Deep Q-Learning -Lunar Lander

Train an agent to land a lunar lander safely on a landing pad on the surface of the moon.

Table of Contents

Lunar Lander Environment

we will be using **OpenAI's Gym Library.** The Gym library provides a wide variety of environments for reinforcement learning. To put it simply, an environment represents a problem or task to be solved. In this notebook, we will try to solve the Lunar Lander environment using reinforcement learning.

The goal of the Lunar Lander environment is to land the lunar lander safely on the landing pad on the surface of the moon. The landing pad is designated by two flag poles and it is always at coordinates (0,0) but the lander is also allowed to land outside of the landing pad. The lander starts at the top center of the environment with a random initial force applied to its center of mass and has infinite fuel. The environment is considered solved if you get 200 points.

Lunar Lander Environment.

2.1 Action Space

The agent has four discrete actions available:

Do nothing.

Fire main engine.

Fire right engine.

Fire left engine.

Each action has a corresponding numerical value:

Do nothing = 0

Fire main engine = 2

Fire right engine = 1

Fire left engine = 3

2.2 Rewards

The Lunar Lander environment has the following reward system:

Landing on the landing pad and coming to rest is about 100-140 points.

If the lander moves away from the landing pad, it loses reward.

If the lander crashes, it receives -100 points.

If the lander comes to rest, it receives +100 points.

Each leg with ground contact is +10 points.

Firing the main engine is -0.3 points each frame.

Firing the side engine is -0.03 points each frame.

2.3 Episode Termination

An EPISODE ENDS (i.e the environment enters a terminal state) if:

The lunar lander crashes (i.e if the body of the lunar lander comes in contact with the surface of the moon).

The lander's x-coordinate is greater than 1.

Deep Q-Learning

In the Deep Q-Learning, we solve this problem by using a neural network to estimate the action-value function **$Q(s,a) = Q^*(s,a)$.**
We call this neural network a Q-Network and it can be trained by adjusting its weights at each iteration to minimize the mean-squared error in the Bellman equation.

Unfortunately, using neural networks in reinforcement learning to estimate action-value functions has proven to be highly unstable. Luckily, there's a couple of techniques that can be employed to avoid instabilities. These techniques consist of using a *Target Network* and *Experience Replay*.

Experience Replay

When an agent interacts with the environment, the states, actions, and rewards the agent experiences are sequential by nature. If the agent tries to learn from these consecutive experiences it can run into problems due to the strong correlations between them. To avoid this, we employ a technique known as Experience Replay to generate uncorrelated experiences for training our agent.

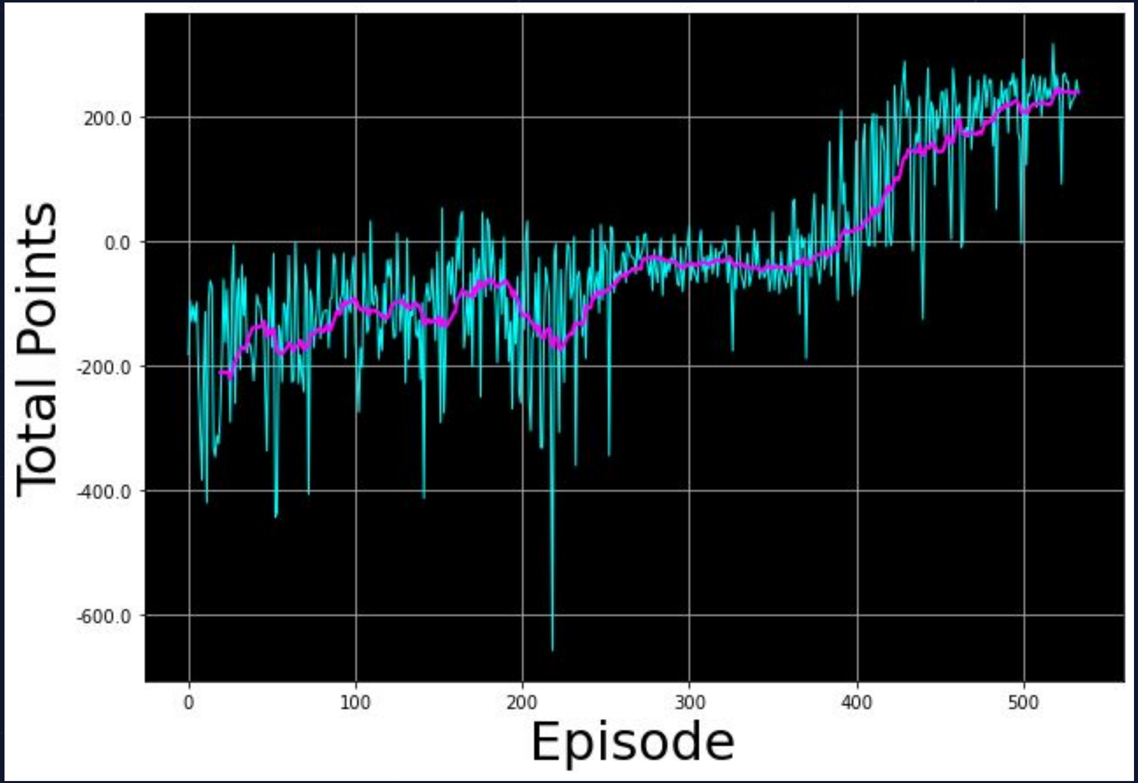
Experience replay consists of storing the agent's experiences (i.e the states, actions, and rewards the agent receives) in a memory buffer and then sampling a random mini-batch of experiences from the buffer to do the learning. The experience tuples will be (S_t, A_t, R_t, S_{t+1}) added to the memory buffer at each time step as the agent interacts with the environment.

Deep Q-Learning Algorithm with Experience Replay

Algorithm 1: Deep Q-Learning with Experience Replay

Initialize memory buffer D with capacity N
Initialize Q-Network with random weights w
Initialize target $\hat{Q}$-Network with weights $w^- = w$
for episode $i = 1$ ***to*** M **do**
    Receive initial observation state S_1
    for $t = 1$ ***to*** T **do**
        Observe state S_t and choose action A_t using an ϵ-greedy policy
        Take action A_t in the environment, receive reward R_t and next state S_{t+1}
        Store experience tuple (S_t, A_t, R_t, S_{t+1}) in memory buffer D
        Every C steps perform a learning update:
        Sample random mini-batch of experience tuples (S_j, A_j, R_j, S_{j+1}) from D
        Set $y_j = R_j$ if episode terminates at step $j+1$, otherwise set $y_i = R_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a')$
        Perform a gradient descent step on $(y_j - Q(s_j, a_j; w))^2$ with respect to the Q-Network weights w
        Update the weights of the $\hat{Q}$-Network using a soft update
    end
end

Plot to display how AGENT improved during TRAINING.



Thank you

—Radha Krishna Garg

GITHUB REPO LINK:

https://github.com/sinisterdaddy/LUNAR-LANDER