Rule-based, Reinforcement Learning and Imitation Learning Solutions for Lux AI Challenge Season 1

Yiming Dou, Xinhao Zheng, Yutian Liu, Qing Yang
Department of Computer Science
Shanghai Jiao Tong University
{douyiming,void__zxh,stau7001,yangqinghush}@sjtu.edu.cn

1 Introduction

As an AI programming game competition, Lux gives teams the Lux design specification. Upon the designed basic strategy, participants will enhance and submit new strategy which will be used to directly compete with other participants to achieve a better score and rank. Like a typical survival game, Lux AI Challenge Season 1 asks agents ,which are controlled under participants policy that delineated by programming, to own as many CityTiles as possible.

To collect CityTiles, resources need to be gathered to build cities and get through terrible night. It's a turn based competition. At the end of each turn, workers automatically receive resources from adjacent resource tile. Workers on CityTile can not collect resources. Instead, resources adjacent will be used as the fuel of that CityTile. In addition to receive resources, worker can move, pillage, transfer resources and build CityTile from which we get a city. Besides, unit carts can act partial actions of workers, including move and transfer resources. Workers consume all resources in their load to build a CityTile and CityTile builds worker and cart as long as the summation of the number of workers and carts is not larget than the number of owned CityTiles.

During the Day cycle turn, agent collects resources and builds cities under the programmed policy. During the Night cycle turn, resources need to be consumed to produce light to survive. Rules of how many units of fuel need to be burn is shown in Table 1.

Unit	Fuel Burn in City	Fuel Burn Outside City
CityTile	23-5*number of adjacent fridenly CityTiles	N/A
Cart	0	10
Worker	0	4

Table 1: Fuel consumption rule in Night Cycle.

Official website offers a baseline strategy that only perform two missions: collecting resources and building cities. However, this simple policy will make a large amount of city running out of resources and being removed of the game. Thus, more policy are required to solve the resources shortage problem.

From the rule description above, we find several trade-offs in this optimization game. Cities and Units need resources to survive in the night, while resources can be used to build CityTile which consists the cities and is the goal of our game. Workers and carts, building by CityTiles, gather resources required to build a CityTile However, their number is constrained by the number of CityTiles. To sum up, we need to find a policy that ask agent to keep a good balance between building cities and fuel converting. How to train a balanced policy under limited data and computation resources pose a great challenge on this optimization game. In addition, how to predict opponent based on its history action and make corresponding action make this competition even more complicated.

In order to solve the questions above, we take three methods: Rule-based strategy, Reinforcement-learning strategy and Imitation-learning strategy. In short, for each method, we have respectively done the following works:

1. Rule-based: Manually design rules and teach the model to follow the rules (just like human playing)
2. RL: Use Deep Q Network (DQN) architecture introduce the expert experience and design a reward function
3. IL: Add data, replace the origin backbone with UNet and manually add rules to prevent totally wrong decisions.

Our code has been made publicly available: https://github.com/Dou-Yiming/CS410-Projects/tree/main/project2

2 Related Works

Imitation Learning(IL) Imitation learning mimic human or machine behavior to reach a specific goal. Agents learn a mapping between observations and actions, depending on which task is performed. IL works for the scenario that it's easier for an expert to delineate the desired behaviour instead of the specific reward function. It's a Markov Decision Process(MDP). In the MDP environment of IL, we has a S set of states, an A set of actions, a $P(s^{'}|s,a)$ transition model, an unknown $R(s,a)$ reward. Based on the environment, loss function, learning algorithm, and the trajectory history $\tau = (s_0, a_0, s_1, a_1, ...)$ which get be statically stored or dynamically gathered at training time, IL learns the mapping.

In 1989, Dean Pomerleau proposed the ALVINN (Pomerleau, 1989), a 3-layer back-propagation network designed for the task of road following. ALVINN provide a great example for behavior cloning (BC), the simplest form of imitation learning. As a improved version of behavioural cloning, Direct policy learning (DPL) query the experts for real-time training data via interactive

Demonstrator (Luo et al., 2015). Combined with reinforcement learning, inverse reinforcement learning (IRL) learns the reward function of the environment based on the expert's demonstrations and then find the optimal policy. Model-given approach and model-free approach are the main approaches of IRL. In this game, since the replay data from the top teams are free to access, regarding them as experts; and imitating their strategy is definitely a feasible choice.

