CS 584 MACHINE LEARNING
SPRING 2023
PROJECT REPORT

Topic: Intelligent AI bot for a 2D game using Reinforcement Learning
Members (1 person): Shashank Parameswaran

1 Introduction:

In today's world, Artificial Intelligence (AI) has become an essential tool for solving complex problems in many domains. One such domain is game AI, where AI agents can be trained to play games and compete with human players. In this project, the idea is to build an AI bot for a 2D city-building game that involves resource collection, optimization, and survival for the Lux AI challenge, utilizing reinforcement learning. Video games are generally used as a platform to test any Reinforcement learning algorithm as it mimics the complexity of a real-world environment. Specifically, this project will be using Proximal Policy Optimization, which is an on-policy reinforcement learning algorithm that tries to identify the optimal action to take during gameplay and enhance its performance.

Previous Work:

Reinforcement learning (RL) algorithms have been successfully applied to video games to evaluate their efficacy in solving complex decision-making problems. Training RL agents on video games has numerous advantages, including providing a controlled and reproducible environment, the availability of rich sensory input, and the potential for scaling to real-world applications.

In a study by OpenAI "Dota 2 with Large Scale Deep Reinforcement Learning" (2019), the authors trained several RL agents for the popular video game Dota 2, and evaluated its performance against human players. The agents were able to learn complex strategies such as team coordination, resource management, and decision making in a dynamic and unpredictable environment. The agents were able to achieve a win rate of more than 99% against amateur players and were able to compete and win against professional players in exhibition matches.

Similarly, another study by Vinyals et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning" (2019) trained an RL agent using the game Starcraft II and showed that the agent was able to achieve superhuman performance in tasks such as resource management and unit control.

Recent advances have concentrated more on off-policy algorithms like MARL, and in a study by Yu et al. on "The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games" (2022) showed that PPO achieves stronger results in both final returns and sample efficiency that are comparable to the state-of-the-art methods on a variety of cooperative multi-agent challenges, which suggests that properly configured PPO can be a competitive baseline for cooperative MARL tasks.

2 Problem Statement:

The Lux AI challenge was a competition hosted by Kaggle in 2021 where participants need to design an AI bot/agent that competes with another agent (bot) in a 1vs1 duel in a 2D game. The goal of the game is to own as many City Tiles as possible. This is achieved by gathering in-game resources, creating Units and creating Cities. Both teams have complete information of the 2D map, i.e., information on location of resources and information of state of the other player.

2.1 The Game:

The game is played on a 16*16 tiled 2D platform and the player with the maximum number of city tiles win. Each player starts with 1 city tile and one unit each. The game lasts for up to 360 turns with cycles of days lasting 30 turns and night lasting 10 turns. During night time, units and cities need to consume a specific amount of resources in order to stay alive. City tiles can spawn workers, carts or gather research points as an action with the limit of one unit, that is, worker or cart per city tile. Each unit can move in directions, North, South, East, West or Centre by one step. Units automatically collect resources from adjacent tiles and they need to move into and deposit these resources in any city tile to generate fuel. Fuel is required to spawn units, gather research points and keep the city tiles alive during night time. Resources are spread throughout the map and there are 3 types of resources – wood, coal and uranium. Coal and Uranium require a specific set of research points in order to be mined. The game ends at turn 360 or if either player loses all cities and units. This is a basic overview of the game, and there are many more specifics to each part which are available in detail in link provided in Appendix-IV.

3 Methodology:

Any Reinforcement learning algorithm would follow the classic appraoch shown below

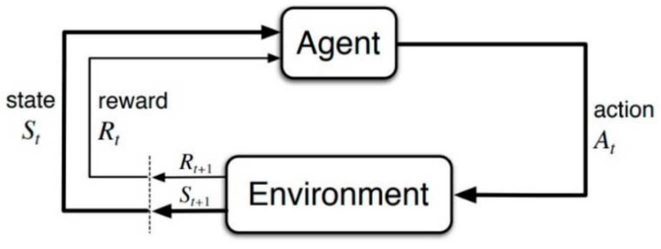


Figure 1: Basic idea of Reinforcement Learning

Using state S_t of the game at every turn, it performs actions A_t and collect rewards R_t for performing those actions. The agent interacts with the environment every turn and tries to maximise the rewards. PPO is a model free algorithm and it tries to learn how to take an action (A_t) given a state S_t. Although we had proposed to use Q learning, Q learning cannot directly be used in environments with continuous action/state spaces. For this reason, we are using the PPO algorithm. PPO comes under the Policy Optimization class of algorithms and tries to learn a policy directly.

