# Conquering the Dark Nights with Machine Learning

A Report for CS410 Artificial Intelligence - Lux AI Challenge

Hang Zeng
nidhogg@sjtu.edu.cn

JingChun Xiao
xjc0365@sjtu.edu.cn

ZiYin Zhang
daenerystargaryen@sjtu.edu.cn

HuaCan Chai
fatcat@sjtu.edu.cn

Abstract—In this report, we give a thorough description of our approaches to challenging the Lux AI competition, a multivariable optimization problem with highly complicated state space. We cover reinforcement learning algorithsm including DDQN, DDPG and PPO, imitation learning algorithms with UNET and Attention mechanism, as well as the more traditional rule-based models. For each model, we briefly introduce its design principles and ideas, followed by implementation details and performance evaluation.

Index Terms—Rule-based Model, Reinforcement Learning, Imitation Learning, Lux AI Challenge

## I. Introduction

The Lux AI Challenge is a turn-based strategic survival game. In every turn, two agents take actions (including building cities, building units and increasing research points) to gather as much resource as possible to survive the dark nights full of terrors. At the end of the 360th turn, the agent with more city tiles wins the game. The detailed specifications of the game can be found on its official website hosted by kaggle[1].

The remainder of this report is organized as follows. In section 2, we give a brief high-level introduction to reinforcement learning and imitation learning, two machine learning paradigms that we exploit to tackle this game. From section 3 through section 5, we present in detail the three methods we adopted: hand-written rules, reinforcement learning, and imitation learning. At the end, we record the division of labor within the authors.

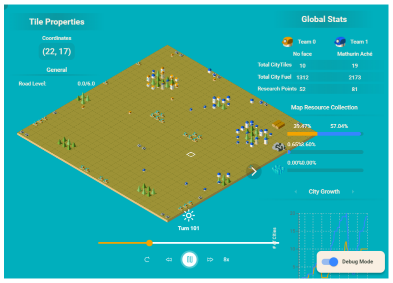


Fig. 1. A screenshot of Lux AI

[1] https://www.kaggle.com/c/lux-ai-2021/overview/lux-ai-specifications

## II. A Brief Introduction to RL and IL

### A. Reinforcement Learning (RL)

Reinforcement Learning is a branch of machine learning that concerns sequential decision making. It is one of the three major paradigms in machine learning alongside supervised learning and unsupervised learning, and puts emphasis on optimizing the actions of an agent to maximize the cumulative reward it receives from its interaction with the environment.

Reinforcement learning is usually based on Markov Decision Process (MDP), which is defined by its state($s_0$), action($a_0$), policy($\pi_0$), reward($s_1$), and return($r(s_0, a_0, s_1)$).

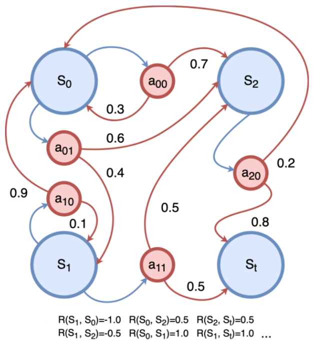

$S_0$
$a_{00}$
0.7
$S_2$
0.3
$a_{01}$
0.6
0.4
$a_{20}$
0.2
0.9
$a_{10}$
0.5
0.1
0.8
$S_1$
$a_{11}$
$S_t$
0.5
$R(S_1, S_0)=-1.0$ $R(S_0, S_2)=0.5$ $R(S_2, S_t)=0.5$
$R(S_1, S_2)=-0.5$ $R(S_0, S_1)=1.0$ $R(S_1, S_t)=1.0$ ...

Fig. 2. An example of MDP

In an MDP, reinforcement learning aims to predict the expected total returns - or value functions - of all possible policies at each state, and the policy that yields the highest value function will be taken by the agent.

### B. Imitation Learning (IL)

Imitation learning provides an alternative approach to learning, aside from the more pervasive trial and error paradigm. The core idea is that instead of providing an agent with the desired objective and leaving it alone to figure out how to achieve it, we directly instruct the agent to emulate specific actions demonstrated by human teachers, based on its perception of the environment. It is much like traditional supervised learning, only that its training examples consist of pairs of features and actions, instead of features and labels. This can help prevent the agent from falling into local optima, as well as making it more human-like. [3]

Both reinforcement learning and imitation learning are widely exploited in robotics, games, and other domains where the dimension of the state space is too large for traditional machine learning algorithms to handle.