Reinforcement Learning(RL) Unlike IL based on the description of desired behavior, reinforcement learning lies on rewarding desired behaviors and/or punishing undesired ones. Introduced in 1960s (Waltz & Fu, 1965), the term 'reinforcement' and 'reinforcement learning' become widely used in engineering and scientific field. Soft Actor-Critic (SAC) (Haarnoja et al., 2018) proposes an off-policy actor-critic deep RL algorithm which can work under even continuous setting. For discrete scenario, off-policy method DQN based on Q network offers a slow convergence but high efficiency algorithm. In addition, (Schulman et al., 2017) proposes proximal policy optimization (PPO) the switches between sampling data through interaction with the environment and optimizing a "surrogate" objective function using stochastic gradient ascent.

3 Method

In this report, we propose and implement three methods, which are Rule-based strategy, Reinforcement Learning strategy and Imitation Learning strategy, and the detailed information is described in this section.

3.1 Rule-based Strategy

3.1.1 Baseline Strategy and Improvement

In our rule-based strategy, we use the implementation in https://www.kaggle.com/huikang/lux-ai-working-title-bot as a framework and optimize on top of it. The main idea of the baseline implementation is very simple: each unit is assigned a task and then the unit makes an action based on its task. In the initial implementation, the only two tasks that may be assigned are gathering resources and building cities. This makes it easy for cities to disappear on a large scale later in the game. In addition, due to its simple pathfinding strategy, there is room to improve the resource collection speed of workers.

Based on the baseline strategy, we add several improvement to make agent more tolerant to the Night Cycle:

- Resource allocation optimization. To keep city alive during terrible night, the Units choose the largest city which will go through resource storage and be removed during the night if no other resource supplement.
- City location picking optimization. If condition permits, Units should choose the location with more adjacent cities and more abundant resources to build the CityTiles. Thus, the new creating CityTile has larger possibility to be part of the city or consist other city with adjacent CityTIles. Besides, abound resources can be reached to go through the night.
- Road chosen optimization. Units will choose road with more resources. Besides, according to the mission we should complete, units will circumvent certain space. For instance, units trying to keep city alive should bypass the non-target cities.

3.1.2 Resource Allocation Optimization

We have observed that the baseline agent lacks the concept of sustainability and that there is often a massive disappearance of the city. Therefore, we added a strategy to maintain the city, the details of which are given below.

- Task Assignment. When the game state is in daytime and the next night is more than 2 turns away, the three tasks are prioritized in descending order of collecting resources, building the city and maintaining the city. In fact, a worker is always assigned the task of collecting resources when its backpack is not full, while a worker with no free space is assigned the task of building a city. And And when there are less than two turns left before it comes to night, the priority of the task will change because there is a high risk that the house built at this time will run out of fuel in the following night. At this point, a city worker near a city that lacks the fuel to survive the upcoming night cycle will prioritize sending resources to that city rather than building another city. It is worth noting that before each task is assigned and during its execution, we repeatedly check the reasonableness of the current task, and some tasks that contain unreasonable actions, such as ordering the worker to move longer distances during the night, are discarded.
- Selecting a city for maintenance. We select the cities to maintain according to the following rules:
  - For a worker, only cities within its moving range can be selected as targets.
  - If a city has enough fuel to get through the current night cycle, it will not be selected as a target city.
  - The algorithm will rank all eligible cities based on their size and fuel quantity, and the city size and fuel quantity will be given two different weights.

3.1.3 City Location Picking Optimization

Connecting CityTiles to form a larger city is a useful strategy in most cases. However, in some cases it can backfire, such as when one of the two connected cities is too large but resource-poor, causing them to disappear together in the next round of nights, or when the buildable locations to connect the two cities are too far apart, reducing the efficiency of resource collection. Therefore, in our algorithm, the optimized location is chosen to build a city when and only when the available resources are less than a certain number or when there is a neighboring city in the four tiles around the nearest buildable tile.