For a vanilla policy optimization algorithm, the policy gradient loss is given by

$$L^{PG}(\theta) = \hat{E}_t\big[\nabla_\theta \log(\pi_\theta)\,(a_t|s_t)\hat{A}_t\big]$$

Where π_θ is the policy (basically a neural network) that takes the observed states as input and suggests actions as output

$\hat{A}_t$ Advantage is the relative value of the selected action which is the difference of discounted rewards (actual value of rewards for the current episode) and the baseline estimate (what is the estimated value of reward for the current episode) where,

a_t is the action at current state being taken

s_t is the current state

If the value of the advantage function $\hat{A}_t$ is positive, it increases the likelihood of the action taken and vice versa if negative. When using gradient descent to update our policies, we need to ensure that our new policy is not too different from the old policy. For achieving this, we can add a KL constraint to the optimization objective (used in TRPO method). However, we need to ensure that we are not increasing the computational overhead. PPO achieves this by using a simple and intuitive loss function as shown below:

$$L^{CLIP}(\theta) = \hat{E}_t\big[\min\big(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\big)\big]$$

where $r(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio of new policy and old policy

The $r_t(\theta)\hat{A}_t$ is the usual policy gradient objective function that tries to find the best policy which gives the maximum Advantage. The second term is simply a clipped version of the first term. This helps us prevent a huge update to the gradient when the action is a lot more probable from the last gradient step.

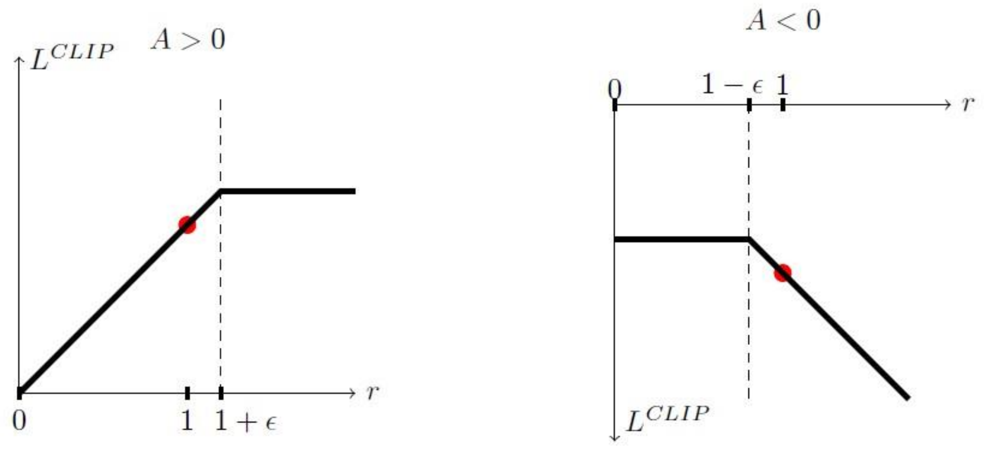


Figure 2: Left: how clipping works when Loss is positive and high. Right: how clipping works when loss is negative

The PPO algorithm is shown below:

```
for iteration=1, 2, . . . do
    for actor=1, 2, . . ., N do
        Run policy $\pi_{\theta_{old}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1 \dots \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{old} = \theta$
end for
```

The MLP policy (the neural backbone) in PPO is used in this project and takes in the current state of the environment as input and produces a probability distribution over possible actions. This distribution is used to sample an action to take in the current state. The MLP policy in PPO works by representing the policy as a neural network and using a surrogate objective function and a proximal constraint to update the policy during training. This allows the policy to learn to produce actions that maximize the expected cumulative reward while maintaining stability during training.

4 Progress:

4.1 Getting Started

Major part of the project in the initial stage involves understanding the codes used for the environment. A lot of boilerplate code is involved for setting up the game environment for various objects like units, map, resources, actions and others. These codes are provided as a part of the Lux environment (listed in Appendix-III) and we need to use them to obtain in-game metrics for performing any function. For example, if we need to find the closest resource from an Agent, then we need to find all the available resources in the map and then find the one that is closest relative to the Agent's position.

