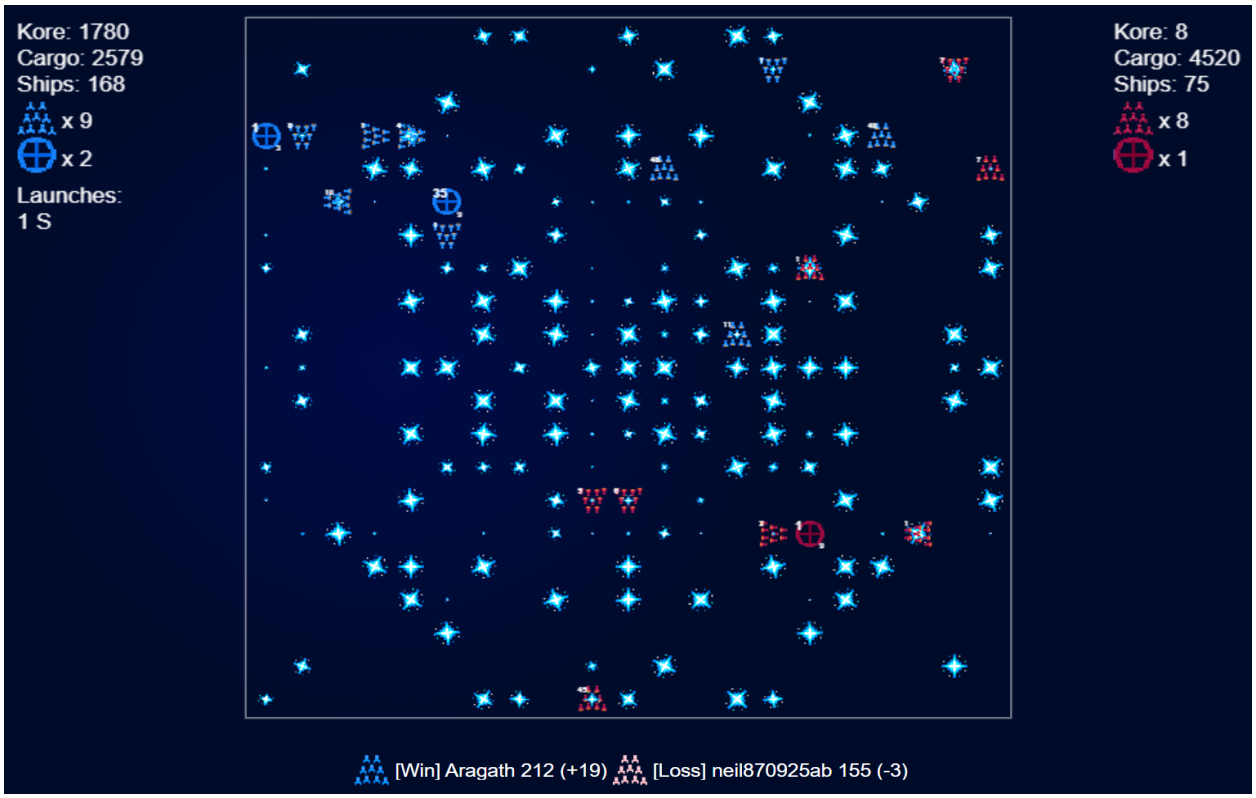
AI Capstone Final Project Report - KORE 2022

By Group 10 “My Machine is Not Learning” ー李泓賢、陳永諭、李品慈

Introduction of the game - KORE 2022



Picture 1. A screenshot of the visualized game state

KORE 2022 is a game competition hosted on Kaggle. It is a turn-based game with 2 players which goes on for 400 turns unless 1 player is eliminated by the other. There are 4 important elements in the game.

1. Kore (blue shiny stars in Picture 1)
   Initially, kore are distributed symmetrically along both vertical and horizontal axes to ensure a fair start. In each turn, every cell having no shipyards or fleets on top regenerates kore by 2% of the existing number (of kore in the cell) until there are 500. The ultimate goal is to collect more kore than the opponent does.