## III. Rule-Based Models

### A. Basic Rule-Based Model

The simplest rule-based model (named “simple_rule”) is what we used at the beginning. The significance of this model is more symbolic than practical, as it only gets a score of about 900 on the leaderboard, but it was our first step towards challenging the competition, and acquainted us with the game API.

The rules used in this model are simple. What the workers do is shown in Figure 3.

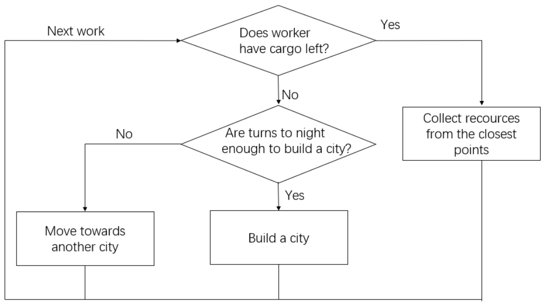

Next work
Does worker have cargo left?
Yes
No
Collect recources from the closest points
No
Are turns to night enough to build a city?
Yes
Move towards another city
Build a city

Fig. 3. Strategy of workers in the simple rule-based model

In “Move towards another city” part, if it is daytime and there are enough turns left from night, the worker will just choose the closest city tile. But if there are less than 10 turns to night, it will try to find a nearby city that are running low on fuel first and save it.

In “Collect resources” part, we assign different weights to different resources according to their importance and the scale of the map. For example, if the width (or height) is 32, then wood is assigned a weight of 15, coal 3 and uranium 1 (the weights are inversely proportional to priority). The rationale is that as the scale of the map increases, the weight of wood is decreased as we will have enough time to research uranium, instead of collecting wood and coal to the end.

As for the city tiles, we simply stipulate that they build workers when feasible (i.e. when the number of workers is less than the number of city tiles), and otherwise research if the research points are less than 200 (which is the doorstep to uranium).

Performance of this model: 946.3 scores.

### B. Advanced Rule-Based Model

The previously introduced rule-based model is just a cornerstone for understanding how the game works. In this subsection, we demonstrate our final rule-based model after seven versions of evolution.

This model is not a pure rule-based model. It only takes over from IL or RL agents (which will be covered in later sections) after 200 turns and when uranium technology has been researched. This is an strategy in response to the observation that at the start of a game, the possibilities of opponent actions are too numerous for rules to cover; but as the turn grows, the game states become far too complicated, rendering IL and RL models inapplicable.

The model is divided into three parts: city actions, worker missions, and worker actions.

The process of generating city actions is shown in Figure 4. In an RL or IL agent, city tiles usually build workers whenever possible. But as the game proceeds into later phases, too many workers may undesirably take up spaces on the map that could have been used to build more city tiles. So we dictates that a city tile only build workers when it is adjacent to uranium or coal, or when the player’s units are too few with respect to the amount of resource left on the map.

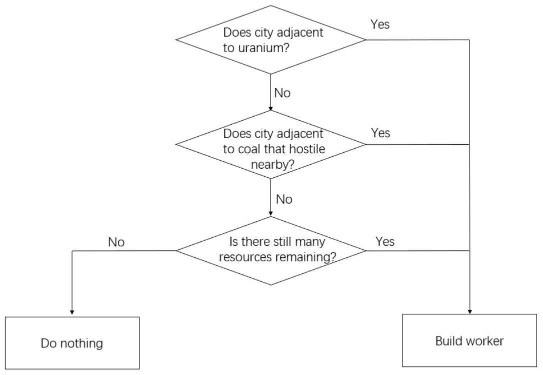

Does city adjacent to uranium?
Yes
No
Does city adjacent to coal that hostile nearby?
Yes
No
Is there still many resources remaining?
No
Yes
Do nothing
Build worker

Fig. 4. Process of generating city actions

The workers’ case is a bit more complicated, as they have more freedom than city tiles. Each worker has a mission that spans across turns. In each turn, every worker’s mission is updated first based on the game state, and then it will get assigned an specific action (move, stay, or build city) based on its mission and its surroundings, as illustrated in Figure 5.