4.2 Importing packages

The Python starter kit for Lux environment is provided which contains python codes for setting up the objects for the game environment like units, map, resources, actions and others. These codes are imported as a part of the environment and used as needed. We will also be using PPO model from the StableBaseline3 package.

Stable Baselines3 is a powerful, open source and user-friendly deep reinforcement learning library that is built on top of Pytorch. It provides a set of high-quality implementations of popular reinforcement learning algorithms, including PPO, A2C, SAC, TD3, and DDPG. The Lux AI environment is compatible with SB3 and this project uses the PPO algorithm using this package. This project will primarily use the above packages for most of the processing.

4.3 Coding

Once we are ready with the environment and packages, we start building the Agent class which will contain the observation space (input to the model), the action space (output for the model) and rewards. The skeleton for a StableBaseline3 (package used for PPO) class is provided by the Lux environment and we need to essentially write wrappers for the key functions like getting rewards, getting observations and provide our own custom functions as needed like getting the closest resource or getting opponent statistics.

Observations Space:

The inputs to the model are not predefined and we needed to define our own observation space for this problem. First, we started with a few observations like unit type, available spaces for the unit to move, free tile Y/N, current resource amount, fuel required to survive, number of turns to night time, research points and opponent statistics. However, this did not train the bot well as we did not give any direction as input (Direction to nearest resource/city). Adding Direction and distance to nearest resource/city made a huge difference to the model as it helped evaluate the rewards obtained based on direction & available resources. Direction variables were one hot encoded for each resource type (3*5) and city (5). All the distance variables in this project were calculated using Manhattan Distance. The final observations space was a 44x1 matrix

Type	Info	Value
Unit	Current resource value	1
	Distance from nearest city	1
	Adjacent cell resources Y/N	4
	Direction to nearest resource & city (OHE)	15 + 5
	Distance to nearest resource & city	2
	Adjacent cell units Y/N	4
	Others (can act Y/N, current tile info etc.)	6
Environment	Night Y/N, Turns to night	2
	Research points	1
	Number of units	1
Resource	Total deposited fuel in city, Other	2
	Total	44x1 matrix

Action Space:

The action space (output) proposed for this problem is standardized here as it would be number of possible actions by a unit + number of possible actions by a city. The trained model would output the action with the highest SoftMax probability based on the input given. The value simply shows the number of outputs from the model for that specific action. The model will output a 9x1 matrix for this problem.

Type	Action	Value
Unit	Centre	1
	North	1
	South	1
	East	1
	West	1
	Build city	1
City	Spawn Worker	1
	Spawn Cart	1
	Research Action	1
Total		9x1 matrix

Rewards:

Building the rewards function was the most complicated part of this project. In this paper, we propose a reward function for this specific game. The rewards function is designed to encourage the agent to take actions that lead to the optimal use of resources, building of cities, and management of units.

Reward Type	Value
Inc. Cities from last turn	Inc. cities*300
Inc. Units from last turn	Inc. cities*100
Inc. research points from last turn	Inc. research*10
Inc. resources from last turn	Inc. resources*10
Unit roaming penalty (with full cargo)	20 for each unit
Unit distance penalty from city at night	Dist*(2,4 or 7) based on turns to night time
Penalty for not building city when cargo is full	40 for each unit
Penalty for Inc. distance from resource from last turn	Inc Dist*(5 or 20) based on night Y/N
Penalty for building city at night	50 for each unit

Positive rewards are given for gathering resources/building cities and negative rewards (penalties) for undesirable behaviour. Undesirable actions were uncovered from watching game simulations with the trained model. For example, repeated simulations revealed that the unit tends to roam around when its cargo is full. For this reason, a roaming penalty and 'Not building a city' penalty was introduced. The values/multipliers for each reward type were chosen based on the importance of that reward to the game. For example, since building cities is the main objective, it is given higher reward multiplier compared to gathering research points. The rewards were modified based on Agent behaviour over multiple simulations. All the rewards were summed up to give one reward value for each turn.

Other Functions:

As shown in the Appendix-II, multiple functions were built to facilitate obtaining the observation and reward function. These functions relied on the game and environment functions to extract essential information like resource amount in a cell, number of cities, unit position etc. In Appendix-III, we have provided the list of python files that contain the necessary classes and functions to execute the game, including the environment functions.