2. Shipyard (the round icon in Picture 1)
   Each player starts with 1 shipyard (with 500 kore and 0 ships) at the fixed position in Picture 1. A Shipyard can increase the number of ships it holds by creating new ones at the cost of 10 kore per each. It can also launch a fleet with the ships it owns.

3. Fleet (the triangle icon in Picture 1)

   The most important mission of a fleet is to mine kore and bring them back to an allied shipyard (only those returned to shipyards are counted in the final score). When launched by a shipyard, the fleet is given a flight plan where it either travels in different directions or converts itself into a new shipyard (costing 50 ships).

   Fleets or shipyards collide with each other if they occupy the same cell. A collision is resolved by:

   Step1:

   All allied fleets are merged into the largest one (for each player).

   Step2:

   After merging, if there still remain fleets from both players, the larger one snatches away the kore carried by the smaller one and both lose as many ships as the smaller one has.

   Step3:

   If there is an allied shipyard, the surviving fleet returns to the yard and the kore it carries is collected.

   Step4:

   If there is a hostile shipyard possessing more ships, it loses as many ships as the fleet has. Otherwise, the fleet loses as many ships as the yard has and the player who owns the fleet takes over the yard.

   A player who has no shipyards and fleets is eliminated immediately and loses the game.

4. Flight Plan

   A flight plan is a string consisting of characters N, E, S, W, C or 0~9. In each turn, every fleet reads its plan and act correspondingly:

N	**E**	**S**	**W**	**C**	**integers**
Go upward	Go rightward	Go downward	Go leftward	Convert itself into a shipyard	Keep going in the current direction

   The flight plan is updated each turn. The table below demonstrates some examples:

Current Flight Plan	Fleet Affect	Next Flight Plan
"N"	Fleet direction set to North	""
"N5E"	Fleet direction set to North	"5E"
"5E"	-	"4E"
"C"	Fleet attempts to convert to shipyard	-
"10E"	-	"9E"

   If the flight plan becomes empty, the fleet keeps wandering in the current direction. Last but not least, the maximum length of the flight plan depends on the number of ships (num_ships):

```
floor(2 * ln(num_ships)) + 1
```

The game is zeros-sum and deterministic with no stochastic events. In every turn, a player passes instructions to each shipyard he or she owns. An instruction tells the shipyard to either build new ships or launch a fleet with a sophisticated flight plan. Moreover, complete information of the board and both players' status is passed to both players (in json format). As a result, the game is played with perfect information.

The above covers the most important rules. Nevertheless, we could not exhaust all details due to the limits on the maximum page. For the complete rules, please refer to https://www.kaggle.com/competitions/kore-2022/overview/kore-rules.

