Report

Team members: Yuqing Xiao(1089109) , Hongsang Yoo(1105791)

Presentation:https://www.youtube.com/watch?v=TjkD4U56Bhc&list=UUCD11gvSS7B1JZ2vdKw9Ltw&index=2

Introduction

We have chosen three methods (Deep Q-learning, Minimax, and Monte Carlo Tree Search) to build agents and used one version of our Minimax as the final agent. During implementation, we have tried different ways to improve our results and tested our agents against each other. The baseline we used to test our agents is the naïve player from the teaching team's code. We have read some previous related work and we modified part of their algorithms to implement our reward function. In sections 2 to 4, we introduce our methods and how they were developed. In section 5, critical assessment of our final agent is illustrated. Finally, two team members' self-reflection are provided in section 6.

1. Deep Q-learning

Why we chose this method: Q-learning is a simple but powerful algorithm. However, it will require a considerable size of q-tables if state space and action space are too large. In our case, we have a large number of state and action spaces. Therefore, we tried deep Q-learning because it is more capable of handling large scale problems in that we can use a neural network to approximate the Q-value function. The basic framework used in the code was based on OpenAI Gym. The formula for updating parameters for Q-learning is shown below to introduce network structure in the next section:

$$\theta \leftarrow \theta+\alpha[r+\gamma \max_{a'} Q(s',a';\theta)-Q(s,a;\theta)]\nabla_\theta Q(s,a;\theta)$$

1.1. Basic Structure

1.1.1. Network

The input of the network is state feature and the output is the all possible actions. We used two networks because there could be divergence when calculating the predicted and the target value using the same network. One is a network for prediction and the other is a frozen network. Basically, the two networks use the same function, but parameters from the prediction network are updated to the frozen network, which makes training more stable, via freezing parameters. Therefore, we used a prediction network to predict Q-values for the current state and frozen network to infer Q-values for the next state in every replay.

1.1.2. Store and Replay Experience

We stored the parameters (replay data) of prediction network for every action, state, reward, and next_state and trained according to batch size. Then when making decisions, we retrieve the stored replay data (series of action, state, reward, and next_state) when each round is over. Further, when the entire game is over, we retrieve the stored replay data throughout the game again. By doing so, we train the network for the length of every round as well as the length of the entire game.

1.2. Feature Vectorization

The feature to represent the player state consists of three parts: player state (47), enemy state (47) and the round information (1). For the game state, we combine Pattern Line, Floor Line, and Grid Status to represent the game state in the network. For example, if currently, we have *R: 1, B: 2, W: 3, K: 4, Y: 5*, the game state is represented as follows:

- Pattern line(15) : ['W', 'Y', 'Y', 'B', 'B', 'B', 'K', 'K', 'K', 0, 'K', 'K', 'K', 'K', 'K']
- Grid(25): ['B', 'Y', 'R', 0, 0, 'W', 'B', 0, 'R', 'K', 'K', 'W', 0, 'Y', 0, 0, 0, 0, 0, 'Y', 0, 'R', 0, 0, 0]
- Floor(7): [-1, -1, -1, -1, 0, 0, 0]
- State_feature(47) = Pattern_line(15) + Floor(7) + Grid(25)

During the implementation, we found that the information of factories is not that useful, so it was not used in the representation of the game state. The player state feature is combined with the enemy state and round information.

1.3. Rewards

The basic policy for reward was player_score - enemy_score. More specifically, we used the difference of change of scores between players in the round to train for the very round and the difference of the total scores including bonuses to train the replay for the entire game. Therefore, the more points player wins by, the more rewards it gets and it will get minus reward if lost. This evaluation was more effective compared to giving a fixed reward.

