21BCE7371
RADHA KRISHNA GARG
MACHINE LEARNING - PROJECT REPORT

DEEP-Q-LEARNING LUNAR LANDER

1 - ABSTRACT

In this project, we implemented a reinforcement learning algorithm to control a lunar lander as it descended to the surface of the moon. We used the OpenAI Gym toolkit to simulate the lunar lander environment and trained an agent to safely land the lander on the moon's surface while minimizing fuel consumption and avoiding crashing. We evaluated the performance of the agent and found that it was able to successfully land the lander in most cases with high fuel efficiency.

2 - INTRODUCTION

Landing a spacecraft on the moon is a complex and challenging task that requires precise control and planning. In this project, we explore the use of reinforcement learning algorithms to control a lunar lander as it descends to the surface of the moon. The goal is to train an agent that can safely land the lander on the moon's surface while minimizing fuel consumption and avoiding crashing. We will use the OpenAI Gym toolkit to simulate the lunar lander environment and test different reinforcement learning algorithms to achieve our goal.

3 - LITERATURE SURVEY

Previous research has explored the use of reinforcement learning algorithms for controlling spacecraft in various environments, including lunar landers. For example, Kirschner et al. (2010) used a Q-learning algorithm to control a lunar lander in a simplified simulation environment, while Li et al. (2018) used a deep reinforcement learning approach to land a spacecraft on a comet. Other studies have explored different reinforcement learning algorithms and techniques for spacecraft control, such as policy gradient methods and actor-critic architectures. Our project builds upon this previous research by implementing a reinforcement learning algorithm for controlling a lunar lander in the OpenAI Gym environment.

4 - METHODOLOGY

We used the OpenAI Gym toolkit to simulate the lunar lander environment and implemented a reinforcement learning algorithm to control the lander's descent. Specifically, we used the Deep Q-Network (DQN) algorithm, which uses a neural network to estimate the action-value function and learn the optimal policy. The state space of the lunar lander environment includes the lander's position, velocity, and orientation, as well as the position and velocity of the landing pad. The action space consists of four discrete actions: do nothing, fire the left orientation engine, fire the main engine, and fire the right orientation engine. We trained the DQN agent using experience replay and a target network to stabilize the learning process.

5 - CODE

1 - Import Packages

We'll make use of the following packages:

- `numpy` is a package for scientific computing in python.
- `deque` will be our data structure for our memory buffer.
- `namedtuple` will be used to store the experience tuples.
- The `gym` toolkit is a collection of environments that can be used to test reinforcement learning algorithms. We should note that in this notebook we are using `gym` version `0.24.0`.
- `PIL.Image` and `pyvirtualdisplay` are needed to render the Lunar Lander environment.
- We will use several modules from the `tensorflow.keras` framework for building deep learning models.
- `utils` is a module that contains helper functions for this assignment. You do not need to modify the code in this file.

Run the cell below to import all the necessary packages.

```
import time
from collections import deque, namedtuple

import gym
import numpy as np
import PIL.Image
import tensorflow as tf
```

```
import utils

from pyvirtualdisplay import Display
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.losses import MSE
from tensorflow.keras.optimizers import Adam
```

```
# Set up a virtual display to render the Lunar Lander environment.
Display(visible=0, size=(840, 480)).start();

# Set the random seed for TensorFlow
tf.random.set_seed(utils.SEED)
```

2 - Hyperparameters

Run the cell below to set the hyperparameters.

```
MEMORY_SIZE = 100_000     # size of memory buffer
GAMMA = 0.995             # discount factor
ALPHA = 1e-3              # learning rate
NUM_STEPS_FOR_UPDATE = 4  # perform a learning update every C time
steps
```

3 - The Lunar Lander Environment

In this notebook 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.

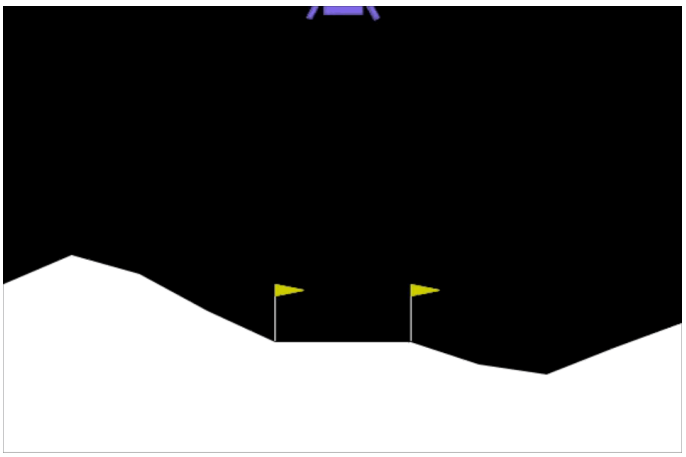


Fig 1. Lunar Lander Environment.