5 Modelling

Several hyperparameters were tuned in the process of building the model. A learning rate scheduler was used to train the model for 10M steps. The first 2M steps was trained with learning rate 0.001 followed by 4M steps with learning rate 0.0005 and finally 4M steps with learning rate 0.0001. This helps in gradual steps towards the optimal policy. Gamma value used was the default value 0.9. A higher gamma value means that future rewards are more heavily discounted, while a lower gamma value gives more weight to future rewards.

Parameter	Value
Algorithm	PPO
Policy model	MLP Policy
Learning rate	Schedule [0.001, 0.0005, 0.0001]
Total steps	Schedule [2M, 4M, 4M]
Gamma	0.9
n_steps	15000

The point at which the policy is updated depends on two factors: batch size and n_steps. Batch size refers to the number of simulated episodes or games before updating the policy, while n_steps refer to the number of turns or steps taken. The policy will be updated based on the first condition that is met. We used the n_steps condition with 15000.

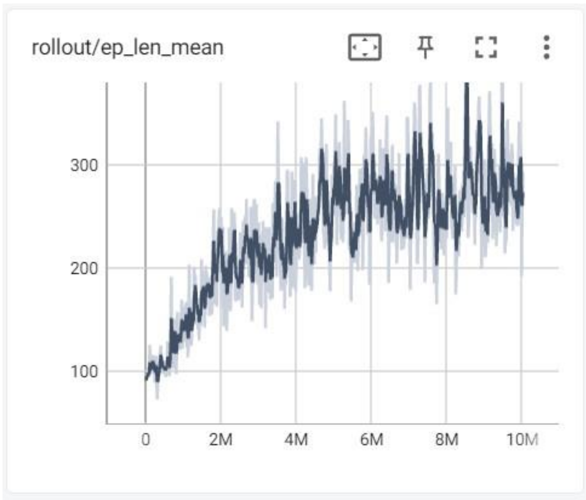
Results:

Opponent	Results	Average Turns	Max Cities
Self	5-5	107	11
Kaggle Agent	0-10	175	31

The model was tested against itself and an Agent downloaded from Kaggle for 10 games. Against itself, it was a draw with 5 matches won by each player. The average turns lasted by the model was 107. Now, a baseline estimate for this game would be a bot which would have an average of 30-40 turns as it would get eliminated during the first night cycle where it would not have gathered enough resources to survive because of its indecision. So, our model is performing almost 3 times better compared to the baseline estimate for the chosen random states. Also, Max Cities is the maximum number of cities built in any one of the 10 simulated games and it was 11 for this match up. Map seeds used in results were 0,1,2,3,6,7,9,15,100,7534.

Against the Kaggle Agent, it lost 0-10 which tells us there is a lot of room for improvement. However, the average turns it lasted was 175 which is great because it is now 4 times better than the baseline for the chosen random states (more concrete information on how the model is performing is discussed below). The maximum cities were also higher with 31 cities being built in one of the games.

Tensorboard logs are used to track the progress of the model and they are an excellent way to understand where we need to improve. We have several metrics to measure the performance of the model like policy loss, mean episode length, mean rewards etc.



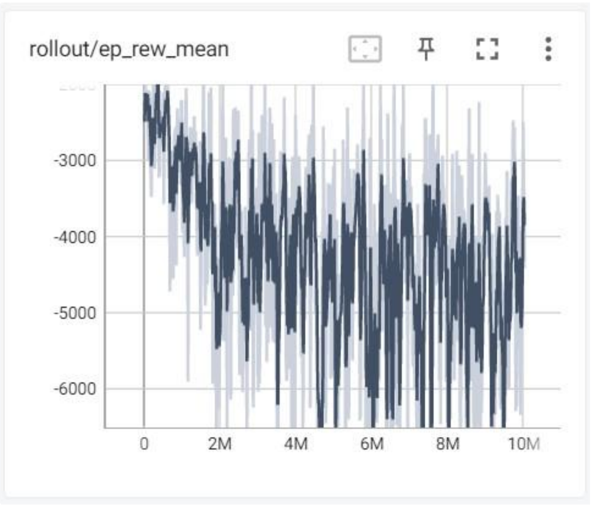
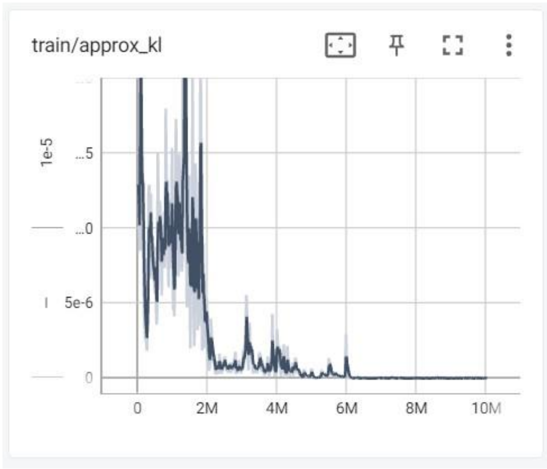


