CSC3180: Fundamentals of Artificial Intelligence, Spring 2023
Project Report, Group 10
May 7, 2023

Adaptive Dynamic Stabilization of a Self-Balancing Scooter Using Epsilon-Greedy Q-learning

Zixing Jiang[1], Xuanyang Xu[1], Muhan Lin[1], Hongyi Yang[2]

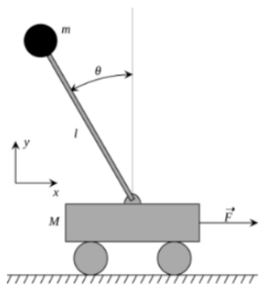
Abstract

In this project, we pointed out the inability of traditional dynamics-based control methods to adapt to changing loads when stabilizing a self-balancing scooter. We utilized reinforcement leaning to address the problem. Through the Epsilon-Greedy Q-Learning algorithm, we let the agent update the optimal control policy in real time. Simulation with OpenAI Gym shows our algorithm can teach the agent how to balance and adapt to loads of different weights and centers of mass.

1 Introduction

The self-balancing scooter (Figure 1a) is a common urban commuter tool. One of the most critical features in self-balancing scooter is dynamic stabilization, which allows the scooter to maintain its balance while in motion. Without this feature, it would be very difficult for riders to stay upright and control the scooter, and even cause safety accidents in severe cases. In order to achieve dynamic stabilization, the traditional model-based automatic control methods, including Proportional-Integral-Derivative (PID) control and Model Predictive Control (MPC), model the self-balancing scooter as a cart-pole system (Figure 1b) and designs the controller by deriving its system dynamics. However, the dynamics-based control is sensitive to the rider's weight and center of mass (CoM) because the weight and CoM are parameters in the system's dynamics, as shown in (1).

(a) Man on a self-balancing scooter



(b) Cart-pole system dynamics

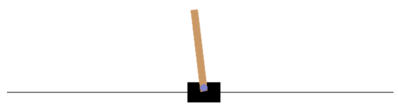

(c) OpenAI Cart Pole Environment

Figure 1: Illustrations of (a) the self-balancing scooter, (b) the cart-pole system dynamics, and (c) the OpenAI Cart Pole environment.

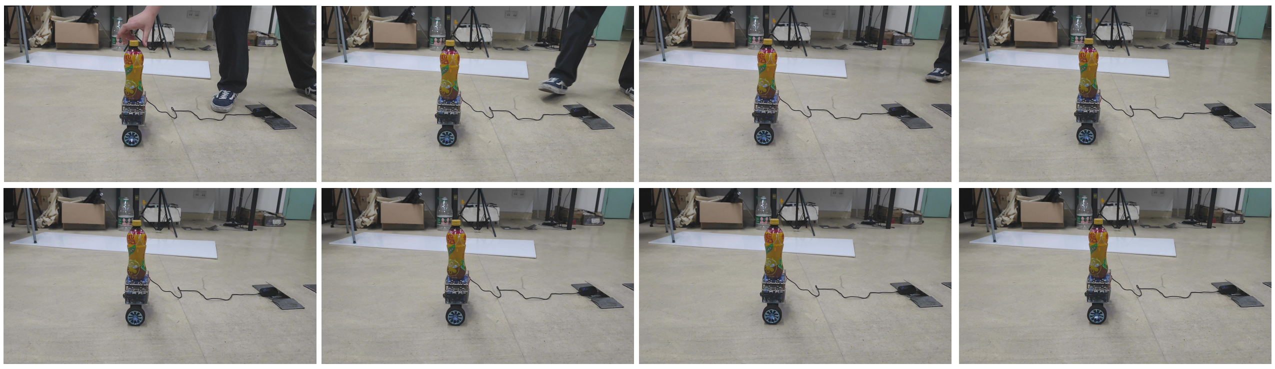
Dynamics of the cart-pole system

$$(M+m)\ddot{x} - ml\ddot{\theta}cos\theta + ml\dot{\theta}^2 sin\theta = F \tag{1a}$$
$$l\ddot{\theta} - gsin\theta = \ddot{x}cos\theta \tag{1b}$$