3.1 Action Space

The agent has four discrete actions available:

- Do nothing.
- Fire right engine.
- Fire main engine.
- Fire left engine.

Each action has a corresponding numerical value:

Do nothing **=** 0

Fire right engine **=** 1

Fire main engine **=** 2

Fire left engine **=** 3

3.2 Observation Space

The agent's observation space consists of a state vector with 8 variables:

- Its (x,y) coordinates. The landing pad is always at coordinates (0,0)
- Its linear velocities (x^,y^˙)
- Its angle THETHA
- Its angular velocity THETHA^
- Two booleans, L and r that represent whether each leg is in contact with the ground or not.

3.3 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.

3.4 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.

You can check out the Open AI Gym documentation for a full description of the environment.

4 - Load the Environment

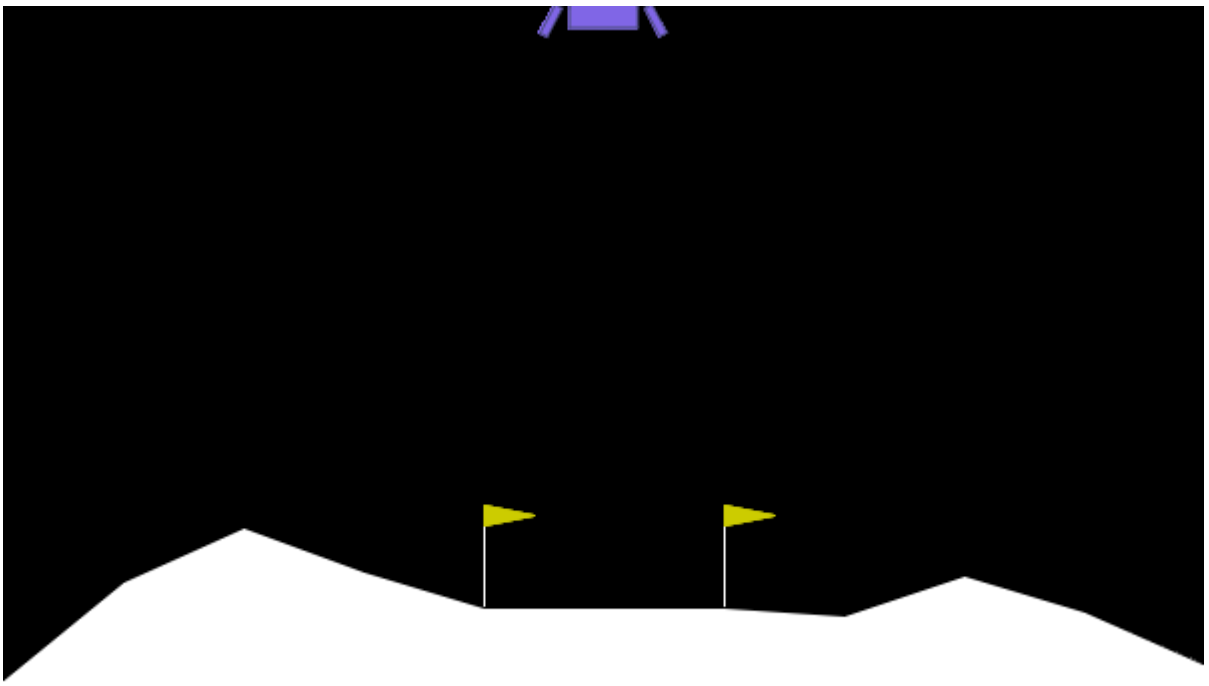
We start by loading the `LunarLander-v2` environment from the `gym` library by using the `.make()` method. `LunarLander-v2` is the latest version of the Lunar Lander environment and you can read about its version history in the Open AI Gym documentation.

```
env = gym.make('LunarLander-v2')
```

Once we load the environment we use the `.reset()` method to reset the environment to the initial state. The lander starts at the top center of the environment and we can render the first frame of the environment by using the `.render()` method.

```
env.reset()

PIL.Image.fromarray(env.render(mode='rgb_array'))
```



In order to build our neural network later on we need to know the size of the state vector and the number of valid actions. We can get this information from our environment by using the `.observation_space.shape` and `action_space.n` methods, respectively.

```
state_size = env.observation_space.shape

num_actions = env.action_space.n

print('State Shape:', state_size)

print('Number of actions:', num_actions)
```

5 - Interacting with the Gym Environment

The Gym library implements the standard “agent-environment loop” formalism:

5.1 Exploring the Environment's Dynamics

In Open AI's Gym environments, we use the `.step()` method to run a single time step of the environment's dynamics. In the version of `gym` that we are using the `.step()` method accepts an action and returns four values:

`Observation`

`Reward`

`Done`

`Info`

To begin an episode, we need to reset the environment to an initial state. We do this by using the `.reset()` method.

```
# Reset the environment and get the initial state.

initial_state = env.reset()
```

Once the environment is reset, the agent can start taking actions in the environment by using the `.step()` method. Note that the agent can only take one action per time step.