Figure 3&4: Left: Mean episode length vs number of steps. Right: Mean episode rewards vs number of steps

In figure 3, we have the mean episode length against the number of steps. Each episode is 360 turns, i.e., the length of the game. We see that the game ends at 90 initially, but as the model progresses, at 6M steps, it goes up to 270. This indicates that our model is making the right changes in the policy and heading towards the best optimal solution. Figure 4 indicates the mean rewards for each episode. Ideally, this metric should be increasing. But we have given huge negative rewards for losing cities and units. Hence, if the bot does not finish the game, it is most likely going to end the episode with negative rewards. The bot learns to build more cities and units as training progresses and when it loses 10 cities, it is going to end up with (-100*10) compared to losing just one city (-100) initially. It may be necessary to make slight modifications to the reward function to improve its clarity and ease of interpretation.



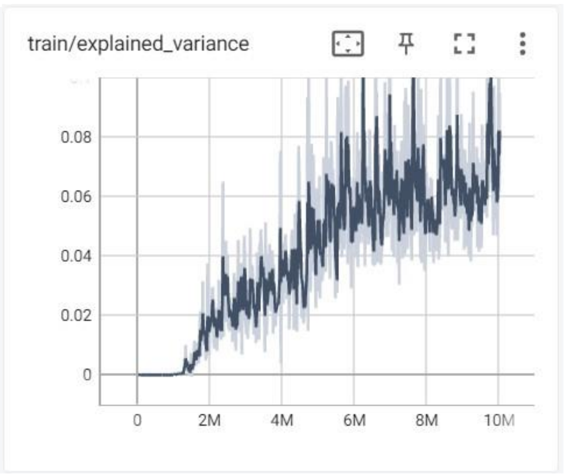
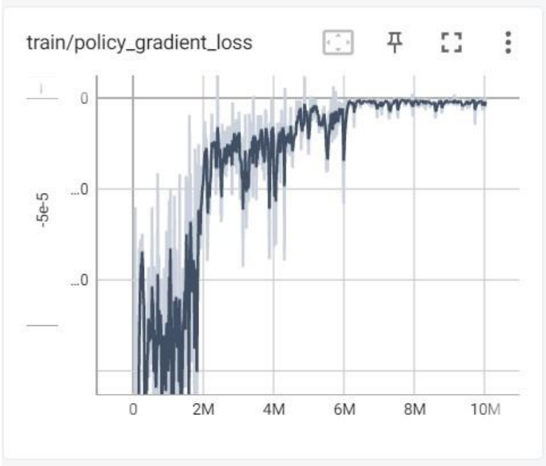


Figure 5&6: Left: Approx KL divergence vs number of steps. Right: Explained Variance vs number of steps

Figure 5 points to the mean KL divergence between the old and new policy and it is an estimate of the degree of change that occurs during an update in the policy gradient algorithm. As we approach 6M steps, the KL is almost 0 which indicates that there are almost no updates to the policy (and this could be our stopping point), and suggests that the agent has converged to a specific strategy. Figure 6 suggests the fraction of return variance explained by the value function, i.e., it measures how much of the variance in the actual returns can be accounted for by the value function and it is nice to have it increasing slowly.