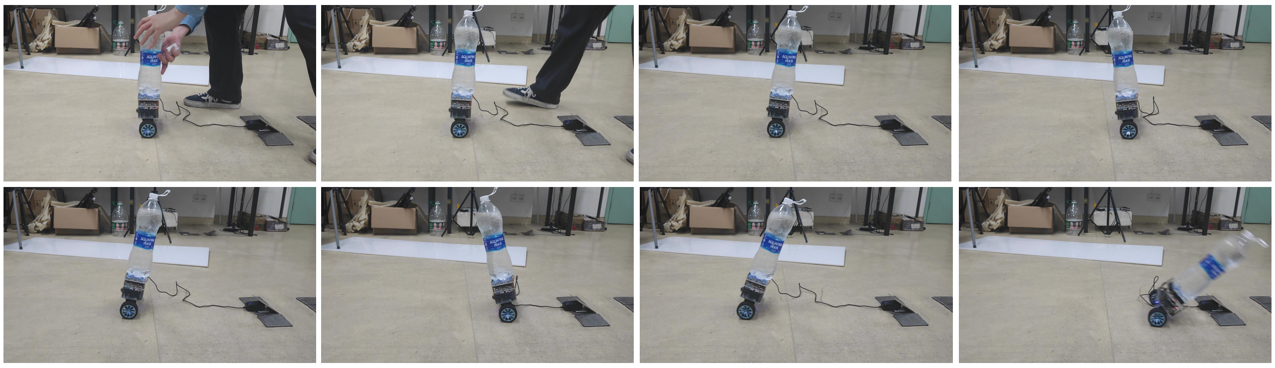
where

- M: the cart mass
- m: the pole mass
- $\ddot{x}$: the cart acceleration
- l: the pole length
- θ: the inclination degree of the pole
- $\dot{\theta}$: the angular velocity of the pole
- $\ddot{\theta}$: the angular acceleration of the pole
- F: the force added to the cart

[1]School of Science and Engineering, The Chinese University of Hong Kong, Shenzhen
[2]School of Data Science, The Chinese University of Hong Kong, Shenzhen



(a) Self-balancing scooter with designed load



(b) Self-balancing scooter with a different load

Figure 2: Model-based control is sensitive to the load's weight and CoMs. (a) The scooter succeed to maintain balance with a designed load. (b) The scooter fail to maintain balance with a different load. Video demonstrations of this phenomena can be found at CSC3180-Cart-Pole/Videos.

It can be seen that the pole mass m and length l get involved in this kinematic model. It means controlling the cart-pole system in a dynamic way requires the information of pole mass and pole length. Once an unexpected change in pole mass or length occurs and these parameters become unknown, it is highly possible for the classical controlling method to fail, as shown in Figure 2. Therefore, for riders of different weights and CoMs, the performance of stabilization with classic model-based control varies.

To address this problem, in this project, we aim to utilize the learning ability of artificial intelligence to design a dynamic stabilizer that adapts to different rider weights and CoMs.

2 Problem Statement

In this project, we aim to solve the adaptive stabilization problem that can be stated as follows: maintain the inclination (θ) of a pole with varying mass and CoM within a certain range by controlling the movement (x) of the cart (leftward or rightward).

To leverage the learning ability of artificial intelligence, we formulate the stabilization task as a reinforcement learning (RL) process. As shown in Figure 3, in each RL loop, the agent acquires an observation of state and selects an action according to its understanding about the environment. The environment sends a reward to the agent for the agent's choice, and transits to a new state based on the agent's action and the environment's inner mechanism. In this project, the RL elements are defined as:

- Agent: The cart
- Environment: The cart-pole system
- Task: Balance a length-varying pole
- State: A four-dimensional vector $\boldsymbol{s} = (\theta, \dot{\theta}, x, \dot{x})$
- Action: Direction of a fixed force exert on the cart. $\boldsymbol{a} = \{left, right\}$
- Reward: An integer. If the pole does not fall down from the cart in one iteration, the sum of reward $+1$.