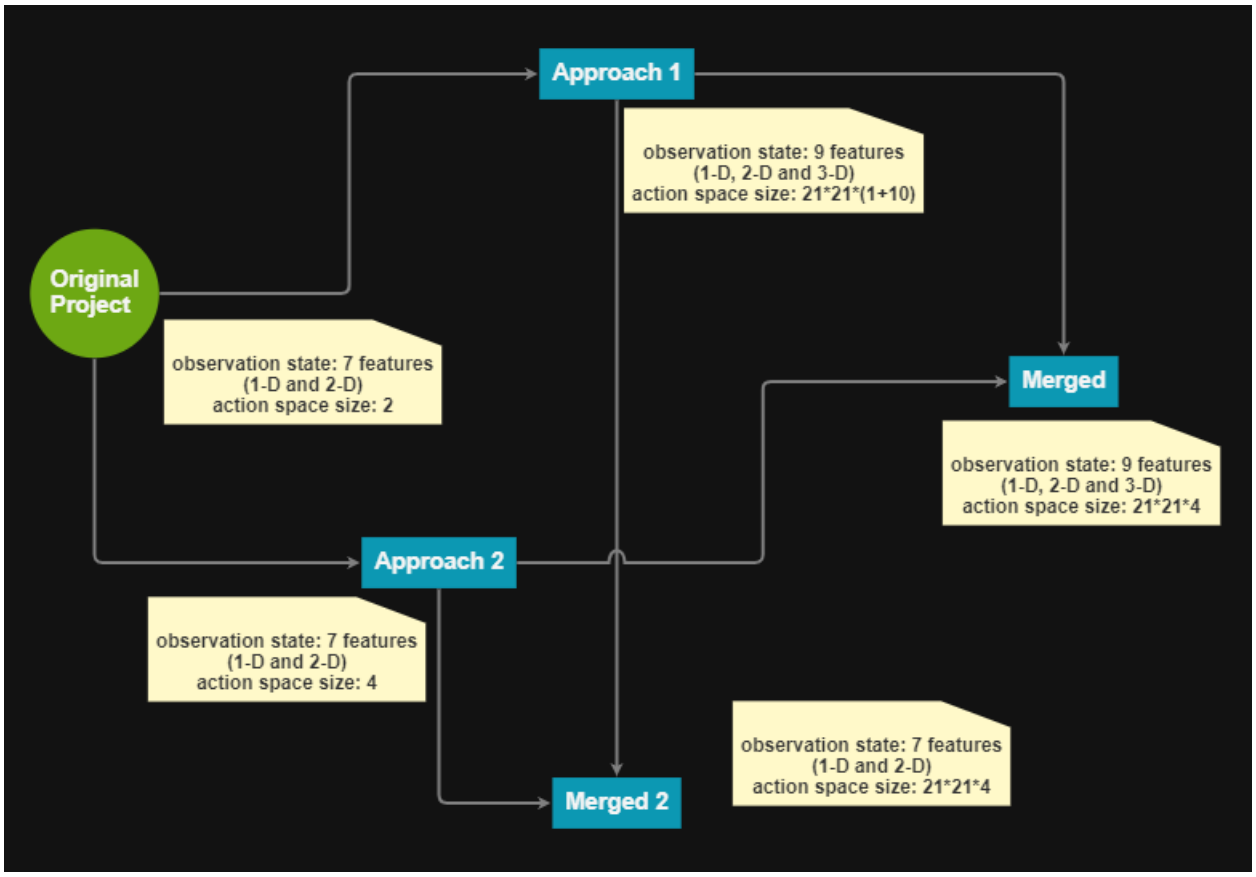
Preface

Our project is a large extension of an existing one on Kaggle, which can be found at Reference. We call it "original project" for the rest of this report. The main purpose of the original project is to demonstrate how to resolve the compatibility issues between the Kaggle environment and RL-related packages. Also, a naive structure for RL is provided. Still, there's still a big room for improvements:

1. Flight plans are not incorporated into state space
2. The small action space only provides limited flexibility to the agent
3. The agent cannot build new shipyards
4. Every shipyard shares the same action space and takes exactly the same action
5. Each fleet launched by shipyards receives a flight plan of only 1 character long, which means it can only keep traveling along the original direction without turning

Below are different RL agent designations attempting to solve problems using different approaches.

Introduction of our project (5 version)



The diagram above summarizes all approaches and the timelines on this project. At first, we tried to expand the size of the action space, and change what RL decides the agent to do. Hence, here came the “Approach 1” and” Approach 2”, with different parameters and settings. However, the performance was not good enough. Dissatisfied with it, we decided to make other changes. These changes are exactly what we call “Merged” and “Merged 2”. Both merged versions inherit the advantages of “Approach 1” and “Approach 2”.