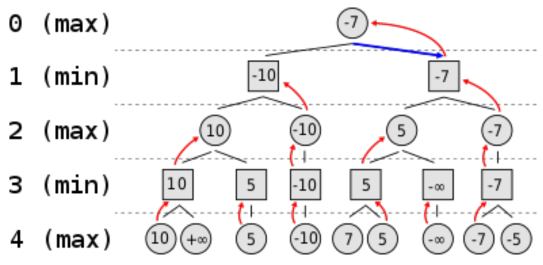
1.4. Results and Thoughts

It beats Naive player with a winning rate of 60%. Not too bad but not too strong. Possibly there are too many action spaces to train. Firstly, we tried to represent state-action pair features in a way of getting only available actions for every state. However, the problem was that these available actions for each state change every time and it was hard to update values to the right one in the network.

2. Minimax

Why we chose this method: Minimax is a decision rule used in game theory for minimizing the loss for the worst case and maximizing the gains. Although the game used for this project, AZUL, can be played with more than two players, there are only two players in the competition. For two-player games, the minimax algorithm is a good tactic because it's a zero-sum game, where two players are working towards opposite goals. In this game, no matter how many scores we get, if the enemy gets more score than mine, we lose. Therefore, we thought minimax is one of the most effective algorithms for this game.

As seen in the figure below, layers have different minimizing or maximizing objectives so as to decide the action that can maximize the gains for the current player.



2.1. Basic Structure

In addition to basic minimax algorithms, since there are many actions to prune, we used alpha-beta pruning to reduce the search space to make the algorithm more effective. In addition, since not all actions are equally worthwhile to explore, we further pruned actions by filtering actions based on the move contributing to the number of tiles needed to fill the pattern line + floor tiles (the lower, the better).

2.2. Evaluation Metrics

At first, we gave fixed points when the player wins or loses (i.e. 10 or -10), but after discussion, we adjusted the fixed points to *(player_score – enemy_score)*, and Minimax outperformed the naïve player. This difference between player and enemy score was much more effective in maximizing and minimizing player/enemy gains. We also added the number of remaining tiles in the pattern line and the grid to gear the agent towards filling the line at one go and putting more tiles in the grid.

Some evaluation metrics that were implemented but were not reflected in the final agent include center strategy and stopping the enemy's goal strategy. The former is to prevent scattered formation in the grid by giving more penalty when completed tiles are not adjacent. The latter is giving points to a move that stops what the enemy wants.

For each line, look at which color enemy will want to take and sum the number of tiles they need. Subtract this number from tiles in the factory (if it is negative then we will get -1 which means it is very good for us, the enemy has no way of completing the line for the round). This will return a vector [1 3 4 9 -1] [B Y R K W], where 3 is number of tiles enemy needs from Y etc. Now our action is also same vector [2 -1 -1 -1 -1]. This means we take 2 blue tiles. If we take the dot product between these two vectors, then if we are taking the resources our enemy needs or if we are not taking the tiles our enemy can't possibly complete, it will result in a high number.

Combined with this stopping opponent goals with the evaluation metric above can be expressed as following formula:

$$Value = a * [(Heuristic\ towards\ own\ goal) - (Heuristic\ towards\ opponent\ goal)] \\ + b * (Heuristic\ stopping\ opponent\ goal)$$

However, contrary to our expectation, adding this metric showed worse performance. One possible explanation is adding this heuristic may impact the primary heuristics [player_score – enemy_score]. And there is an assumption that every enemy's action should be reasonable, which is not always the case. This remains further to be studied.

2.3. Results and Thoughts

Although we used alpha-beta pruning, due to the large number of available actions for each state, initially we couldn't go into more than the depth of 2. However, by pruning further actions that are not worthwhile, we could apply depth 4 to 5. Minimax agents showed a quite effective and powerful movement and achieve a winning rate of about 90 percent against the naive player. Therefore, it was adopted as our final agent. More details of strengths and weaknesses are covered in the 5. Final agent section.