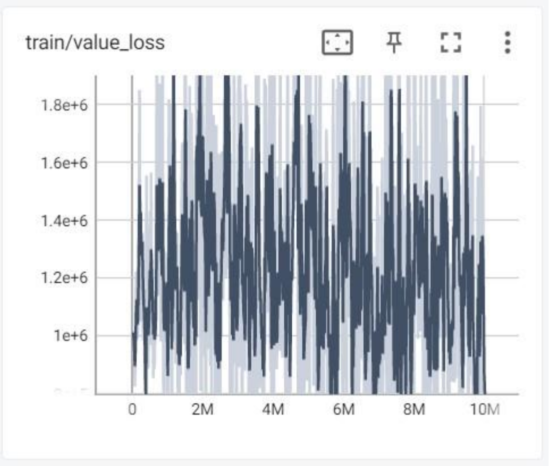


Figure 7&8: Left: Policy Gradient loss vs number of steps. Right: Value loss vs number of steps

Figure 7 reveals the policy gradient loss (as defined in the methodology section). Ideally, this should decrease over time. However, there is a reason why it is increasing. Reinforcement learning is a trade-off between exploration and exploitation. As the model progresses, the bot is able to access more of the later game stages (increasing episode mean) and uncovers usage of other resources like coal/uranium or spawning worker carts or even being surrounded by a lot of cities. As a result, the agent is incentivized to explore more and this could be a reason for increasing policy gradient loss. Other reasons could be insufficient observations or better a reward function. Figure 8 shows the value loss increasing first and then decreases as the model gets better at predicting the value function.

Summary:

Using Reinforcement learning, we have built an AI bot that can collect resources, optimize movement and build cities. By using a customized observation space and reward function, the model can perform approximately 3 times better than a random baseline bot as shown above with the average turns increasing from ~90 to ~270 as we reach 6M steps. The ideal stopping point for the model is 6M steps as there are no updates in the policy post that. Details are provided in the Appendix-I on how to execute and run the codes.

Future Work:

We have already indicated that there is room for improvement in the model. There could be multiple reasons for lack of improvement after 6M steps - rewards function needs to be improved, learning rate is too low or quality of inputs used. Furthermore, the smart transfer action, which allows for the transfer of resources between bots, was not utilized in this model. Implementing this action could potentially reduce unnecessary movements and improve overall efficiency. The rewards are also not curated for spawning carts, which are units with higher capabilities. We could consider improving this aspect of the rewards function to enhance its performance.

In terms of model training, we could consider using a more intelligent agent for the later stages of training (beyond 6M steps), and also explore the possibility of utilizing a more granular rewards function during the initial stages of training, before transitioning to a generalized rewards function later on.

Replays:

In order to fully appreciate and understand the working of the model, I would suggest to watch the replays of a couple of games available in the "LuxAI->replays" folder as shown in Appendix-I. The replays take only a couple of minutes and it can be fast forwarded. To watch a replay, simply upload the JSON replay file on https://2021vis.lux-ai.org/ and you will be able to view it.

References:

[1] Schulman, John, et al. "Proximal Policy Optimization Algorithms." ArXiv.org, 2017, arxiv.org/abs/1707.06347.

[2] Openai, et al. Dota 2 with Large Scale Deep Reinforcement Learning. 2019.

[3] Vinyals, Oriol, et al. "Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning." Nature, vol. 0, no. 0, 2019.

[4] Yu, Chao, et al. The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games. 4 Nov. 2022.

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning.

[6] Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A brief survey of deep reinforcement learning.

[7] "Logger — Stable Baselines3 1.8.0a3 Documentation." Stable-Baselines3.Readthedocs.io, stable-baselines3.readthedocs.io/en/master/common/logger.html.

Appendix-I:

Please refer to the documentation in https://github.com/Lux-AI-Challenge/Lux-Design-S1 if you have trouble in setting up using the below steps. The instructions are provided for Windows but most of it should work with Mac as well.

In order to run the codes please follow the below steps as indicated.

- To run the codes in Jupyter notebooks:
    - Open the Lux_AI_RL_Bot.ipynb file in Jupyter

- Copy the 'luxai2021' folder available in 'LuxAI' in the same space where the Jupyter notebook is located. If you need to use another location, please change the import command as needed
- Then install the below packages (if I missed any please install them as required):

```
pip install stable-baselines3

pip install gym
```

- Now you can start running the codes. Please note that it takes 4 hours for the model to run. You can change number of the iterations in the variable 'schedule' to 1000 or some other number if you want to run a sample

➢ To view Tensorboard logs:
- Tensorboard log files are provided under 'lux_tensorboard'
- Open the command line and run the below code after changing the path to folder:

```
tensorboard --logdir="path_to_folder\LuxAI\lux_tensorboard"
```