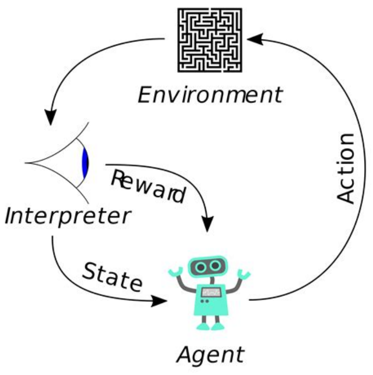


Figure 3: Framework of reinforcement learning

3 Methodology: Epsilon-Greedy Q-Learning

In this project, we solve the formulated reinforcement learning problem with Epsilon-Greedy Q-Learning, which is a well-developed reinforcement learning algorithm that balances exploration and exploitation in sequential decision-making problems. In this section, we will introduce the Epsilon-Greedy Q-Learning algorithm in detail and explain how we link this algorithm to the dynamic balancing problem we aim to solve.

3.1 Q-Learning

Q-Learning is a classic value-based reinforcement learning algorithm that allow an agent to learn from its own actions and rewards in an environment. Unlike some other model-based control and reinforcement learning methods, it does not require a model of the environment. This feature helps to solve the traditional model-based control method's dependence on the rider's center of gravity mentioned in previous sections.

The goal of Q-Learning is to find an optimal policy that maximizes the expected value of the total reward over any sequence of actions. The main idea of Q-Learning is to use a table, called Q-Table, that stores the estimated value $Q(\boldsymbol{s},\boldsymbol{a})$ (Q-Value) of each action $\boldsymbol{a}$ in each state $\boldsymbol{s}$, as shown in Table 1. The Q-Value, which reflects the value of taking a certain action in a certain state, is defined as the expectation of overall received rewards after taking action $\boldsymbol{a}$ at state $\boldsymbol{s}$, as shown in (2).

$$Q(\boldsymbol{s},\boldsymbol{a}) = \mathbb{E}\left(\sum_{t=0}^{\infty}\gamma^t r(\boldsymbol{s}_t,\boldsymbol{a}_t) \;\middle|\; \boldsymbol{s}_0=\boldsymbol{s},\boldsymbol{a}_0=\boldsymbol{a}\right) \tag{2}$$

where $r(\boldsymbol{s}_t,\boldsymbol{a}_t)$ is the reward received from the environment for state-action pair $(\boldsymbol{s}_t,\boldsymbol{a}_t)$ and γ is the discount factor.

In Q-Learning, in order to obtain the most accurate estimate of the Q-Value, the Bellman Equation is applied to update $Q(\boldsymbol{s},\boldsymbol{a})$ in every iteration that state-action pair $(\boldsymbol{s},\boldsymbol{a})$ is chosen, as shown in (3)

$$\text{New } Q(\boldsymbol{s},\boldsymbol{a}) \leftarrow (1-\lambda)\text{Current } Q(\boldsymbol{s},\boldsymbol{a}) + \lambda\left[r(\boldsymbol{s},\boldsymbol{a}) + \gamma\max_{\boldsymbol{a}'} Q(\boldsymbol{s}',\boldsymbol{a}')\right] \tag{3}$$

where $r(\boldsymbol{s},\boldsymbol{a})$ is the reward received in this iteration, $\boldsymbol{s}'$ is the transient state after taking $\boldsymbol{a}$ at $\boldsymbol{s}$, $\boldsymbol{a}'$ is the legal action at $\boldsymbol{s}'$, γ is the discount factor, and λ is the learning rate.

In the Q-Learning process, the agent updates the Q-Table as it explores the environment and receives feedback. The agent chooses the action $\boldsymbol{a}^*(\boldsymbol{s})$ that has the highest value in the current state $\boldsymbol{s}$, according to the Q-Table, as shown in (4).

$$\boldsymbol{a}^*(\boldsymbol{s}) = \arg\max_{\boldsymbol{a}} Q(\boldsymbol{s},\boldsymbol{a}) \tag{4}$$

This way, the agent learns to act optimally over time.

In our scenario, the state vector $\boldsymbol{s}$ of the cart-pole system is continuous. In order to prevent infinitely many state-action pairs, we discretized the system's state space by intervals. The length of each interval is fine-tuned to ensure the effectiveness of learning. Plus, since the Q-Table is keeping updating during the algorithm runtime, it is adaptive to dynamic environment parameters. Therefore the stabilization controller built with Q-Learning is capable of adapting to varying rider weights and CoMs.