3. Monte Carlo Tree Search (MCTS)

Why we chose this method: The Monte Carlo method is to use randomly sampled values to estimate the true value based on Central Limit Theorem. The larger number of samples we use, the closer the average value will be to the true value. Hence, MCTS has its advantage for game algorithms such as the famous one AlphaZero. MCTS is simple to implement with heuristic algorithm that can operate effectively without any knowledge in the particular domain such as rules and end conditions. It can find the move we need and learn from them by playing random rollouts. Besides, MCTS can be saved in an intermediate state which can be used in the future. MCTS supports asymmetric expansion of the search tree based on the circumstances in which it is operating.

3.1. Search Algorithm

In this game, we used model-free Monte Carlo Tree Searching first to play games randomly and get the reward value after a single round. The algorithm is as follows:
When we meet a new game state: New MCT node, rollout many times and select the best moves.
For one rollout, do the following:

- Selection: select a node unexpanded fully
- Expansion: expand this node by adding all children of it
- Simulation: run rollouts with another agent to give a reward for this node
- Backpropagation: use the reward of the current node to update its ancestors

3.2. UCT and Q-value

UCT method is a very popular method that can be used in MCTS. It balances the exploration-exploitation trade-off. UCT is a combination of MCT and Upper Confidence Bound (UCB) which is applied to trees as the strategy in the selection process to traverse the tree. For two-player adversarial games, UCT converges to a minimax algorithm. The parameters are characterized by the formula that is typically used for this purpose, which is given below.

$$\pi(s) = argmaxQ(s,a) + 2Cp\sqrt{\frac{2lnN(s)}{N(s,a)}}$$

The Q-value of a leaf is its reward, the formula of Q-value of other nodes in our method is as follows.

$$Q = \frac{1}{\sum_{s\in son} N(s)} \sum_{s\in son} N(s) * \mathrm{Q}(s)$$

3.3. Evaluation Metrics

At first, we used the random player to do the simulation. From the result against the naïve player, we can tell with the smaller rollout, we won't time out and the result was good. After that, we used the naive player and minimax player to do the simulation. During these tests, the result became worse because it became easier to timeout. Then we updated the reward function by calculating the expected bonus after a single round instead of the entire game. The result was better. Besides, we noticed that the UCT formula is also very important. The original one that we used is the formula in the lecture, as the equation is the 4.2. By testing against our another player and naive player, we found a more usable equation as follows:

$$\pi(s) = argmaxQ(s,a) + \frac{P(s,s')}{1+N(s)}$$

4. Final Agent

Although we could not test the final agent in the practice tournament, comparing to our minimax player in the initial stage it has the following strengths. 1) Depth increased from 1 to 4 by pruning further implausible actions. 2) More rich evaluation metrics were used other than initial evaluation metrics. Just a few examples of these are the number of tiles in the grid and the number of remaining tiles in the pattern line. By adding these, it worked in a way that fills the line at one go and tries to put as many tiles in the grid as possible. Some weakness may include that the number of actions to be explored were sacrificed to increase depth. In addition, it is not very likely, but it may time-out with bad luck due to the increased depth. Apart from weakness or strength, there were more brilliant ideas we implemented (explained in the evaluation section of 3. Minimax) but could not be included in the final version because it performed worse in contrast to our expectation and did not have enough time to test our changed model in the last phase.

5. SELF-REFLECTION

5.1. HONGSANG

5.1.1. WHAT DID I LEARN ABOUT WORKING IN A TEAM?

I could learn about how to work on group project. Since I have not had experience in doing group projects before, participating in a group work was a big challenge for me. In this special situation when we can't directly meet, I learned how to collaborate via team meeting online and by utilising tools such as trello and slack. And using google doc, we wrote down our ideas and give feedback to each other. Even though not all of this resulted in the actual implementation, discussing ideas itself was something that I could not obtain from lectures and tutorials.