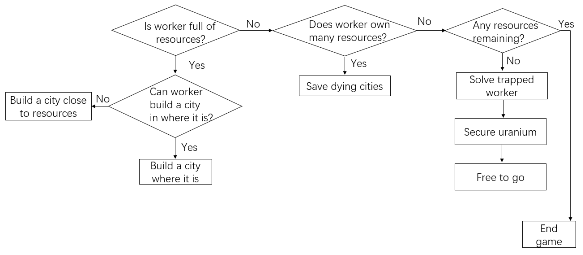


Fig. 5. Process of generating worker actions

The missions of a worker can be classified into the following categories. In each turn, an agent will iteratively consider these types of mission, and choose one that best suits its current state.

1) find a desirable position to build a city tile;
2) save a dying city (one that is running low on fuel);
3) collect resources in order;
4) endgame;

When targeting to build a city tile, the agent will first consider extending existing cities that have abundant fuel to sustain them through the next few nights, or building on positions adjacent to uranium, and then other options less desirable - for example, adjacent to coal or cities not that rich in fuel. However, if the agent is carrying a large amount of uranium, it should not use it to build a city tile as uranium is too valuable too be wasted in building. So, in this case, the agent simply seeks out a city that's running low on fuel, and saves it.

On the other hand, if an agent does not have full cargo, it will look for resources on the map to collect. When collecting, the utmost priority an agent considers is to secure exposed uranium on the map. That is, an agent will always try to occupy an empty position by the side of uranium, if there is any. Collectively, our agents will thereby try to surround all the uranium on the map, cutting it off from the opponent. If no uranium is available, an agent will then consider coal and wood in turn. We also specified rules for endgame cases, when there is no more resource on the map. In this situation, all the workers will go to the cities that do not have enough fuel to sustain themselves to the end of game but can be saved with the resource carried by the workers.

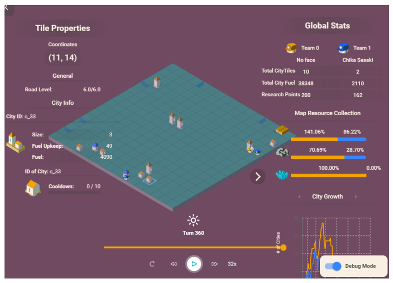
Lastly, we note that all the decisions of cities and workers are made based on highly complicated information provided by the Game class, which is updated at the start of each turn. The class we used is built upon baseline 2[2], but with much more tailored information about the map. The details can be found in our source code.

## C. Performance of the Rule-based Model

Performance of the rule-based model: 1373.5 scores. Figure 6 shows a screenshot of its games (our team is the yellow Team0).

One major shortcoming of this model is that it lacks a systematic organization, and some workers change their missions too frequently. Oscillations in workers' actions may also occur, as a result of the inefficient coordination between workers. These problems will be dealt with by the following machine learning methods.

[2]https://www.kaggle.com/huikang/lux-ai-working-title-botUpgraded-Game-Kit


Tile Properties
Coordinates
(11, 14)
General
Road Level: 6.0/6.0
City Info
City ID: c_33
Size: 3
Fuel Upkeep: 49
Fuel: 4090
ID of City: c_33
Cooldown: 0 / 10

Global Stats
Team 0 Team 1
No face Chika Sasaki
Total CityTiles 10 2
Total City Fuel 38348 2110
Research Points 200 162
Map Resource Collection
141.06% 86.22%
70.69% 28.70%
100.00% 0.00%
City Growth
Turn 360
32x
Debug Mode

Fig. 6. Screenshot of the Rule-based Model

## IV. RL Models

As this game is a multi-agent work, we need to design a reward for each worker, which is hard to implement. Thus, RL Models are quite difficult to train.

### A. DDQN Model

1) A Brief Introduction to DQN and DDQN:

As is mentioned in section 2, in reinforcement learning we choose an action from the action space of the current state that yields the highest return (Q value). But the complexity of calculating the exact Q value soon becomes intractable with the increase of the dimensions of state space and action space. Deep Q network, or DQN, is thus introduced to get an estimation of the Q value. DQN is essentially a variant of the traditional Q-learning with deep neural networks applied in it. [2] Once trained, it basically takes the state information as input, and outputs $n$ Q values to help the agent make a decision, where $n$ is the dimension of action space.