3.1.4 Pathfinding Optimization

To improve the efficiency of resource collection, we use the A^* algorithm with weights to bias toward/away from certain things to optimize the worker's path selection. Specifically, the heuristic function of the algorithm is designed to prefer the paths adjacent to resource tiles:

$$h(n) = manhattan\_dist(current\_pos, target\_pos) - \epsilon \# adjacent\ resource\ tile$$

The heuristic function clearly satisfies acceptability when the value of ϵ is greater than 0.

3.2 RL-based Strategy

3.2.1 Basic Idea

In Lux AI, two players issue commands respectively for their teams which is composed of workers, carts, and city tiles on the game grid in different size from 12x12 to 32x32. Both players try to take control of and mine the accessible resources in order to build more cities which can provide the research points, new workers, new carts and the final victory in return. There may be some implicit policy to play the game but it causes a large workload to conclude and set the policy artificially. Thus, we use the RL-based strategy based on the deep Q-learning algorithm.

In this strategy, an agent interacts with an environment repeatedly by taking actions at each turn and receiving rewards and new observations, and in so doing tries to learn the best sequence of actions given observations to maximize the expected sum of rewards. After many games of experience, the agent will hopefully learn which actions are good and lead to a positive reward, and which ones are bad and lead to a negative one. A deep convolutional neural network parameterized an action policy (in other words, the strategy), which was trained using backpropagation to maximize the probability (specifically, the log-likelihood) of winning the game over losing it (since the game result was the only source of non-zero reward), and in the process of playing against itself many many times, learned to take good actions over bad ones at each step.

Now it's how to design rewards and new observations and arrange the items consisting of the agent ,like workers, carts and city tiles, that remains a big challenage due to the large state space and the time and space limtiation of our RL model.We implement our RL model based on the code in https://www.kaggle.com/voidzxh/luxai-simple-reinforcement-learning

3.2.2 A simple RL model of DQN

Firstly, we use this observation model for all the model below to store the raw data from the environment in the game.

The current observation during the game is defined in Tab. 2..

width,height	the size of the real grid
padding_width,padding_height	the padding size for the extenstion from the real size to 32x32
resources maps	the map marked for the three resources(wood,coal,uranium)
worker tile state list for both side	the state about each worker of both players (current position, cooldown time and capacity)
city tile state list for both side	the state about each city tile state of both players (current position, cooldown time and remaining maintenance time)
global information	the time step and the time point during a diurnal cycle

Table 2: Observation setting of RL model

Based on the current observation, we simply concentrate all the map information as the grid input and all the global input as the global feature and let both of them be the input for the Deep Q network.

Then, We use the simple idea that the agent treat all the workers and city tiles separately. We define the actions of workersin this six ways below:

- North move(N): the worker take a step north.
- South move(S): the worker take a step south.
- West move(W): the worker take a step west.
- Earth move(E): the worker take a step east.
- Stay(S): the worker stay here without doing anything.
- Build city(B): the worker stay here and build a city tile.

We put the input mentioned above and throw it into the DQN to get the actions for each worker. The structure of the Deep Q network is shown below.

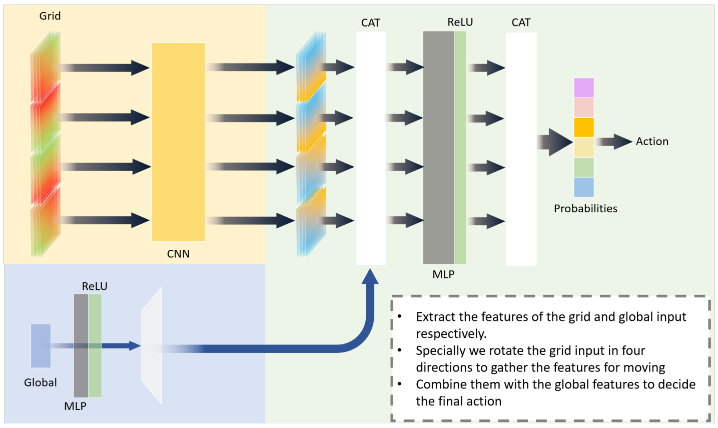

Grid
CNN
CAT
ReLU
CAT
MLP
Probabilities
Action
ReLU
Global
MLP
Extract the features of the grid and global input respectively.
Specially we rotate the grid input in four directions to gather the features for moving
Combine them with the global features to decide the final action

Figure 1: The structure of Deep Q Network

We respectively extract features of the grid input in four direction to move and the global input. Then we combine the new global and grid features to deliver the final decision of the actions.