For the purpose of making you understand more clearly, I want to remind you something: The features in “observation state” within “Original Project”, “Approach 2” and “Merged 2” are the same; the features in “observation state” within “Approach 1” and “Merged” are the same. Besides, the action space within “Merged” and “Merged 2” is the same. The brief introduction of each is listed below:

Original Project:

From Kaggle, the non-fixed project which we tried to extend. (Please refer to *Reference*) However, the original opponent.py is replaced by a rule-based agent by us. The disadvantages are mentioned at the *Preface* part. Since the features in “observation state” are the same as “Approach 2”, we will discuss it at *Approach 2 - not so pure RL* part. As for the action space, the author used a very small array (**gym_action[]**) to define it, with the size of 2. That **gym_action[0]** > 0 means the shipyard should launch a fleet, and that **gym_action[0]** < 0 means the shipyard should spawn ships. Then, **abs(gym_action[0])** encodes the number of ships to launch or spawn. Besides,

gym_action[1] represents the direction to move. As you can see, because **gym_action[1]** can only store one character, this fleet can only move along the same direction. Hence, it doesn't have a flight plan (a sequence of few characters).

Approach 1 (on branch approach1):

https://github.com/Aragath/NCTU-AICapstone2022/tree/approach1

As mentioned before, a project which is extended from "Original Project". RL decides actions (only spawn, launch, mine and build) and flight plan. The observation state has 9 features, ranging from 1-D to 3-D. We will give a further explanation about them at *Approach1 - Pure Reinforcement Learning* part. As for the action space, we will discuss it at the same time.

Approach 2 (on branch main):

https://github.com/Aragath/NCTU-AICapstone2022/tree/main

As mentioned before, a project which is extended from "Original Project". RL decides actions (spawn, launch, mine, attack, defend, and build) and the target position which the fleets have to go. Then, the flight plan is calculated by this target position through the rule-based function code. The observation state has 7 features, only presented in 1-D and 2-D. We will discuss it at *Approach 2 - not so pure RL* part. As for the action space, we will discuss it at the same time. Besides, owing to the small size of action space, all the shipyards owned by our agent will do the same action. We consider it a huge problem, and finally improve it in "Merged" and "Merged 2".

Merged (on branch merged_version):

https://github.com/Aragath/NCTU-AICapstone2022/tree/merged_version

As mentioned before, a project which is a mix of approach 1 and 2. RL decides actions (spawn, launch, mine, attack, defend, and build) and the target position which the fleets have to go. Then, the flight plan is calculated by this target position through the rule-based function code. The observation state has the same 9 features as "Approach 1". As for the action space, we will discuss it at *Merged - an evolution* part.

Merged 2 (Temp):

(Just for checking whether the 9-features observation state is good. Hence we won't show you the github link.) A project which is a mix of approach 1 and 2. RL decides actions (spawn, launch, mine, attack, defend, and build) and the target position which the fleets have to go. Then, the flight plan is calculated by this target position through the rule-based function code. The observation state has the same 7 features as "Approach 2". As for the action space, it's the same as the "Merged".

Code or Settings which 5 version have in common

Environment returns

For each turn, the game returns the complete information of the current state in json format.

General Information

Number of turns so far
Kore on the board: position & amount

Player Information

Kore: number each player owns
Shipyards: owner, current position, #ships, turns controlled
Fleets: owner, current position, #ships, flight plan

Opponent.py

The most interesting thing is, we can take any agent from others' submissions shared on Kaggle. At first, we took a quite simple agent to be our opponent. However, it's easy to be defeated by our agent. Therefore, we took a rule-based agent to be our current opponent. This rule-based agent is highly ranked on Kaggle. The link is provided at Reference.

Reward_utils.py

In the field of reinforcement learning, an agent learns to behave in an unknown environment based on the rewards it receives. However, an agent has to lay emphasis on the final rewards, rather than the current rewards. That is, we don't need to possess the most kore at the beginning of the game. Hence, we should formulate a weighting strategy to handle this. The followings are some weight factors:

Weights_kore:

It means the kore count of a player. It will increase linearly from 0 to 1. The value reaches 1 when the game ends.

Weights_assets:

It means the fleets and the shipyards a player has. It will decrease linearly from 1 to 0. The value reaches 0 when the game ends. In other words, a player should spawn more ships or build more shipyards at the beginning of the game.