When training a DQN, we sample a sufficient number of state-action sequences to form an experience pool. In each episode, we pull a batch of $(state, action, reward, state')$ tuples from the experience pool and update the Q values of $(state, action)$ pairs therein. After each episode, the policy is updated, and more data can be obtained with the new policy. In the algorithm, two networks are maintained: one is update-Q-network, and the other target-Q-learning. As the name suggests, the parameters of update-Q-network are updated at every step according to the temporal difference estimated from target-Q-network, which stays fixed until the parameters of the update-Q-network are all updated, and then gets its parameters copied from the update-Q-network. This idea saves the learning process from oscillation, and allows the network to converge more quickly.

However, DQN often suffers from overoptimism, as in each step the optimal action is selected according to the fixed target-Q-network. Once the Q value of an action is overestimated, the agent tends to choose it more often than it should, causing deviations from the optimal solution. That's where double DQN, or DDQN, comes into play - it decouples the estimation of states and actions, and selects the optimal action according to the update-Q-network instead of the fixed target-Q-network, thereby avoiding overestimation of Q values.

2) Our Model:

The application of DDQN in our game is quite straightforward. We train a model with the simple network structure shown in Figure 7.

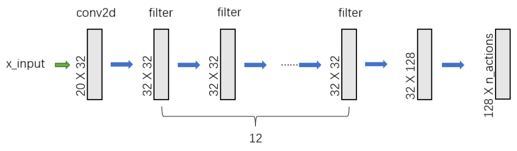
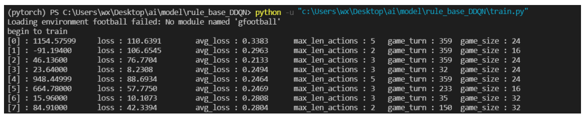


Fig. 7. Structure of our DDQN

In training stage (Figure 8), we feed the environment information - i.e. the game states - into the network, get a action, and then update the target net according to the reward we get from the online net. The model is saved periodically during the training process as checkpoints, and the final model is obtained from thousands of turns of training. Then, when put into actual games, it simply takes the game states as input in each turn, and quickly predicts the best actions for the player.



```
(pytorch) PS C:\Users\wx\Desktop\ai\model\rule_base_DDQN> python -u "c:\Users\wx\Desktop\ai\model\rule_base_DDQN\train.py"
Loading environment football failed: No module named 'gfootball'
begin to train
[0] : 1154.57599    loss : 110.6391      avg_loss : 0.3383     max_len_actions : 5   game_turn : 359  game_size : 24
[1] : -91.19400     loss : 106.6545      avg_loss : 0.2963     max_len_actions : 2   game_turn : 359  game_size : 16
[2] : 46.13600      loss : 76.7704       avg_loss : 0.2133     max_len_actions : 3   game_turn : 359  game_size : 24
[3] : 23.64000      loss : 8.2308        avg_loss : 0.2494     max_len_actions : 3   game_turn : 32   game_size : 24
[4] : 948.44999     loss : 88.6934       avg_loss : 0.2464     max_len_actions : 5   game_turn : 359  game_size : 24
[5] : 664.78000     loss : 57.7750       avg_loss : 0.2469     max_len_actions : 3   game_turn : 233  game_size : 16
[6] : 15.96000      loss : 10.1073       avg_loss : 0.2808     max_len_actions : 3   game_turn : 35   game_size : 32
[7] : 84.91000      loss : 42.3394       avg_loss : 0.2804     max_len_actions : 2   game_turn : 150  game_size : 32
```

Fig. 8. Training of the DDQN Model

As is in the rule based model, the actions of the game agent are catagorized into city actions and unit actions. The city tiles' actions are still dictated by rules, but unlike what is discussed in section 3, here we need to consider researching as well. This will be covered in the next section, as RL models and IL models share the same strategy for cities.

As for the workers' actions, they are produced by the DDQN model. In each turn, the game agent sends the game status into the network, and the network will calculate the Q values of possible actions and return the most promising one for each worker on the map, who will carry out the commands from the model without any objection. The mission list for workers is as below:

1) find a suitable position nearby to build a city tile;
2) return to the closest city;
3) move to a friendly city cluster;
4) gather resources;

Of course, aside from these high-level designs, there are many special situations that may cause trouble for the agent. For example, we let a worker abort its mission and get a new one, if it has been blocked by the opponent for multiple turns. Readers interested in these technical details may refer to our source code, as discussing them here may take up too much space.

3) Performance of the DDQN Model:

Figure 9 shows a screenshot of a game played by this model. We observe that while this model performs quite satisfactorily on small maps, it does not seem to adapt well to large ones.