Besides, to let our model more creative, we design this algorithm below for workers' and city tiles' actions and we calculate the reaward function for the total actions and after one episode, we calculate the loss based on the reward in the time sequence to update our DQN. The original reward and loss function is shown below:

$$loss = DQN_{loss} \quad , \quad Reward = \sum_{w \in Workers} fuel_w + \sum_{c \in Citytiles} fuel_c + V \times \#Citytiles$$

where $V = 100$ and $fuel_w$ and $fuel_c$ respectively represent the fuel the worker and city tile possess.

For each worker, we use the algorithm below based on the Deep Q Network above to decide its actions.

Algorithm 1: Actions selection algorithm

```
$rnd = random()$
  if $rnd \geq \epsilon$ then
  |  Use the deep Q network to decide the actions
  else
  |  Choose the actions randomly
```

Practically, we start with $\epsilon_1 = 1$ and it will decrease to 0.1 gradually.

For each city tiles which can act, we simply judge whether the number of workers is less than the number of city tiles. If not, it do the research else it creates a worker.

3.2.3 Reward function optimisation

In the model above, after some epsisodes of training, we find that the agent always order the workers to stay at the beginning place and it only can survive until the game is over but doesn't have any chance to win.

Base on this, we define the reward function below to let the worker and city tiles learned to simply harvest wood and build cities near the forests and implicitly learned to deny the opponent access to resources according to the phenomenon that workers always build the cities to surround the resources and to survive until the game is over.

$$\begin{aligned}
Reward_{workers} &= \sum_{w \in Workers_{far}} min\{\sqrt{\frac{fuel_w}{\text{\#fuel need in night}}}, 1\} \times V_w \times fuel_w \\
&+ \sum_{w \in Workers_{close}} min\{\sqrt{\frac{fuel_w}{\text{\#Distance to city tiles}}}, 1\} \times V_w \times fuel_w \\
Reward_{citytiles} &= \sum_{cb \in \text{Citytiles blocks}} \min\{\sqrt{\frac{fuel_c}{\text{\#fuel need in night}}}, 1\} \times V_1 \times \text{\#Size of cb} \\
&+ \sum_{cb \in \text{Citytiles blocks}} V_2 \times \text{\#Surrounded Resources} \\
Reward_{proportion} &= sigmoid(Reward_{workers} + Reward_{citytiles}) \\
Reward &= Reward_{proportion} \times \sqrt[3]{Reward_{workers} + Reward_{citytiles}}
\end{aligned}$$

where we use $V_1 = V_2 = 1000, V_w = 500$. $Workers_{close}$ represent the workers which can go to city before the night and $Workers_{far}$ represent the worker which can't do it. #Surrounded Resources represent the surrounded resources of the city blocks which we define as the resource among the bounding box of the city blocks.

3.2.4 Expert experience-assisted training

In practice, in spite of the optimisation about the reward function which let the worker and city tiles learned to simply harvest wood and build cities near the forests and the importance of denying the opponent access to resources by circling the resources, it is still difficult to get the RL model by training the model in the way to randomly sample the actions of all the workers and city tiles in the batch.

Thus, in order to impove the performance of our RL model, we introduce the IL model as a kind of expert experience to guide the training process.

Specificaly, we introduce two parameter to use the expert experience when selecting workers' actions. We have this algorithm below to the workers' actions while training.