Weights_max_spawn:

It means the ability of a shipyard to spawn ships. According to the game rules, the longer a player controls a shipyard, the more ships a shipyard can spawn. Hence, a long-held shipyard has the larger value of it.

Weights_kore_in_fleets:

It means the kore in every fleet. This value will no longer be emphasized when

the game ends, since only the kore in the shipyards will be summed up. Besides, its value must be upper-bounded by Weights_kore.

Config.py

As the name of this file, this part contains the settings of the environment, and defines the parameters of our RL machine.

Settings of the environment (which are per-defined by the game rules):

Ship_cost = 10 kore
Shipyard_cost = 50 ships
Episodesteps = 400 turns
Map_size = 21*21

Parameters of RL machine (we only mention a few):

Max_flight_plan = 10 (length) (This only exists in “Approach 1”)
Size_of_observation_space = x
Size_of_action_space = y
Win_reward = 1000
(x and y depends on what version of project we are currently executing; we have mentioned them before)

Approach1 - Pure Reinforcement Learning

Features in observation state:

State received from the game is in json format, but the **openAI-gym** package accepts only **numpy** arrays. Below elaborates how the conversion is conducted. By dimension, the converted state (9 features in total, with a dot in front of each) can be sorted into 3 parts:

1D information

- Number of turns so far
- Amount of Kore I own
- Amount of Kore the opponent owns

2D information (**board size** * **board size**)

Kore-related

- Amount of Kore in the cell

Shipyard-related (if there is a shipyard in the cell)

- Number of ships the yard holds
  The value is positive if I own the yard, negative otherwise.
- Maximum number of ships the yard can spawn (create) per turn

Fleet-related (if there is a fleet in the cell)

- Number of ships the fleet composed of
  The value is positive if I own the fleet, negative otherwise.

- Current direction
  0 for N (upward)
  1 for E (rightward)
  2 for S (downward)
  3 for W (leftward)

3D information (**board size** * **board size** * **MAX_FP_LEN**)

- Flight plan

  A plan consists of tokens (NESWC) and numbers (0~9), which has no upper bound by the rule. To encode the plan, we first expand it to eliminate numbers. For example, “E2W3” -> “E00W000”, where 0 indicates “keep moving in the current direction”. Next, we convert the expanded plan into array with the following 1-to-1 relationship (*Table1*):

0	N	E	S	W	C (create a shipyard)
0	1	2	3	4	5

Table1 lookup table for encoding & decoding

Finally, to fix the length of the flight plan, we trim the array if it is too long (length exceeds **MAX_FP_LEN**) and append paddings (“0”) if it is too short. For instance, a plan “E2W3” will be processed and converted as following:
“E2W3” -> “E00W000” -> “2004000” -> “2004000000”

In the end, all information above is concatenated and flattened into a 1D array so that **Gym** can process it. The total length is **board size** * **board size** * (5 + **MAX_FP_LEN**) + 3.

Action space:

Our **gym_action** is a 3-dimensional vector of size 21*21*(1+10). That is, a 3D (**board size** * **board size** * (1 + **MAX_FP_LEN**)) array (**Actions**). For each cell located at **(x, y)**, the action (**A**) for a shipyard on this cell is determined by **Actions[x][y]**. The first element (**A[0]**) indicates the type of action to take. Here’s the code explanation:

“If positive:

If an ally shipyard stands here, it should launch a fleet and do nothing otherwise.

Number of ships

The number of ships depends on **abs(A[0])**, which $\in$ (0, 1). We linearly expand it into (0, **MAX_ACTION_FLEET_SIZE**) to determine the number of ships of the fleet.

Flight Plan

The plan is determined by the rest of the array (**A[1:]**). All elements here $\in$ (-1, 1). We linearly expand and shift the values into [0, 6) and apply floor function. In this way, values are restricted to 0, 1, 2, 3, 4 or 5. Finally, the array can be decoded according to *Table1*. For instance, the array “2004000000” will be processed and converted as following:

“2004000000” -> “E00W000000” -> “E2W6”

Else if negative:

If an ally shipyard stands here, it should create new ships and do nothing otherwise. The number of ships is determined as we did for fleet launching.