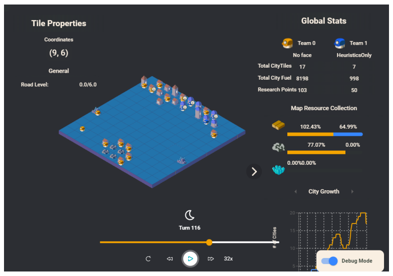


Fig. 9. A screenshot of our DDQN Model

## B. DDPG Model

1) A Brief introduction to PG and DDPG:

The Policy Gradient algorithm, abbreviated as PG algorithm, is another frequently used algorithm in RL. Unlike value-based algorithms like Q learning and DQN, which evaluate the Q values of $(state, action)$ pairs given the *state* and then produce a deterministic result of the best action, PG algorithm takes a stochastic approach, and does not calculate the Q value at every step. In this algorithm, the policy maintains a list of probabilities of different actions to take in a certain environment state. In the training stage, if an action is found to yield a high reward in a state, the algorithm increases its corresponding probability, instead of updating the Q value associated with the state-action pair, as Figure 10 indicates.

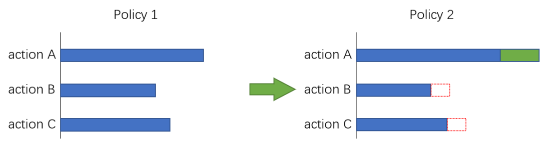


Fig. 10. Update the action probabilities in a given state

However, the application of PG algorithm is limited to low-dimension discrete state space. To tackle problems in high dimension or continuous space, its variant Deep Deterministic Policy Gradient, or DDPG, is introduced. Similar to DQN, the DDPG algorithm applies neural networks into PG - more specifically, Actor Network and Critic Network. However, while the Q-network in DQN calculates the Q values of every $(state, action)$ pair to determine the best action, the Actor Network in DDPG estimates the probability of every *action* in a given *state*, and outputs an *action* according to the probabilities.

The Critic Network will then evaluate the Q value of this $(state, action)$ pair, which will be used to update the parameters of the networks.

2) Our Model:

Figure 11 shows an overview of the training process of our DDPG model. The current game state is fed into the Actor Network as input, which will evaluate the probabilities of each action to take in this state and accordingly recommend one with higher odds. This chosen action, along with the game state, is then sent into the Critic Network to estimate its Q value. In the Critic Network, two target subnets are maintained in parallel: Actor_Target Network and Critic_Target. Together they calculate the Q' value, and the parameters of the Actor Network and Critic Network are updated accordingly. The parameters of the two target subnets, however, are only updated periodically from the actor and critic to stabilize the training process.

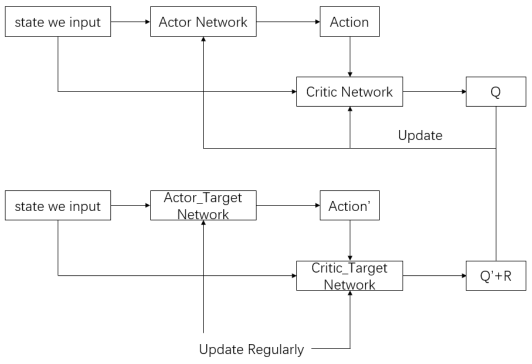


Fig. 11. Training process of DDPG model

## C. PPO Model

1) A Brief Introduction to PPO [5]:

Proximal Policy Optimization (PPO) is yet another important idea in reinforcement learning. PPO algorithm is based on PG, but overcomes many shortcomings of PG. As we previously mentioned, PG algorithm can only be applied to problems with discrete action space, but PPO is not subject to this constraint as it uses policy distribution function instead of probability tables to evaluate possible actions. Mathematically, a policy distribution function can be succinctly represented by several parameters, like the Gaussian distribution. This property is utilized by PPO, which outputs these parameters of a distribution instead of specific actions. To further determine upon an action, we can simply sample the produced policy distribution function.

Additionally, the Actor-Critic (AC) network architecture is also used in PPO. The AC architecture is just like the one we used in DDPG, as in Figure 11. The Actor Network determines a best action according to the probability distribution, and the Critic Network is used to calculate the Q value of the $(state, action)$ and update the parameters according to the TD-error between the Critic Network and Critic_Target Network.