Algorithm 2: Actions selection algorithm including the expert experience

```
$rnd = random()$
  if $rnd \geq \epsilon_1$ then
  |  Use the deep Q network to decide the actions
  else if $rnd \geq \epsilon_2$ then
  |  Use the IL model to decide the actions
  else
  |  Choose the actions randomly
```

In practice, we start with $\epsilon_1 = 1$ and let $\epsilon_2 = 0.1$. During the training, ϵ_1 will decrease to 0.3 gradually which will let our RL trained under the guidance of expert experience and remain the opportunity to give creative actions because of the choice for random options.

Besides, we update the loss function as follows to train our model well:

$$loss = DQN_{loss} + \lambda \times Expert_{loss}$$

where we use $\lambda = 0.2$ in practice. The $Expert_{loss}$ consist of the error between the expert experience and the current training model for the same workers.

3.2.5 High level RL implementation

Although we design the reward function to guide our agent to learn how to solve the problem we mention above, the state space and the time for training is quite large. Thus, to improve this, we introduce the Toad Brigade's Approach which implement IMPALA algorithm to achieve the DQN network.

This structure contains several groups of actors that continuously generate trajectory experiences, after which one or more learners use these experiences sent from the actor for off-policy learning.

Due to the time limitation, we doesn't implement this algorithm but it doesn't matter it become a possible solution for the faster and better training process of the DQN network.

3.3 IL-based Strategy

3.3.1 Basic Idea

Since the replay data from the top teams are free to access, regarding them as experts and imitating their strategy is definitely a feasible choice. The basic idea is very simple: Given the replay, which shows the amount of resources, cities and agents, etc., we train a network that predicts the action of the agent for each place. The final goal is successfully predicting what the experts will do under the given situation.

Specifically, we regard the information of the whole map as a representing one feature. The features are shown in Tab. 3.

whether the unit is making the decision	unit cargo level
existence of self unit	self unit cargo level
existence of opponent unit	opponent unit cooldown level
opponent unit cargo level	existence of self city
self city tile night survival duration	existence of opponent city tile
opponent city tile night survival duration	resource wood level
resource coal level	resource uranium level
self research point	opponent research point
day night cycle number	current turn number
whether it's out of bounds in the map	map size

Table 3: Input features of imitation learning

3.3.2 Data Collection

Next, we take the map as the input and extract the feature by a NN, and the output is the prediction of actions.

In order to obtain the data to train the model, the replays of the top agents are downloaded from the website. We follow this notebook, which uses Kaggle's GetEpisodeReplay API to download the episodes. To make sure that the data we utilize is the true expert data, we do the following choosing operations:

1. Only the episodes with LB score that is higher than 1800 are remained.
2. Only the episodes of 1st and 2rd teams are remained.

After these operations, 2080 episodes are downloaded, which are utilized as meta-data.

Now that we have collected the meta-data, the dataset is created in the following way:

1. Split train:val=9:1
2. For each episode, we only look at the winner
3. For each step (turn), and for each agent, we construct a matrix named as obs, and each channel is assigned by the value defined above. Moreover, the label is set as the action of the agent, which should be one of [center, north, south, west, east, build-city]

3.3.3 Model Design

For imitation learning, we start with this notebook, which is regarded as a baseline model in this project.

The model structure of the baseline model is very simple, it takes the feature map with 20 channels as the input, and the backbone of the model includes 12 CNN layers with batch normalization and identity shortcuts.

After the feature extarction process done by the CNN layers, the feature vertor of the corresponding position is taken out and the final layer is a fully-connected layer that predicts the action of the agent at the corresponding location based on the vector.

Although the performance of it is good, many aspects of this model can be optimized:

1. Backbone Architecture

   Each position of the final feature map of the CNN layer can be regarded as the information needed for the prediction of action class, thus the whole process is very similar to that of the image segmentation, which predicts the class of each pixel.Therefore, we consider replacing the backbone of our model with the UNet structure.

   UNet, which is firstly proposed in 2015 for image segmentation, is build upon is so-called "fully connected convolutional network". The idea is supplementing the basic contracting network by adding successive layers, in which pooling layers are replaced by up-sampling layers.

