Consequently, T = 0.3, $\gamma = 0.7$, $\alpha = 0.1$ or 0.2 under 100,000 games of training should be a combination that is relatively good. Of course, this combination is not necessarily the best, because there are a lot of randomness in the training process (Table 3).

Note that if we want to train my reinforcement learning agent under small number of training games (for example, only 5,000 times), we need to increase α to 0.3 or 0.4 for better performance.

1.4 With which agents and how many times to train

I found it very easy to beat the value agent or the valueplus agent. This is because they have fixed bidding patterns without any randomness. As a result, a good action is more likely to lead to a good result when competing with agents with fixed patterns. However, random agents are harder to beat because their behaviors are unpredictable and it is more difficult to build up a relationship between different states.

To train an agent that can compete well against various opponents, we can try to train it with different agents with different training times. I trained my reinforcement learning agent with different training session combinations. However, the result showed that training against two random agents for 100,000 times resulted in better performance than training against different agents.

With a training rate $\alpha = 0.2$, increasing the number of training times to 200,000 doesn't perform as well as training for 100,000 times. However, decreasing the learning rate α to 0.1 when training 200,000 times can achieve a better result when competing with random agents. This shows that when we train for a large number of training times, a small learning rate is more suitable.

The result shows that by training 100,000 times with $\alpha = 0.2$ or 200,000 times with $\alpha = 0.1$ against two random agents can both produce a relatively good performance in a reasonable time(approximately 108 seconds for 100,000 times training and 218 seconds for 200,000 times on my Macbook) (Table 4).

T	γ	α	Training Games	Test Games	my_rl_agent	random_agent	random_agent
0.2	0.7	0.3	10,000	5,000	2.78	2.14	2.05
0.2	0.7	0.5	10,000	5,000	2.82	2.06	2.09
0.2	0.7	0.7	10,000	5,000	2.69	2.08	2.19
0.2	0.8	0.3	10,000	5,000	2.79	2.06	2.11
0.2	0.8	0.5	10,000	5,000	2.83	2.06	2.08
0.2	0.8	0.7	10,000	5,000	2.83	2.04	2.1
0.2	0.9	0.3	10,000	5,000	2.82	2.08	2.06
0.2	0.9	0.5	10,000	5,000	2.76	2.05	2.15
0.2	0.9	0.7	10,000	5,000	2.78	2.08	2.1
0.6	0.8	0.3	10,000	5,000	2.99	1.94	2.02
0.6	0.8	0.7	10,000	5,000	2.74	2.15	2.08
0.6	0.6	0.3	10,000	5,000	3.01	1.98	1.97
0.2	0.6	0.3	10,000	5,000	2.81	2.08	2.07
0.4	0.6	0.3	10,000	5,000	3.09	1.96	1.91
0.4	0.4	0.3	10,000	5,000	2.94	2.03	1.99
0.4	0.8	0.3	10,000	5,000	3.06	1.99	1.92
0.4	0.6	0.5	10,000	5,000	2.91	2.06	2.01
0.4	0.6	0.2	10,000	5,000	2.97	1.98	2.03
0.2	0.7	0.2	100,000	5,000	3.28	1.86	1.82
0.4	0.6	0.3	100,000	5,000	3.17	1.91	1.89
0.3	0.7	0.2	100,000	5,000	3.42	1.74	1.8
0.3	0.7	0.1	100,000	5,000	3.42	1.75	1.81
0.4	0.6	0.1	100,000	5,000	3.24	1.85	1.88
0.2	0.7	0.1	100,000	5,000	3.25	1.9	1.81
0.4	0.7	0.1	100,000	5,000	3.32	1.82	1.83
0.2	0.8	0.1	100,000	5,000	3.17	1.91	1.88
0.2	0.6	0.1	100,000	5,000	3.39	1.89	1.69
0.2	0.7	0.05	100,000	5,000	3.15	1.91	1.9

Table 3: Comparison of difference combination of T, γ, and α, under different number of training games

Training times with value&valueplus	Training times with two random agents	α	Test 5,000 games with value&valueplus	Test 5,000 games with two random agents	Training time (seconds)
50,000	50,000	0.2	4.72/1.34/0.64	3.42/1.74/1.8	109
25,000	75,000	0.2	4.50/1.41/0.80	3.44/1.79/1.73	108
0	100,000	0.2	4.85/0.94/0.71	3.57/1.68/1.72	108
0	200,000	0.2	4.77/1.11/0.90	3.55/1.72/1.69	217
0	200,000	0.1	4.69/1.12/0.90	3.68/1.62/1.66	218

Table 4: Comparison of training with different agents under different number of training games

2 Agent performance

I chose an agent with the following characteristics to test its performance, which has a relatively good result under various settings:

(1) State: bidding_on/items_left/my_cards

(2) Reward function: reward = (mybank * 2 - opponents_bank1 - opponents_bank2) in the terminal state, and 1/10 of the sum of reward and card_score is given to the non-terminal states

(3) Parameters T = 0.3, $\gamma = 0.7$, $\alpha = 0.1$

(4) Trained 200,000 times with two random agents, which takes approximately 218 seconds.

2.1 Test cases

To test the performance of the agent, I use it to play against four different pairs of opponents:

(1) Value agent and valueplus agent

(2) Two random agents

(3) Value agent and random agent

(4) Valueplus agent and random agent

For each opponent pair, I tested three times with seeds 0, 1, 2, and each time played 10,000 of games to get the average scores of the three agents.

The testing environment is MacBook Pro 2019, with a processor of 2.6 GHz 6-Core Intel Core i7 processor.