- Navigate to http://localhost:6006/ to view the results

➢ To test the model by running a single game:
- Install the competition design using the below command in a command line interface:

```
npm install -g @lux-ai/2021-challenge@latest
```

- Navigate to the folder where the 'main2.py' is located. This is available in LuxAI -> codes
- Run the below code (you can ignore the cloudpickle warnings). If you encounter problems with model file not being located, please open main2.py and give the right path

```
lux-ai-2021 --seed=6 main2.py main2.py --maxtime 10000
```

➢ To view the results for 10 games at once:
- Navigate to the folder in the command line where the 'main2.py' is located. This is available in LuxAI -> codes
- Run the below code. This bat file has been packaged with 10 game commands with different seeds to test the results. Change the bat file as needed for Mac.

```
run_10_games.bat
```

- Then navigate to the Jupyter notebook and run the second 'code' cell under 'Appendix'. Change the directory as mentioned in the notebook

➢ To watch a replay of the game for any of the files, simply upload the JSON replay file on https://2021vis.lux-ai.org/ and you will be able to view it. A few replay files (JSON) are available in the LuxAI->replays folder. Also, for any games you simulate using the lux-ai-2021 command using the model, it should be stored in another 'replays' folder which would be available from wherever you executed the code. Although I have taken a screen recording, I am not uploading it because of its size. Please let me know if you need it and I will be happy to send it across.

Appendix-II:

A. Main Wrapper functions:

Below are the wrapper functions that were modified

Function Name	Arguments	Return type	Description
game_start()	Game	None	This function is called during the start of the game and initializes a few variables
get_observation()	Game, Unit, City_tile, int, is_new_turn	List[int]	This is the input that goes into the model. Several functions defined under custom function in Appendix-II are used to get the required observations
get_reward()	Game, is_game_finished, is_new_turn, is_game_error	Int	Returns the reward for the current turn based on the actions taken.

B. Custom Functions used to obtain in-game metrics:

Note: Most of the arguments are of Class type referring to the files in Appendix-III

Custom Function Name	Arguments	Return type	Description
get_unit_obs()	Game, Unit	List[int]	Returns the observations related to a unit
get_env_obs()	Game	List[int]	Returns the observations related to the game environment
get_resource_obs()	Game, Unit	List[int]	Returns a list of all observations related to resources
get_resources()	Game	List[List[str, int, int]]	Returns a list of all resources with its type and location
find_nearest_resource()	X, Y	Int	Finds the Manhattan distance from the nearest resource location
get_map_info()	Game	List[int]	Returns the OHE list of all cells in the map based on resource/Unit/City
get_nearest_city()	Game, Position	Int	Finds the Manhattan distance from the nearest city
direction_to_nearest_city()	Game, Unit	Int	Returns an integer based on the direction towards the nearest city

direction_to_nearest_resource()	Game, Unit	Int	Returns an integer based on the direction towards the nearest resource

C. Functions from other sources:

Here are the functions included in the code and obtained from other sources:

They are obtained from other sources, as they are standardized and used only for inference, and do not impact the results of the project.

Function Name	Arguments	Return type	Description
process_turn()	Game, Team	List[Action]	Decides on a set of actions for the current turn. Not used in training, only inference. This is sourced from 'Agent.py' file in 'luxai2021\env\'
get_agent_type()	None	Constant	Returns the type of agent. This is sourced from 'Agent.py' file in 'luxai2021\env\'
action_code_to_action()	Int, Game, Unit, City_tile, int	Action	This function coverts the action code given into an action. This is obtained from a Kaggle notebook [1].
take_action()	Int, Game, Unit, City_tile, int	None	This function transfer the action to the controller which takes effect in the game. This is obtained from a Kaggle notebook[1]

[1] https://www.kaggle.com/code/glmcdona/reinforcement-learning-openai-ppo-with-python-game

Appendix-III

Python files available in lux AI package 'luxai2021' which were used to obtain in-game metrics:

actions.py	city.py	lux_env.py
actionable.py	constants.py	match_controller.py
actions.py	game.py	position.py
agent.py	game_constants.py	resource.py
cell.py	game_map.py	unit.py

Appendix-IV

Link Description	**Link**
Lux AI game details	https://www.lux-ai.org/specs-2021
Lux AI replay viewer	https://2021vis.lux-ai.org/
Lux AI Github docs	https://github.com/Lux-AI-Challenge/Lux-Design-S1