   During the contraction, the spatial information is reduced while feature information is increased. The expansive pathway combines the feature and spatial information through a sequence of up-convolutions and concatenations with high-resolution features from the contracting path. A large number of up-sampling parts allow it to propagate context information to higher resolution feature layers.

   The structure of the network is shown in Fig. 2. The network consists of a contracting path (left side) and an expansive path (right side), which gives it the U-shaped architecture. The contracting path is a typical convolutional network consisting of convolution layers that extract features, max-pooling layers that down-sample the feature and ReLU activation layers. The expansive path includes up-sampling layers followed by convolution layer (up-convolution), which increases the resolution of the input. The final layer is a simple 1x1 convolutional layer. In all, the whole network includes 23 convolutional layers.

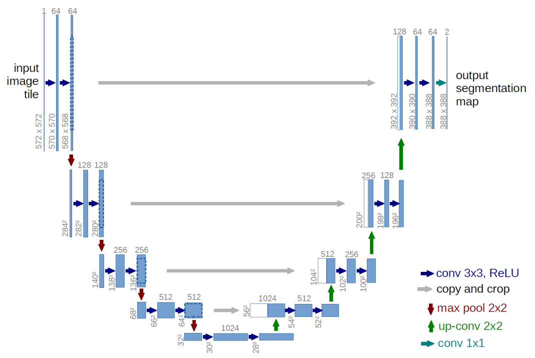


Figure 2: UNet Architecture

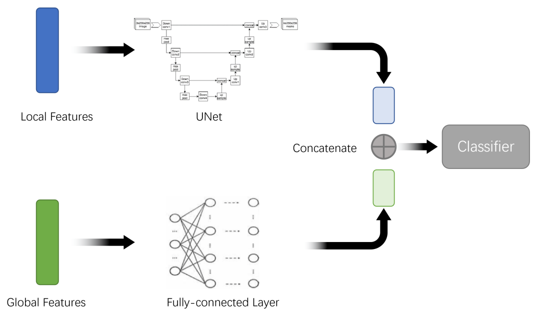


Figure 3: The global features are concatenated with the local features before being sent into the final classifier

2. Add Global Features

   Most of the features in the input map are local features that pay attention to the information of specific positions on the map. However, there many global features which take the whole game into consideration both spatially and temporally, such as map size, day / night cycle, total number of agents, etc.

   These features can definitely help the model to adopt different strategies under different situations. For example, if the current turn is close to the next night cycle, then intuitively the model should make each agent go back to the nearest city as soon as possible, preventing from the agents' running out of fuel outside the city during the night.

   In order to take the global features into consideration, we consider concatenating the global features with the local features extracted by the CNN before sending the features into the final classifier. Specifically, we respectively use the CNN and the fully-connected layer to encode the local and global feature into the same space and then concatenate them with each other. The progress is shown in Fig. 3.

3. Manually Add Rules

   Although the training of this model is under the supervision of expert data, the model may sometimes make very awful decisions (e.g. leave the city at night then disappear due to fuel exhaustion). Therefore, we decide to manually add several rules that the model should follow so as to prevent the totally wrong actions from happening. Here are some examples of the added rules:

   (a) At each turn, the Manhattan distance between each agent and its nearest citytile is computed, which is defined as D_M. Then, the time remaining before entering the night (defined as T) is compared with D_M. If D_M is equal to T, then the agent is forced to go back to the nearest city no matter what the prediction is, since this prevents the agent from disappearing at night.

   (b) During night, every agent in the city are forced to stay still, preventing them from leaving the city and disappear.

   (c) At each turn, the destination of every agent is saved. If the destination of the current agent is the same as that of another agent, which means the two agents may collide with each other and the actions will become invalid, then the current agent will gradually select the sub-optimal solution until the two agents will not collide with each other.

4. Map-Level Inference

   When performing the inference, the baseline model mask all of the output of the UNet except for the position of the agent that is doing decision. This method lead to two major problems. Firstly, at each turn, the inference process of the model should be done for each agent, thus leading to much higher time cost. Secondly, during the training process, due to the mask, only a very small portion of the parameter will get updated, resulting in much longer training time.

