# Egg Drop Game and Agent Documentation

**Modifying the Environment**
I chose to modify the lunar lander environment from the OpenAI Gymnasium library in Python. I modified the game's colors and the shape of the lander to give it an egg-drop theme. This involved creating a custom environment and registering it with the Gymnasium library.

The custom environment inherits from the original lunar lander class and overrides the reset() and render() functions. The new render function sets the color of the lander legs to red to look like popsicle sticks which might be used in an egg drop challenge. The center of the lander is also changed to an egg color, the background color is changed to blue (sky), and the ground color is changed to green (grass). Since the color values for the sky and ground were hard-coded into the original render function I copied the old function and changed the color values. The reset function was modified to change the lander shape to an egg shape. To do this, I copied the original reset function and changed the value of LANDER_POLY before the function ran.

ChatGPT was helpful when changing the lander shape since it was able to write the new value for LANDER_POLY. This meant I didn't need to determine which points would make an egg shape.

**Environment Details**
When playing the game several rewards and penalties can be earned.

| Rewards | Penalties |
|---|---|
| ● Moving closer to the landing pad<br>● Moving slowly<br>● 10 points for each leg touching the ground<br>● 100 points for a safe landing | ● Moving away from the landing pad<br>● Moving too quickly<br>● Not staying Horizontal<br>● -0.03 for using side wind<br>● -0.3 for using upward wind<br>● -100 for a crash landing |

Episodes with 200 or more points are considered successful.

The observation space for the environment (what the agent can see) includes the egg's x and y coordinates, x and y velocities, angle, angular velocity, and two booleans indicating whether the legs are touching the ground.

The action space for the environment (what the agent can do) includes nothing, right wind, left wind, and upward wind.

**Q Learning Agent**
The Q learning agent can be found in the Q_agent.py file. It includes functions to get an action from the agent, update the policy weights of the agent, save the agent's policy to a file, or load the agent's policy from a file.

The agent works by taking the observation space for different steps and storing these states as keys in a Python dictionary. Each state maps to a list of weights corresponding to the expected reward/penalty when taking a certain action from that state. Since a very large amount of states can be created by the observation space the values are rounded to the nearest whole number before being added to the dictionary.

Rounded observation space → [Nothing, Left, Upward, Right]
Nothing = expected reward/penalty for doing nothing
Left = expected reward/penalty for left wind
Upward = expected reward/penalty for upward wind
Right = expected reward/penalty for right wind

The agent's training loop can be found in the wrapper.py file in the trainModel_Q() function.

**PPO Agent**
The PPO agent can be found in the PPO_agent.py file. It contains two neural networks. One is the actor and the other is a critic. The actor chooses which move to take while the critic evaluates the move. These neural networks work to train each other on batches of steps in the environment.

Training is done in the PPO_trainer.py file with the run() function. The run function uses the simulate_steps() function to collect step data on the environment. Then the generalized advantage estimation is computed using the compute_gae() function. After this, the neural networks are updated using the advantages in optimize_minibatch().

The PPO agent I used was modified from Costa Huang's tutorial. It was difficult to understand and I had to watch the video several times.
Video: https://youtu.be/MEt6rrxH8W4
Code: https://github.com/vwxyzjn/ppo-implementation-details/blob/main/ppo.py

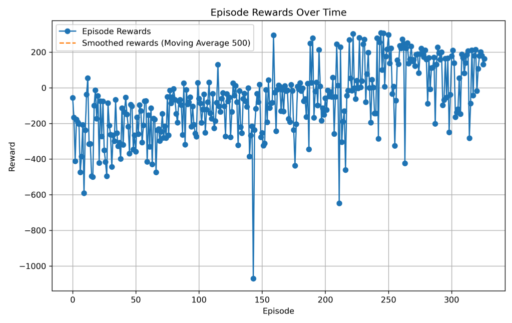
CodeEmporium's video helped me understand the idea behind the PPO algorithm.
Video: https://youtu.be/MVXdncpCbYE

Nicholas Renotte's video shows how to use the stable baselines 3 library in Python for PPO and A2C.
Video: https://youtu.be/Mut_u40Sqz4

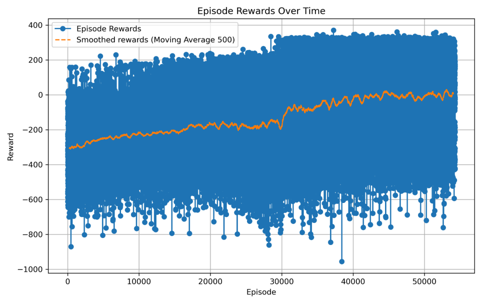
**Testing the agents**
There are several ways to see the agent's progress. While training, some videos of the agent's training episodes are recorded. Each agent also records interesting values while training including the reward for each episode, the steps taken in each episode, and the losses used to update the models. These values are graphed when the agent finishes training and stored in a plots folder.

**Results**

**PPO 200,000 training steps**



**Q learning 10 million training steps**



After 200k steps the PPO algorithm already outperforms the Q learning agent trained for 10 million steps. I believe this may be because the Q-learning algorithm can only learn about one state action pair for each step while the PPO algorithm updates its neural networks which can be applied to any state.

The saved Q learning agen (557 KB) t is much larger than the saved PPO agent (43 KB). These agents were not trained for the same amount of time however the PPO saved agent is likely as large as it will get since each step value is used to update weights of the neural nets. Q learning creates 4 new weights (action space size) for each new state it finds.

**ChatGPT use**
I used chatGPT in a few different ways while working on this project. It helped me understand the PPO algorithm since I am unfamiliar with pytorch (used in the tutorial) and I could ask it what parts of the tutorial code did. ChatGPT also created a menu for the project that can be used to train or run the agents.