Else:

Do nothing."

Kaggle provides many hints about the advanced action of a shipyard, such as defending and attacking. However, our action space will do nothing but spawn, launch, mine and build. Hence, we will try to fix it in "Approach 2".

Approach2 - not so pure RL

Features in observation state:

By dimension, the converted state (8 features in total, with a dot in front of each) shows up in 1-D and 2-D:

1D information

- Number of turns so far
- Amount of Kore I own
- Amount of Kore the opponent owns

2D information(**board size** * **board size**)

Kore-related

- Amount of Kore in the cell

Shipyard-related

- Whether there is a shipyard in the cell
  - 1 for our shipyard
  - -1 for enemy's shipyard
  - 0 for no shipyard

Fleet-related

- Number of ships the fleet composed of (if there is a fleet in the cell)
  - The value is positive if I own the fleet, negative otherwise.
- Current direction
  - 1, 2, 3, 4 represent four different directions

Action space:

Our **gym_action** is a 1-dimensional vector of size 4. We will interpret the values as follows:

1. **gym_action[0]** represents the "identity" of the launched fleet or for shipyards to build ships
2. **abs(gym_action[1])** encodes the "number of ships" to build/launch.
3. **gym_action[2]** represents the "target to go (x axis)"
4. **gym_action[3]** represents the "target to go (y axis)"

Then, we create a variable called **target_pos** from "target to go".
And by a given identity, they take different actions as follows:

1. Shipyard defender: greedy spawns fleets
2. Attacker:
   a. Tries reaching **target_pos** to attack

    b. If target too far to reach, try attack the weakest shipyard
    c. If target still too far, launch the fleet in random directions
3. Builder:
    a. Tries reaching **target_pos** to build new shipyard
    b. If target too far to reach, do greedy mine(get kore nearby)
    c. If target still too far, launch the fleet in random directions
4. Spawner: spawn fleets according to “number of ships”, we check if it’s affordable and doable.
5. Miner:
    a. Tries reaching **target_pos** to mine
    b. If target too far to reach, try mining the maximum kore
    c. If target too still too far, try mining the nearby kore
    d. And at last if target still too far, launch the fleet in random directions

By doing so, we let RL decide what identity a shipyard or fleet is, and decide what target position a fleet must reach. For the details about the definition of functions, please refer to “*Helper.py*”.

However, due to the small dimension of action space, the shipyards won’t do the different actions at all. Hence, we try to fix it in “Merged”.

Helper.py

This part of the code aims to produce the corresponding flight plan with a given target position.

1. **getNearestLargestKore(shipyardPos: Point, board: Board) -> Point**

    Returns the position of the cell on board that has the largest number of kore

2. **getNearbyLargestKore(shipyardPos: Point, board: Board) -> Point**

    Returns the position of the cell on board that has the largest number of kore within nearby cells. Where we define "nearby” as cells within the 3 x 3 area outside the shipyard.

3. **getFlightPlan(shipyardPos: Point, targetPos: Point, num_ships: int, board: Board) -> Point**

    This function is specifically for fleets that are assigned as “miners", for a given target position, it returns the shortest flight plan that reaches the target then back to the original shipyard position to unload the cargo.

4. **max_flight_plan_len(num_ships)**

    Because the max length of the flight plan of fleets is limited by number of ships, this function is to calculate the maximum length of a given number of ships

5. **simplify(plan: str) -> str**

This function turns an input flight plan string to a simpler form so that it won't surpass the limit of length of it. For example, it turns "NNNEEWSSSS" to "N2E1WS3".

6. **reversed_str(original: str)**

   This function reverses the input flight plan string so that the ship we launch can return to its original shipyard starting point. For example, it turns "N2E1WS3" to "N3EW1S2". And by concatenating the two strings above, the fleet could reach its target then back to the start.

7. **getWeakestShipyard(shipyardPos: Point, board: Board) -> Point**

   Returns the position of the opponent shipyard on board that has the least number of fleets.