   In order to address the two problems, the map-level instead of agent-level inference is adopted. Map-level inference enables the model to predict the action for all of the agents in the inference process, meaning that only one inference process is performed during each turn. This saves much time, both in training and inference process.

   Specifically, the implementation of map-level inference is very similar to that of UNet. We treat the prediction as image segmentation, which means assigning a class label for each pixel. Therefore, the only modification is that the final fully-connected layer is no longer needed, and the output of the final layer of UNet is treated as the predictions.

5. Map Rotation

   This is a very simple optimization method: during training process, the map is randomly rotated for 90, 180 and 270 degrees, and the labels are correspondingly changed to fit the rotation. This method can be considered as a data augmentation method, which enlarges the train-set for about four times.

4 Major Results

In this section, we show the major results of this project, including the LB score and comparison between different strategies.

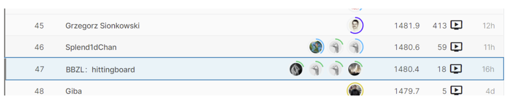
4.1 Leader Board Score

Fig. 4 and Fig. 5 respectively shows the highest and the final LB score of our models.

Note that we have once achieved the LB score that is higher than 1480, but it is very strange that the score decreases rapidly in the final day of the contest, resulting in the final score of about 1410.

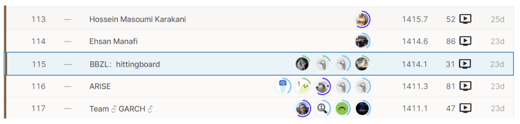
Moreover, it is even more strange that the model with highest score at last is the baseline model that is submitted more than a month ago, and its score had stayed around 1300 during the whole month, while the score of the model finetuned on much larger dataset with a much higher accuracy decreased from 1480 to less than 1400 just in one day!

Therefore, we think that the matching mechanism is somehow unreasonable and it does not truly show the capability of our model.



45	Grzegorz Sionkowski	1481.9	413	12h
46	Splend1dChan	1480.6	59	11h
47	BBZL: hittingboard	1480.4	18	16h
48	Giba	1479.7	5	4d

Figure 4: The highest LB score we have achieved



113	—	Hossein Masoumi Karakani	1415.7	52	25d
114	—	Ehsan Manafi	1414.6	86	23d
115	—	BBZL: hittingboard	1414.1	31	23d
116	—	ARISE	1411.3	81	23d
117	—	Team ♂ GARCH ♂	1411.1	47	23d

Figure 5: The final LB score

4.2 Comparison between Strategies

Tab. 4 shows the comparison of the final results of our different strategies. We may come to the following conclusions:

1. Imitation learning strategy outperforms the others, since it is based on supervised learning, which is much easier to train.
2. Rule-based strategy is very promising, since it comes close to the performance of imitation learning and we are able to further extend its potential by adding more detailed rules.
3. Due to limited training time and high convergence difficulty, the RL model only shows limited performance, but it is still better than the baseline model and we are confident that its performance can be largely optimized if given enough training time.

Rule-based	RL	IL
1197.4	345.9	1414.1

Table 4: IL > Rule-based » RL

References

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In International conference on machine learning, pp. 1861–1870. PMLR, 2018.

Jiyun Luo, Xuchu Dong, and Hui Yang. Session search by direct policy learning. In Proceedings of the 2015 International Conference on The Theory of Information Retrieval, pp. 261–270, 2015.

Dean A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In D. Touretzky (ed.), Advances in Neural Information Processing Systems, volume 1. Morgan-Kaufmann, 1989. URL https://proceedings.neurips.cc/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

M Waltz and K Fu. A heuristic approach to reinforcement learning control systems. IEEE Transactions on Automatic Control, 10(4):390–398, 1965.

A Group Member Contribution

In this section, we will show the contribution of each group member, respectively:

- Yiming Dou: Modify the code of imitation learning strategy and write the corresponding part of the final report.
- Xinhao Zheng: Modify the code of reinforcement learning strategy and write the corresponding part of the final report.
- Yutian Liu: Modify the code of rule-based strategy and write the corresponding part of the final report.
- Qing Yang: Search for related work, write the Introduction and Related-work section of the final report