Besides, PPO algorithm exploits Importance Sampling to learn from another policy. When updating the parameters according to the TD-error between the Critic Network and Critic_Target Network, the TD-error is first multiplied by an importance weight. The Importance Weight is defined as

$$IW = \frac{P(a)}{B(a)}$$

where $P(a)$ is the probability of taking action $a$ under target policy (the one we are optimizing) and $B(a)$ is the probability of taking action $a$ under the second policy (which is not updated and only produces actions and data) [6]. In this way, the errors caused by the differences between two policies can be avoided. But it should be noted that PPO is still on-policy, which means the policy used to collect data is evaluated and improved over time as well.

2) Our Model:

After much deliberation and experiments, we chose OpenAI PPO model to play this game as it proves to be most effective. Since PPO model is readily provided as a high-level abstraction by OpenAI, our work to train a PPO agent is relatively simple. It suffices to import the libraries (openai gym + stable_baselines3), define two agents, and start a game. One agent uses the PPO model that we are training, while the other follows an existing model (either rule-based or machine-learning-based) and serves as the opponent.

Figure 12 demonstrates the smoothed learning curve of our model.

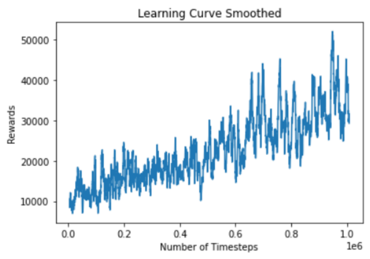


Fig. 12. Smoothed learning curve of PPO Model

3) Performance of the PPO Model:

A screenshot of our PPO model is given in Figure 13 (Our Team is the yellow Team0). The score of this model is not that ideal - less than 1000 - because it has not been trained to reach convergence.

As a matter of fact, this is a problem shared by all three reinforcement learning methods. Since in RL algorithms,

Fig. 13. Screenshot of the PPO Model

an agent is left on its own to figure out a desirable action policy through exploration and exploitation, the process may take quite a lengthy time, especially without high-power GPUs. That's where imitation learning comes into play, where we provide the agent with examples to follow.

## V. IL Models

### A. The Voting Mechanism

The voting mechanism is one of the most basic ideas in imitation learning. As the name suggests, it makes a decision by counting votes from different policies. When training an IL agent, we provide it with a number of existing policies - either produced by other models, or observed from the games of better players, especially those at the top of the public leaderboard. When the agent needs to take an action, it will ask these policies for it, and each policy will produce its own solution (some of which may be the same). These actions are polled together, and the one among them that's recommended by most policies will be chosen. Ties are broken by random selection.

Our group trained three IL models following this idea: the first one from a large data set, the second one from data sets of the top players using RESNET18, and the third from the top player data set as well, but with decreasing learning rate.

### B. IL with UNET Structure

#### 1) A Brief Introduction to UNET:

UNET is a neural network structure proposed in 2015, and originally used in medical imaging field [4]. It is an end-to-end model that achieves higher utilization of limited amount of training samples through data augmentation. The original structure of UNET is shown in Figure 14.

The core idea of this network structure is three components: downsampling, convolution, and upsampling. In the training stage, the input is first encoded into a smaller scale through polling, extracting low-level features in the process. Then, after convolving, the features are decoded back to a lager scale, in the meantime extracting features in higher levels. And thus the feature map from the

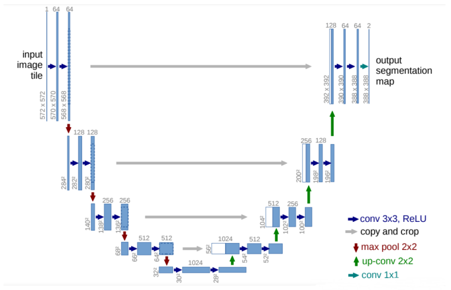


Fig. 14. UNET Structure

encoding stage will be combined with the that from the decoding stage.

The benefits of UNET are listed as below:

1) ensures that the final recovered feature map incorporates more low-level features;
2) enables the fusion of features of different scales;
3) allows for multi-scale prediction and deep supervision.

These benefits motivated us to apply this network structure into our IL models.

#### 2) Our Model:

The UNET structure we used to train our model is shown in Figure 15. It is adapted from the original UNET with different scales to better fit the game we are playing. The input features are position-dependent, and reflect detailed information of each unit or resource on the map. The output is an action map, indicating what kind of action will be taken in each position of the map. We also add to the bottleneck of the network some global features of the game, such as resource points, number of cities and units, average fuel condition, game turn and amount of remaining resources. They imply the global state of the game, and are not related to any specific position. We believe combining local features with global ones in this way can help the agent learn better.