3.2 Epsilon-Greedy Action Selection

In order to learn an optimal Q-table, we want our algorithm to explore all possible state-action pairs. Thus, in addition to the action selection criterion (4), an epsilon-greedy policy is applied. Given an epsilon-greedy factor

Actions / States	$\boldsymbol{a}_0$	$\boldsymbol{a}_1$	$\boldsymbol{a}_2$	...
$\boldsymbol{s}_0$	$Q(\boldsymbol{s}_0, \boldsymbol{a}_0)$	$Q(\boldsymbol{s}_0, \boldsymbol{a}_1)$	$Q(\boldsymbol{s}_0, \boldsymbol{a}_2)$	...
$\boldsymbol{s}_1$	$Q(\boldsymbol{s}_1, \boldsymbol{a}_0)$	$Q(\boldsymbol{s}_1, \boldsymbol{a}_1)$	$Q(\boldsymbol{s}_1, \boldsymbol{a}_2)$	...
$\boldsymbol{s}_2$	$Q(\boldsymbol{s}_2, \boldsymbol{a}_0)$	$Q(\boldsymbol{s}_2, \boldsymbol{a}_1)$	$Q(\boldsymbol{s}_2, \boldsymbol{a}_2)$	...
$\boldsymbol{s}_3$	$Q(\boldsymbol{s}_3, \boldsymbol{a}_0)$	$Q(\boldsymbol{s}_3, \boldsymbol{a}_1)$	$Q(\boldsymbol{s}_3, \boldsymbol{a}_2)$	...
...	...	...	...	...

Table 1: An example of Q-Table. $Q(\boldsymbol{s}_i, \boldsymbol{a}_j)$ denotes the Q-Value of a state-action pair $(\boldsymbol{s}_i, \boldsymbol{a}_j)$. The higher $Q(\boldsymbol{s}_i, \boldsymbol{a}_j)$ is, the more worthy to take action $\boldsymbol{a}_j$ at state $\boldsymbol{s}_i$.

$0 \le \varepsilon \le 1$, in each iteration, there is a probability of ε that the agent takes random actions to explore new possible state-action pairs, and there is a probability of $1-\varepsilon$ that the agent exploits investigated state-action pairs via the Q-Table. The overall logic flow of the Epsilon-Greedy Q-Learning is shown in algorithm 1.

Algorithm 1 Epsilon-Greedy Q-Learning

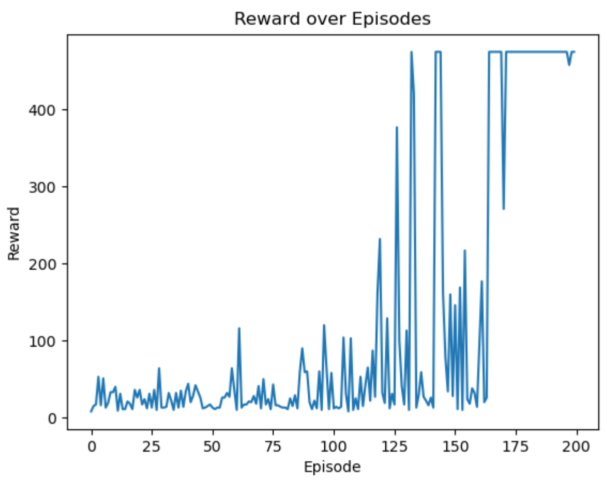
```
Require: observation of state $\boldsymbol{s}$, epsilon-greedy factor $\varepsilon \in [0, 1]$
  while learning not terminate do
    Generate a random number $i \in [0, 1]$
    if $i \le \varepsilon$ then
      Take random action $\boldsymbol{a}$
    else
      Select action $\boldsymbol{a}$ according to the Q-Table                ▷ Equation (4)
    end if
    Update Q-Table entry for $(\boldsymbol{s}, \boldsymbol{a})$ according to the received reward $r(\boldsymbol{s}, \boldsymbol{a})$    ▷ Equation (3)
  end while
```

