AI Agent for Lunar Lander

Prabhjot Singh Rai [prabhjot], Abhishek Bharani [abharani], Amey Naik [ameynaik]

1 Task Definition

1.1 Scope

The task accomplished by this project is to build an AI agent which would land with constant high rewards on a landing pad, defined by Box2D Lunar Lander available on OpenAI gym. This is accomplished by Reinforcement Learning, particularly by applying different Q-learning techniques. This project has explored Full DQN, Double DQN and Dueling Network Architecture (Dueling DQN), their performances in "solving" the game.

1.2 Evaluation

We have considered a game to be solved when the agent starts getting average reward of 200 over 100 consecutive episodes. Performances have been compared with baselines and oracle. Moreover, we have also tried to get comparable results with the leader-board on OpenAI gym.

2 Infrastructure

We have used OpenAI library as our infrastructure. Although some insights are provided in Box2D Lunar Lander on OpenAI website, but thorough exploration of actions, state space, environment etc. were done before starting to solve the problem. Following is the description:

2.1 Actions

In this game, four discrete actions are available to the playing agent at any time frame:

(a) Do nothing

(b) Fire left orientation engine (rotates the lunar lander clockwise)

(c) Fire main engine (provides upward thrust)

(d) Fire right orientation engine (rotates the lunar lander anti-clockwise)

The agent can choose only one action among the given actions at a given time frame.

2.2 Terrain

The terrain is a combination of 10 points, and the helipad(landing zone) is fixed between 5th and 6th points towards the center. The values of the height of the landing zone(5th and 6th points on the terrain) are viewport height divided by 4, and the rest of the points are randomly sampled between 0 to H/2 using *numpy* random and smoothened (averaging 3 continuous points).

2.3 Initial State

The state is 8 dimensional values of different parameters of the lunar lander at any given time. The starting state is randomly initialized (the lunar lander takes a step in the world through the "idle" action) with certain bounds based on the environment.

Inspecting code of open_ai lunar lander, we see that the initial states are defined by simulating the environment in one frame (calling box world simulation using time step as $1/FPS$). Using this simulation, elaborating the initial state as follows:

1. Position X (Initial Position X: final x which changed from half of viewport width to a value after taking "idle" action before the simulation)

2. Position Y (Initial Position Y: final y which changed from from the viewport height to a value after taking "idle" action before simulation)

3. Velocity X (Initial Velocity X: final velocity x changed from 0 to a value after taking "idle" action before simulation)

4. Velocity Y (Initial Velocity Y: final velocity y which changed from 0 to a value after taking "idle" action before simulation)

5. Current lander angle (Initial lander angle: final lander angle after simulation on "idle" action from 0 degrees)

6. Angular velocity (Initial angular velocity: final angular velocity after simulation on "idle" action from 0 angular velocity)

7. Left leg contacted the surface (Initial value: False, since there's is no probability that the lunar lander's leg will touch the moon surface when at the top)

8. Right leg contacted the surface (Initial value: False)

2.4 End State

The episode ends in the following scenarios:

1. When the lunar lander goes outside of the viewport bounds, the game is over with -100 is negative reward.

2. When the lunar lander touches the ground with a high velocity

3. When the lunar lander touches the ground with body part except the legs

4. When the lunar lander stabilises on the moon's surface (change in shape of lunar lander is constantly 0 for a number of frames)

2.5 Rewards and Transitions

Before defining the rewards, let's define the shape of the lunar lander which decides the rewards. The shape of the lunar lander is a function of position coordinates (x, y), linear velocities (v_x, v_y), lander angle θ and contact of both the lander legs. We are interested in finding the change of shape at every step for the lunar lander to calculate the rewards for each given action. Shape change is given by subtraction of previous shape and current shape. Formally, shaping at time frame t:

$$\text{shaping}_t = -100 * (x^2 + y^2)$$
$$- 100 * (v_x^2 + v_y^2)$$
$$- 100 * abs(\theta) + 10 * (\text{Left leg contacted}) + 10 * (\text{Right leg contacted})$$
$$\text{shape change} = \text{shaping}_t - \text{shaping}_{t-1}$$

The rewards are defined as follows:

1. If the lunar lander crashes, or goes out of the bounds: -100

2. If the lunar lander is not awake anymore (stabilises at 0 shape change): $+100$

3. Doing nothing: shape change

4. Firing the engine: shape change - 0.3

5. Rotating: shape change - 0.03

The total reward will automatically be a sum of all the rewards at each time frame, and if the lander touches the ground with it's legs, will add those rewards to the total rewards earned during an episode. Transition probabilities are unknown, we get next states by simulating the lunar lander in the box environment given the current state and action taken.

Note: The transition probabilities and rewards are unknown to our agent, which it will try to figure out through exploration and incorporate in learning.
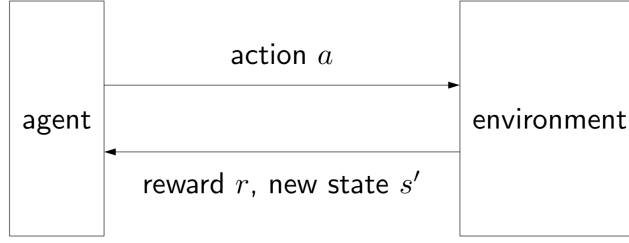
Figure 1: Reinforcement Learning framework

3 Approach

3.1 Challenges

We dedicated time to understand how the openai-gym is working internally. For this specific problem, we understood how the rewards and environment is behaving on every episode, as described in infrastructure. Moreover, to get faster results, we focused on better machine and GPU settings on google cloud and getting docker environment setup for codalab and reproducibility across platforms.

3.2 Modelling the problem

Basic framework is defined in Fig 1. The agent performs action(a) on the environment. Performing the action on the environment returns reward(r) and new state s'. Our agent then incorporates this information.

In such a model, for our problem, here is quick description of notations:

1. S is defined as all possible states, s is one particular 8 dimensional state

2. A is defined as all possible actions, a is one of the actions out of [idle, fire engine, rotate left, rotate right

3. R is the reward distribution given (s, a)

4. P is the set of possible transitions and their probabilities given (s, a)

5. γ: the discount factor. How much we want our agent to discount future reward. It is a hyperparameter that we define ourselves.

Our model builds off of Q-learning algorithms by using a Deep Neural Network (DNN) for approximating the state-action Q-value, Q(s, a). Given the state s, our goal is to identify a policy π_{opt} that maps the states to actions in order to maximize the total reward we get.

$$S \rightarrow A \text{ based on } \pi_{opt}$$

As we know, Q value is defined as the expected reward that we get following action a in a given state s and then following the policy π, our objective is to define a $Q_{opt}(s, a)$, which can be maximised over all possible actions at a state s, in order to find the π_{opt}.

// describe oracle

// describe basic framework for rl learning

// describe all three learning algorithms

4 Results

// show all the results here, explain something about them

5 Error Analysis