8. **getAttackFlightPlan(shipyardPos: Point, targetPos: Point, num_ships: int, board: Board) -> str**

   This function is specifically for fleets that are assigned as "attackers", for a given target position, it returns the shortest flight plan that reaches the target.

9. **getBuildFlightPlan(shipyardPos: Point, targetPos: Point, num_ships: int, board: Board) -> str**

   This function is specifically for fleets that are assigned as "builders", for a given target position, it returns the shortest flight plan that reaches the target and adds the action "C" at the back of the flight plan string.

Merged - an evolution

To fix the problem that appears in "Approach 1" and "Approach 2", we figure out a version that not only combines the advantages from each other, but also deals with the issue of each other.

Features in observation state:

The same as "Approach 1".

Action space:

Our **gym_action** is a 3-dimensional vector of size 21*21*4. That is, a 3D (**board size** * **board size** * four features in "Approach 2") array (**Actions**). For each cell located at **(x, y)**, the action (**A**) for a shipyard on this cell is determined by **Actions[x][y]**. By doing so, just like "Approach 1", the different shipyards can do different types of action. Besides, **A[0]**, **A[1]**, **A[2]** and **A[3]** represent the same thing that we have just mentioned in "Approach 2", respectively. By doing so, just like "Approach 2", a shipyard can do advanced actions such as attacking or defending.

Merged2 - just for checking

The purpose of this version is to check whether our 9-features observation state is helpful to the project.

Features in observation state:

The same as “Approach 2” (7 features, in 1-D and 2-D).

Action space:

The same as “Merged”.

Utils and Tools

1. Gym

   Gym is an open source Python library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments.

2. Stable - Baselines3

   Stable-Baselines3 is a set of reliable implementations of reinforcement learning algorithms in PyTorch.The algorithms follow a consistent interface and are accompanied by extensive documentation, making it simple to train and compare different RL algorithms.

Questions & Improvements

As we know, the most challenging part for training a RL agent is to wrap our custom environment to the one that RL model uses. When designing Approach 1 and 2, we encountered three major problems in total, which are listed separately below:

Approach 1:

Given the environment (state space), action space and reward function, a RL agent can learn complicated patterns by itself without further help. Therefore, this approach originally incorporated every piece of information into the state space (complete flight plans) and provided the agent with unlimited flexibility (except the length) to construct flight plans. However, the results turned out to be extremely terrible (as shown in the *Score Table* below).

We suspected that this was due to the clumsy state and action space. Consequently, we created a version called “Improved Approach 1” to simplify the action space, especially the dimension of it. However, as a result, the score was only enhanced by about 50 points, but it’s still a negative value. Hence, that’s why we don’t count this version in our so-called 5 versions. The details about this version will be shown in the “*Improved Approach 1 - minimizing the dimension*” part.

Approach 2:

There are two problems in this part. The first question we encounter is "how to decode the returned array into KORE game actions". Since it's quite hard for RL to produce sequential output, we took a simpler way: only decode data that's numeric (in this approach, "target position"), and we decide the sequential part ourselves. Moreover, knowing where the agent wants us to go, we also need to decode the reason why the agent wants the fleets over there for, therefore adding the "identity" part to distinguish them.

After decoding the action space, the next challenging part is to produce a flight plan from it. Here we use a rather robust solution, where we only calculate the difference between the target and the starting point and assign the shortest corresponding flight plan. But here's the tricky part of KORE 2022: the length of the flight plan is constrained by the number of fleets, and our flight plan might be automatically truncated. We then added the greedy and random flight plan function mentioned above to make sure our output is reasonable.

Comparison

As we have made a lot of efforts on different approaches and multiple combinations of functions, here we list the comparison table of all the ways we have tried. The score is given by the leaderboard of the "KORE 2022" competition, which can somehow be a good performance measurement. By the way, this score is not always static, because Kaggle uses one's submitted agent to fight against another. The winner gets some additional scores, but the loser gets punished through the drop of his score.