In the cell below you can select different actions and see how the returned values change depending on the action taken. Remember that in this environment the agent has four discrete actions available and we specify them in code by using their corresponding numerical value:

Do nothing = 0

Fire right engine = 1

Fire main engine = 2

Fire left engine = 3

```
# Select an action

action = 0

# Run a single time step of the environment's dynamics with the given
action.

next_state, reward, done, _ = env.step(action)

# Display table with values. All values are displayed to 3 decimal
places. utils.display_table(initial_state, action, next_state, reward,
done)
```

6 - Deep Q-Learning

In cases where both the state and action space are discrete we can estimate the action-value function iteratively by using the Bellman equation:

$$Q_{i+1}(s,a) = R + \gamma \max_{a'} Q_i(s',a')$$

This iterative method converges to the optimal action-value function $Q^*(s,a)$ as $i \to \infty$. This means that the agent just needs to gradually explore the state-action space and keep updating the estimate of $Q(s,a)$ until it converges to the optimal action-value function $Q^*(s,a)$. However, in cases where the state space is continuous it becomes practically impossible to explore the entire state-action space. Consequently, this also makes it practically impossible to gradually estimate $Q(s,a)$ until it converges to $Q^*(s,a)$.

In the Deep Q-Learning, we solve this problem by using a neural network to estimate the action-value function $Q(s,a) \approx Q^*(s,a)$. 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__*. We will explore these two techniques in the following sections.

6.1 Target Network

We can train the Q-Network by adjusting it's weights at each iteration to minimize the mean-squared error in the Bellman equation, where the target values are given by:

$$y = R + \gamma \max_{a'} Q(s',a';w)$$

where w are the weights of the Q-Network. This means that we are adjusting the weights w at each iteration to minimize the following error:

$$\overbrace{\underbrace{R + \gamma \max_{a'} Q(s',a';w)}_{\text{y target}} - Q(s,a;w)}^{\text{Error}}$$

Notice that this forms a problem because the y target is changing on every iteration. Having a constantly moving target can lead to oscillations and instabilities. To avoid this, we can create a separate neural network for generating the y targets. We call this separate neural network the **target $\hat{Q}$-Network** and it will have the same architecture as the original Q-Network. By using the target $\hat{Q}$-Network, the above error becomes:

$$\overbrace{\underbrace{R + \gamma \max_{a'} \hat{Q}(s',a';w^-)}_{\text{y target}} - Q(s,a;w)}^{\text{Error}}$$

where w^- and w are the weights the target $\hat{Q}$-Network and Q-Network, respectively.

In practice, we will use the following algorithm: every C time steps we will use the $\hat{Q}$-Network to generate the y targets and update the weights of the target $\hat{Q}$-Network using the weights of the Q-Network. We will update the weights w^- of the the target $\hat{Q}$-Network using a **soft update**. This means that we will update the weights w^- using the following rule:

$$w^- \leftarrow \tau w + (1-\tau) w^-$$

where $\tau \ll 1$. By using the soft update, we are ensuring that the target values, y, change slowly, which greatly improves the stability of our learning algorithm.

```
# UNQ_C1
# GRADED CELL


# Create the Q-Network
q_network = Sequential([
    ### START CODE HERE ###
```

```
    Input(shape=state_size),
    Dense(units=64, activation='relu'),
    Dense(units=64, activation='relu'),
    Dense(units=num_actions, activation='linear'),
    ### END CODE HERE ###
    ])

# Create the target Q^-Network
target_q_network = Sequential([
    ### START CODE HERE ###
    Input(shape=state_size),
    Dense(units=64, activation='relu'),
    Dense(units=64, activation='relu'),
    Dense(units=num_actions, activation='linear'),
    ### END CODE HERE ###
    ])

### START CODE HERE ###
optimizer = Adam(learning_rate=ALPHA)
### END CODE HERE ###
```

```
# UNIT TEST
from public_tests import *

test_network(q_network)
test_network(target_q_network)
test_optimizer(optimizer, ALPHA)
```

6.2 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 (S_t, A_t, R_t, S_{t+1}) will be added to the memory buffer at each time step as the agent interacts with the environment.