4 Experiments Analysis

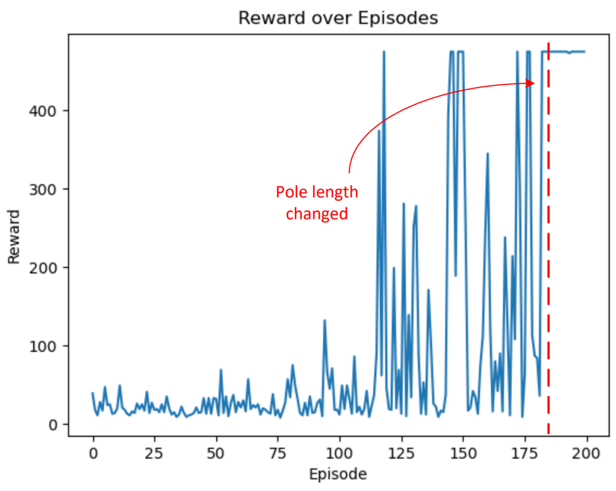
We implemented the proposed Epsilon-Greedy Q-Learning with Python 3.11 and tested it in a modified OpenAI Gym "cart-pole" environment, which will be introduced in section 5.2. Experiment results will be shown in the following. The metric we used is the cumulative reward the agent earned across episodes.

In experiments, parameters of the proposed algorithm are set as follows.

- Epsilon-Greedy factor: $\varepsilon = \max\left\{0.01, \min\left\{1 - \log\left(\frac{i+1}{25}\right)\right\}\right\}$ for episode i
- Learning rate: $\lambda = \max\left\{0.01, \min\left\{0.5 - \log\left(\frac{i+1}{25}\right)\right\}\right\}$ for episode i
- Discount factor: $\gamma = 0.99$



(a) Reward plot of "learn to balance" experiment



(b) Reward plot of "learn to adapt" experiment

Figure 4: Reward over episodes plot of (a) learn to balance experiment and (b) learn to adapt experiment.

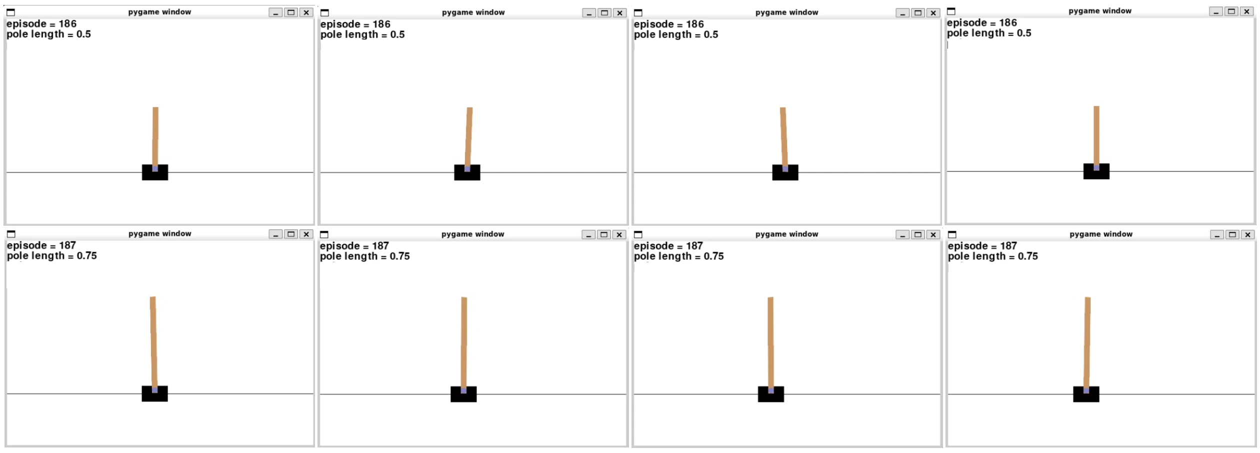


Figure 5: Illustration of "learn to adapt" experiment

4.1 Learn to balance

First, we show that our algorithm can make the cart learn to balance the pole. We let the algorithm to run 200 episodes, the reward over episodes is plotted in Figure 4a. As shown in the figure, the reward gradually increases with iterations. After about the 160th iteration, the reward can reach the upper limit we preset (475), which indicating the agent learned how to balance via the proposed algorithm. Video demonstration of the experiment can be found at CSC3180-Cart-Pole/Videos/200_iterations.mp4.

4.2 Learn to adapt to varying weights and CoMs

Next, we show that our algorithm can make the cart adapts to varying pole. This time we still run 200 episodes, but when the agent reached the maximum reward five times in a row, we changed the length of the pole (the linear density of the pole is constant, so as the length is changed, the weight and height of the center of gravity of the pole change), as shown in Figure 5. The reward over episodes is plotted in Figure 4b. The reward earned does not decrease after pole change, which indicating the agent learned how to adapt to varying weights and CoMs via the proposed algorithm. Video demonstration of this experiment can be found at CSC3180-Cart-Pole/Videos/200_iterations_pole_changed_187.mp4.