5.1.2. WHAT DID I LEARN ABOUT ARTIFICIAL INTELLIGENCE?

I played Go for a long time when young, and even when I was shocked to see the results of game between Alphago-Lee Sedol, I did not expect myself learning this stuff. I just can't believe I learned core algorithms that's used in the Alphago and understand a bit. First, I learned about blind search and various techniques in reinforcement learning. I did not like planning, especially PDDL, but it was my happiest day of this semester when I received the feedback that one of my approaches was impressive in the last assignment. From lectures and materials, I learned a good deal of reinforcement techniques. However, it was never mine until I went through numerous failures in implementation. During the project, I spent most of my time in code implementation. Although I learned a lot from this process, I had a very good lesson that theoretical part and skills to implement this should be balanced.

5.1.3. WHAT WAS THE BEST ASPECT OF MY PERFORMANCE IN MY TEAM?

While all team members contributed significantly, I could say I spent time a lot. Although this project is mainly about implementing major algorithms, I had to know how the game works. While I failed for countless times, I analysed this project's code and could understood and modify all the important functions in many classes. Time contribution did not always lead to good results but that was something I could do best. In addition, we used google doc to exchange ideas. I tried to give some ideas and give feedbacks to others' opinions and update some important changes in what I have been currently working on.

5.1.4. WHAT IS THE AREA THAT I NEED TO IMPROVE THE MOST?

I want to be a research engineer. While learning throughout the semester and doing projects, I realised a good balance between theory and actual skills for implementation is necessary. While I need to be equipped with more formal knowledges, I should not neglect the effort of trying to put this into implementation. I tried my best to follow the lecture contents but did not spare time for further reading papers and publications. I should review what was covered throughout semester and read relevant interesting papers related to each lecture topic. In addition, I want to improve communication skills and project administrations skills required in group project.

5.2. Yuqing Xiao's self-reflection

5.2.1. What did I learn about working in a team?

During working on this assignment, I have learned the importance of communication, the value of responsibility, and the sense of consciousness. During the previous two assignments of these subjects, I met one of my teammates in the discussion board on the LMS. I noticed that they are asking the same question that I was confused about. And after I know we need to build a team to finish this assignment, I send a message to them over the canvas and we become teammates. I learned how great it is to meet some people share your excitement about new Artificial Intelligence knowledge and confusion about some technical concerns. During we worked together, we used Discord, Slack, Zoom, and Trello to organize our workflow and support our information sharing.

5.2.2. What did I learn about artificial intelligence?

I learned more deeply about Monte Carlo Tree Search MCTS, Minimax Tree Searching, and heuristic function during this assignment. By implementing those algorithms, I have deeper thoughts about how those algorithm is working and why it benefits for the game. For example, during coding for the MCTs, I noticed the difference MCTs with and without Markov Decision Process. I am more clear about how every rollout updates the existing reward value of those nodes in the tree and why different formula will change the result. I understand the relationship between exploration and exploitation profoundly. By implementing the minimax, I noticed how different depths we looked at will influence the performance and how different strategies of pruning will change the data structure which will affect the result in the following.

5.2.3. What was the best aspect of my performance in my team?

The best aspect of me in this team will be my responsive attitude for the whole time. I am willing to spend time organizing our system and taking more challenging parts during the whole time. From the first part of the building team and planning the whole procession of this assignment, to the end part of reviewing each other's report writing and updating our results, I have always been there loyally to our team and build the basis for team to build more reasonable result. When something goes wrong, or someone in the team has to withdraw this subjects, I deal with the situation pretty well by thinking about how to recover our loss and solve the problem instead of dividing the responsibilities.

5.2.4. What is the area that I need to improve the most?

The most area I need to improve will my programming ability. During learning lectures, I can understand the professor's point and take those quizzes quickly. But when I want to implement that algorithm in a particular problem, sometimes it is hard to give a perfect solution. I think what I need to do is implementing small algorithms during regular lectures to improve my programming ability. In this way, better programming ability can also help me implement my algorithm more quickly which will give me more time to improve it.