2.2 Test result

As shown in Table 5, we can separate the result into two parts.

1. Efficiency

   For each 10,000-game test, the running time of my agent on the test machine(MacBook Pro 2019) is approximately 10 seconds. The running speed is fast because when testing, the reinforcement learning agent only needs to choose the action with the maximum Q-value in the Q-table, which takes a short time.

2. Competitiveness

   In each test, we can see that my reinforcement learning agent can beat all the opponents with a great advantage of more than 2 points. The test with the value agent and the valueplus agent can see the largest difference between the average scores(more than 3 points), while competing with the random agent, the difference is about 2 points.

Opponents	Seed	Number of games	Average scores	Running time (seconds)
value&valueplus	0	10,000	4.63/1.13/0.93	10
value&valueplus	1	10,000	4.63/1.11/0.96	10
value&valueplus	2	10,000	4.65/1.13/0.93	10
random&random	0	10,000	3.70/1.64/1.63	10
random&random	1	10,000	3.65/1.69/1.62	10
random&random	2	10,000	3.72/1.62/1.63	10
value&random	0	10,000	3.64/1.52/1.79	10
value&random	1	10,000	3.62/1.56/1.78	10
value&random	2	10,000	3.65/1.54/1.76	10
valueplus&random	0	10,000	3.71/1.26/2.00	10
valueplus&random	1	10,000	3.73/1.24/1.98	10
valueplus&random	2	10,000	3.70/1.30/1.96	10

Table 5: My reinforcement learning agent's performance. Test environment: MacBook Pro 2019

Opponent	Number of Games	Random seed	Total running time(in seconds)	My agent average score	Opponent average score
random_agent.py	1,000	0	8	4.93	1.86
random_agent.py	1,000	1	8	4.98	1.90
random_agent.py	1,000	2	8	4.99	1.80
value_agent.py	1,000	0	8	3.67	2.36
value_agent.py	1,000	1	8	3.78	2.42
value_agent.py	1,000	2	8	3.74	2.40
valueplus_agent.py	1,000	0	8	3.58	2.21
valueplus_agent.py	1,000	1	8	3.65	2.27
valueplus_agent.py	1,000	2	8	3.67	2.25

Table 6: My minimax agent's performance. Test environment: MacBook Pro 2019

3 Comparison with Minimax Algorithm

3.1 Minimax Agent

As shown in Table 6, my minimax agent performs better in the competition against a random agent(wins by more than 3 points difference). When competing with the value agent or the valueplus agent, although my minimax agent can still be the winner, it has a smaller advantage over its opponent(about 1.3 points). Thus, the minimax agent is more competitive when competing with opponents with randomness rather than competing with opponents with fixed strategy patterns.

In terms of running speed, it takes 8 seconds for the minimax agent to run 1,000 games, which is much slower than the reinforcement learning agent, which can run 10,000 games in 10 seconds. This is because the minimax agent needs to search the tree and calculate the utility of leaf nodes with a heuristic evaluation function, which requires a lot of computing time.

3.2 Reinforcement Learning Agent

As shown in Table 5, my reinforcement learning agent plays better against the value and the valueplus agent and has a score difference of more than 3 points. However, even though it can still defeat the random agents, the advantage is less significant(about 2 points). The reinforcement learning agent is more capable of competing with opponents with fixed strategy patterns than opponents with randomness.

In terms of running speed, the reinforcement learning agent shows a great advantage over the minimax agent(10 seconds for 10,000 games vs. 8 seconds for 1,000 games). However, the reinforcement learning agent requires a long training time(108 seconds for 100,000 training games and 218 seconds for 200,000 training games). If the number of training times is limited, the agent will not be competitive enough because it hasn't learned adequate information about the game.

3.3 Analysis

The reason why the minimax agent is more competitive when competing with the random agent than competing with the value and valueplus agent is that the minimax agent always finds the best action under the worst cases. As long as the heuristic evaluation function is well designed, the minimax agent can find a better solution than the random agent in most cases.

The reinforcement learning agent can easily beat agents with fixed strategies because with a lot of training, the reinforcement learning agent can detect and exploit the patterns of its opponents, so the best action to take in states is more likely to lead to a positive result. However, when competing with random agents, it is difficult for the reinforcement learning agent to build up relationships between different states, since the opponents' behavior is always unpredictable, making it hard to find a path towards the best result.

Another advantage of reinforcement learning is that once the model is trained, it takes very little time to make decisions because it only needs to search the Q-table for the best action, which doesn't require a lot of calculations. However, the training process and finding a good training approach (designing the reward function, states, and parameters) is very time-consuming.

4 Conclusion

In this assignment, I developed a reinforcement learning agent that can play well in the Raj game in terms of both efficiency and competitiveness. By training with a wise strategy, it can beat the random agent, value agent, and valueplus agent.

To enhance the competitiveness of my agent, I tried to modify the reward function, the state, the parameters T, γ, α, the opponents for training, and the number of training times. By improving and comparing different results of different methods, I eventually developed an agent that is relatively competitive.

The assignment deepened my understanding of reinforcement learning algorithms, including the basic implementation of reinforcement learning, when to update the Q-table, and how to implement the softmax function. After implementing the basics of the algorithm, I started to improve its performance by designing the state, improving the reward function, adjusting the parameters T, γ, α, and training with different opponents and different times to maximize its performance against different agents. Finally, I compared the advantages and disadvantages of the minimax agent and the reinforcement learning agent and found that they perform differently in different scenarios.