I want to remind you though, the negative value of the score doesn't mean the agent is not trustworthy. It means this agent can't defeat anyone with a positive score. That is, the score-calculating system on Kaggle is not based on a positive value.

Score Table

Train for \ Way	Original Project	Approach 1	Approach 2	Merged	Merged 2 (temp)
5w timestamps	73.5	-113.7	200.7	194.8	188.3
8w timestamps	73.7	-112.3	178.1	177.8	192.9
12w timestamps	OM	-167.3	201.2	202.2	230.3

OM: out of memory

Result and Epilogue

As you can see in the *Score Table*, we have successfully made an obvious improvement from the original project. Let's take a glance at the different training timestamps first: The differences among them are not large enough. Hence, we consider 5w timestamps of training are enough for our RL agent. Next, let's see the distinct version of the project (5 versions in total): Approach 1 performs badly, and we have assumed the reason for it at the *Questions & Improvement* part. Approach 2 seems to have the greatest achievement, and we guess it is caused by the "identity". The "identity" gives different work to the different fleets on a certain

timing. For example, the “attacker” means the agent knows the weaker time (turn) of the enemy’s shipyard or fleets. Hence, it’s time to destroy them! This agent having such a great fighting tactic gives us a great improvement of the score from the original project.

However, Merged, which is a combination of Approach 1 and Approach 2, gives us a nearly same score of Approach 2. To our surprise, the score of Merged isn’t badly influenced by the design of Approach 1. Maybe the power of the “identity” design is big enough to offset the drawback of Approach 1, or maybe the different actions that all the shipyards can do balance the fraud of Approach 2.

By doing this project, we have learned how to make a RL agent wiser. However, the trade-off of it is the large dimension of the action space and the observation state. With the evolution of each version, we finally figure out an appropriate way to handle this trade-off. We hope this experience would give us an eternal elevation in the Machine Learning field.

Improved Approach 1 - minimizing the dimension

https://github.com/Aragath/NCTU-AICapstone2022/tree/approach1_improved

As shown in the *Score Table*, *Approach 1* performed the worst. After reviewing the match history on Kaggle, it seemed like the agent did not really “learn” from the game. We believe that this results from the extremely large number (21*21*15 + 3) of the observation state, and action space (21*21*11). The majority of spaces are occupied by flight plans because we attempted to encode the plan as completely as possible, ending up a size of 21*21*10. We applied the following to fix the it:

Action Space:

Similar to Approach 2, the only difference is that the action space is 21*21*4 so that each shipyard can take different actions. That is, the action space is the same as Merged.

Features in observation state:

Instead of encoding the full plan, we simulate and play the game for 10 more rounds based on the current state. In each round, all fleets move to the next position according to their plan, which creates a new state. Thus, flight plans can in fact be expressed as the sum of these states. Specifically, Approach 1 converted fleets into a 21*21 matrix, with each element storing the number of ships (under the fleet) in the corresponding cell. Now we have 1+10 such matrices thanks to the simulations. We can sum up all of them to incorporate flight plans into the state space. In addition, the matrix generated in the ith round is multiplied by a discount factor 0.9^i in order to distinguish plans in the near future from the ones in the far future. Finally, information of flight plans is included in the state space with only 21*21*1 size instead of 21*21*10.

Reference

Reinforcement Learning baseline in Python (Original Project)
https://www.kaggle.com/code/lesamu/reinforcement-learning-baseline-in-python
Rule-based Opponent
https://www.kaggle.com/code/solverworld/kore-beta-1st-place-solution/notebook

Contributions

李泓賢	1. Merging the code from the others 2. Part of the report 3. Debugging 4. Project video
陳永諭	1. Attempting on the first approach - **obs_as_gym_state** and **gym_to_kore_action** in Environment.py 2. Part of the report
李品慈	1. Project proposal 2. Attempting on the second approach - Helper.py - **gym_to_kore_action** in Environment.py 3. Part of the report