This UNET structure is used to imitate the policy observed from the player with high scores. When training, we track the actions of players in the game, and send them into the network, where they will be classified as features with respect to the current game state. Once trained, we can simply feed the game state into the model, and get an action from its output in a real game.

### C. UNET Model with 4 Directions

This is an interesting idea that we applied to the primitive UNET IL model introduced in the previous subsection: we rotate the map in four directions. In this way, we can control the movement of units with only one instruction to move north. Hence the output dimension of the UNET is reduced from 6 to 3, corresponding to the three possible actions of units: move north, build a city,

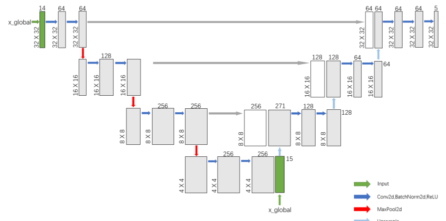


Fig. 15. UNET Structure of our model

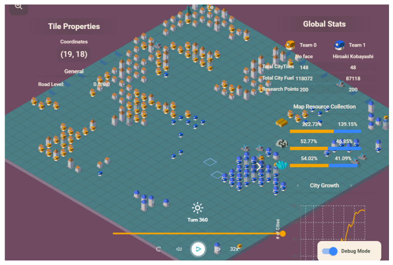


Fig. 16. Screenshot of IL agent with Attention Model

and stay still. This simplification reduces the complexity of the network, and produces better performance to a certain extent.

### D. UNET Model with Attention

#### 1) A Brief Introduction to Attention Mechanism:

Attention Mechanism originated from the field of Natural Language Processing [1], and is now a widely adopted method in machine learning. The core of this method is just like the attention of humans: when we look at a picture or read a piece of text, we usually won't pay attention to all the details at the same time. Instead, we fix our eyes on a certain area of the picture or the text, and only see the other parts through the corner of our eyes. Applied in machine learning, Attention Mechanism helps a model focus on the important information, and learn to assimilate it more thoroughly. This helps to overcome the curse of dimensionality, i.e. when the scales of the input and output are very large and the encoding, decoding processes are quite long, information may be lost in midway.

#### 2) Our Model:

We add two network layers to our UNET: Channel Attention Module and Spatial Attention Module. Channel Attention Module polarizes large values in the input, and the Spatial Attention Module helps to focus on the large values of the output. Together these two layers form the CBAM layer to achieve the Attention Mechanism. In our UNET model, three CBAM layers are added to to each level to emphasize the features we want to keep focus on.

#### 3) Performance of the Attention-based IL Model:

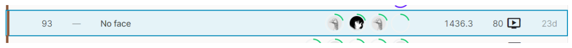
A screenshot of the games played by the attention-based UNET-IL model is shown in Figure 16. The score of this model is 1436.3.

### E. City Strategies in IL Models

As we mentioned in section 4, both IL and RL models only cover the decision for worker actions. In these models, city tiles greedily build workers whenever possible, and do research otherwise. However, we later incorporated ideas from the rule-based models into the machine learning methods. With the help from the modified Game class introduced in section 3, we assign research tasks to city tiles distant from the enemy, and let those on the frontline build workers, so that the newly produced workers can throw themselves into battle as fast as possible. Also, we stop city tiles from building units when the research point draws near to the breakthrough for coal or uranium technology, and focus on researching instead. Together, these two improvements can make our models more aggressive, and pose greater pressure on the opponent agent.

## VI. The Highest Agent

Our team submitted 80 agents in total, and the highest score is 1436.3, from the UNET structured IL model.



93 — No face 1436.3 80 23d

Fig. 17. Score of the UNET structured model

## References

[1] Jan Chorowski, Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. End-to-end continuous speech recognition using attention-based recurrent nn: First results, 2014.

[2] Bhavesh Dhera, M Mani Teja, and G Yashwanth Reddy. An overview of deep q learning. 2019.

[3] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. ACM Comput. Surv., 50(2), apr 2017.

[4] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[5] John Schulman, Oleg Klimov, Filip Wolski, Prafulla Dhariwal, and Alec Radford. Proximal policy optimization. https://openai.com/blog/openai-baselines-ppo/#ppo, 2017.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.