5 Resources

Here we list the resources that we used to complete the project.

5.1 Project Repository

All source codes of this project are hosted on a public repository CSC3180-Cart-Pole on GitHub.

5.2 Modified "cart-pole" Environment

In this project, we customized the OpenAI Gym environment for the cart-pole problem to create a more challenging scenario for our RL agent. We adjusted the cart's mass, pole length, and gravitational constant to create a more volatile environment that our agent could learn to balance the pole in. The customized environment is "csc3180_cartpole_env". You can find the source code at "CSC3180-Cart-Pole/src/csc3180_cartpole_env".

Our aim was to train an agent using an RL algorithm that could maintain balance for as long as possible in this customized environment. As part of our project, we customized the environment and designed specific APIs for the RL simulation program, such as "change_poleLength", "change_poleMass" and "set_i_episode". These APIs allowed the RL simulation program could effectively train the agent in the customized environment and improve performance in the specific task we designed.

5.2.1 How to run our code with the customized Environment

5.2.1.1 Create a Virtual-Env We highly recommend you create a virtual environment to test our project. The Virtual environment will isolate project dependencies and package installations, allowing for effective version control of packages and avoiding conflicts with other Python projects or the system environment. However, the virtual environment is not mandatory and you can skip it.

We recommend you to install the packet in the virtual env

```
pip install virtualenv virtualenvwrapper
```

Listing 1: Installation of virtualenv

Then, you first need to add some lines to your " /.bashrc" profile. Open the file using nano , vim , or emacs and append these lines to the end:

```
export WORKON_HOME=$HOME/.local/bin/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
export VIRTUALENVWRAPPER_VIRTUALENV=$HOME/.local/bin/virtualenv
source $HOME/.local/bin/virtualenvwrapper.sh
```

Listing 2: Setup of Environment Path

Save the file. Then “source it” in your terminal:

```
source ~/.bashrc
```

Listing 3: Setup of Environment Path

Create a Python 3 virtual environment for csc3180

```
mkvirtualenv csc3180 -p python3

workon csc3180
```

Listing 4: Create a Virtual-Env

5.2.1.2 Installation by setup.py

You can install our customized packet into your virtual environment by executing setup.py

```
cd ./CSC3180-Cart-Pole/src/csc3180_cartpole_env

python3 -m pip install .
```

Listing 5: To use setup.py to install our env packet

Once the installation process is complete, ensure that the customized environment has been installed successfully by using the "pip list" command. This will provide a comprehensive list of all installed Python packages, including "csc3180_cartpole_env".

5.2.1.3 Run our Q-Leaning Program

You can use the API from our customized gym environment in your Python code as below:

```
import csc3180_cartpole_env
import gymnasium as gym

env = gym.make('csc3180/CartPole_v1', render_mode='human')
```

Listing 6: use the customized environment in python

Moreover, we have successfully developed an RL algorithm that is tailored to our specific gym environment. To run the algorithm, please execute the following code, which will enable our algorithm to interact with the customized environment and begin the training process.

```
cd ./CSC3180-Cart-Pole/src

python qlearning.py
```

Listing 7: To use setup.py to install our env packet

5.3 Project Videos

In addition to the project report, we hosted 5 video demonstrations on OneDrive, available through the following link.

6 Group Information

Student Name	**Student ID**	**Email**
Zixing Jiang	119010130	119010130@link.cuhk.edu.cn
Xuanyang Xu	119010361	119010361@link.cuhk.edu.cn
Muhan Lin	119010177	119010177@link.cuhk.edu.cn
Hongyi Yang	119010379	119010379@link.cuhk.edu.cn

Table 2: Group Information

Declaration of Contributions

- *Project topic and proposal:* Zixing Jiang, Xuanyang Xu, Muhan Lin, Hongyi Yang
- *Reinforcement learning environment:* Xuanyang Xu, Muhan Lin
- *Epsilon-Greedy Q-Learning algorithm:* Zixing Jiang
- *Experiments:* Zixing Jiang, Xuanyang Xu
- *Project presentation video:* Hongyi Yang
- *Project report:* Zixing Jiang, Xuanyang Xu, Muhan Lin