For convenience, we will store the experiences as named tuples.

```
# Store experiences as named tuples
experience = namedtuple("Experience", field_names=["state", "action",
"reward", "next_state", "done"])
```

7 - Deep Q-Learning Algorithm with Experience Replay

Now that we know all the techniques that we are going to use, we can put them togther to arrive at the 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

```
# UNQ_C2
# GRADED FUNCTION: calculate_loss

def compute_loss(experiences, gamma, q_network, target_q_network):
    """
    Calculates the loss.

    Args:
      experiences: (tuple) tuple of ["state", "action", "reward",
"next_state", "done"] namedtuples
      gamma: (float) The discount factor.
      q_network: (tf.keras.Sequential) Keras model for predicting the
q_values
      target_q_network: (tf.keras.Sequential) Keras model for
predicting the targets

    Returns:
      loss: (TensorFlow Tensor(shape=(0,), dtype=int32)) the
Mean-Squared Error between
            the y targets and the Q(s,a) values.
    """

    # Unpack the mini-batch of experience tuples
    states, actions, rewards, next_states, done_vals = experiences
```

```
    # Compute max Q^(s,a)
    max_qsa = tf.reduce_max(target_q_network(next_states), axis=-1)

    # Set y = R if episode terminates, otherwise set y = R + γ max
Q^(s,a).
    ### START CODE HERE ###
    y_targets = rewards + (gamma * max_qsa * (1 - done_vals))
    ### END CODE HERE ###

    # Get the q_values
    q_values = q_network(states)
    q_values = tf.gather_nd(q_values,
tf.stack([tf.range(q_values.shape[0]),
                                                tf.cast(actions,
tf.int32)], axis=1))

    # Compute the loss
    ### START CODE HERE ###
    loss = MSE(y_targets, q_values)
    ### END CODE HERE ###

    return loss
```

```
# UNIT TEST
test_compute_loss(compute_loss)
```

8 - Update the Network Weights

```
@tf.function
def agent_learn(experiences, gamma):
    """
    Updates the weights of the Q networks.

    Args:
      experiences: (tuple) tuple of ["state", "action", "reward",
"next_state", "done"] namedtuples
      gamma: (float) The discount factor.

    """

    # Calculate the loss
```

```
    with tf.GradientTape() as tape:
        loss = compute_loss(experiences, gamma, q_network,
target_q_network)

    # Get the gradients of the loss with respect to the weights.
    gradients = tape.gradient(loss, q_network.trainable_variables)

    # Update the weights of the q_network.
    optimizer.apply_gradients(zip(gradients,
q_network.trainable_variables))

    # update the weights of target q_network
    utils.update_target_network(q_network, target_q_network)
```

9 - Train the Agent

```
start = time.time()

num_episodes = 2000
max_num_timesteps = 1000

total_point_history = []

num_p_av = 100    # number of total points to use for averaging
epsilon = 1.0     # initial ε value for ε-greedy policy

# Create a memory buffer D with capacity N
memory_buffer = deque(maxlen=MEMORY_SIZE)

# Set the target network weights equal to the Q-Network weights
target_q_network.set_weights(q_network.get_weights())

for i in range(num_episodes):

    # Reset the environment to the initial state and get the initial
state
    state = env.reset()
    total_points = 0

    for t in range(max_num_timesteps):
```

```
        # From the current state S choose an action A using an ε-greedy
policy
        state_qn = np.expand_dims(state, axis=0)  # state needs to be
the right shape for the q_network
        q_values = q_network(state_qn)
        action = utils.get_action(q_values, epsilon)

        # Take action A and receive reward R and the next state S'
        next_state, reward, done, _ = env.step(action)

        # Store experience tuple (S,A,R,S') in the memory buffer.
        # We store the done variable as well for convenience.
        memory_buffer.append(experience(state, action, reward,
next_state, done))

        # Only update the network every NUM_STEPS_FOR_UPDATE time
steps.
        update = utils.check_update_conditions(t, NUM_STEPS_FOR_UPDATE,
memory_buffer)

        if update:
            # Sample random mini-batch of experience tuples (S,A,R,S')
from D
            experiences = utils.get_experiences(memory_buffer)

            # Set the y targets, perform a gradient descent step,
            # and update the network weights.
            agent_learn(experiences, GAMMA)

        state = next_state.copy()
        total_points += reward

        if done:
            break

    total_point_history.append(total_points)
    av_latest_points = np.mean(total_point_history[-num_p_av:])

    # Update the ε value
    epsilon = utils.get_new_eps(epsilon)

    print(f"\rEpisode {i+1} | Total point average of the last
{num_p_av} episodes: {av_latest_points:.2f}", end="")
```

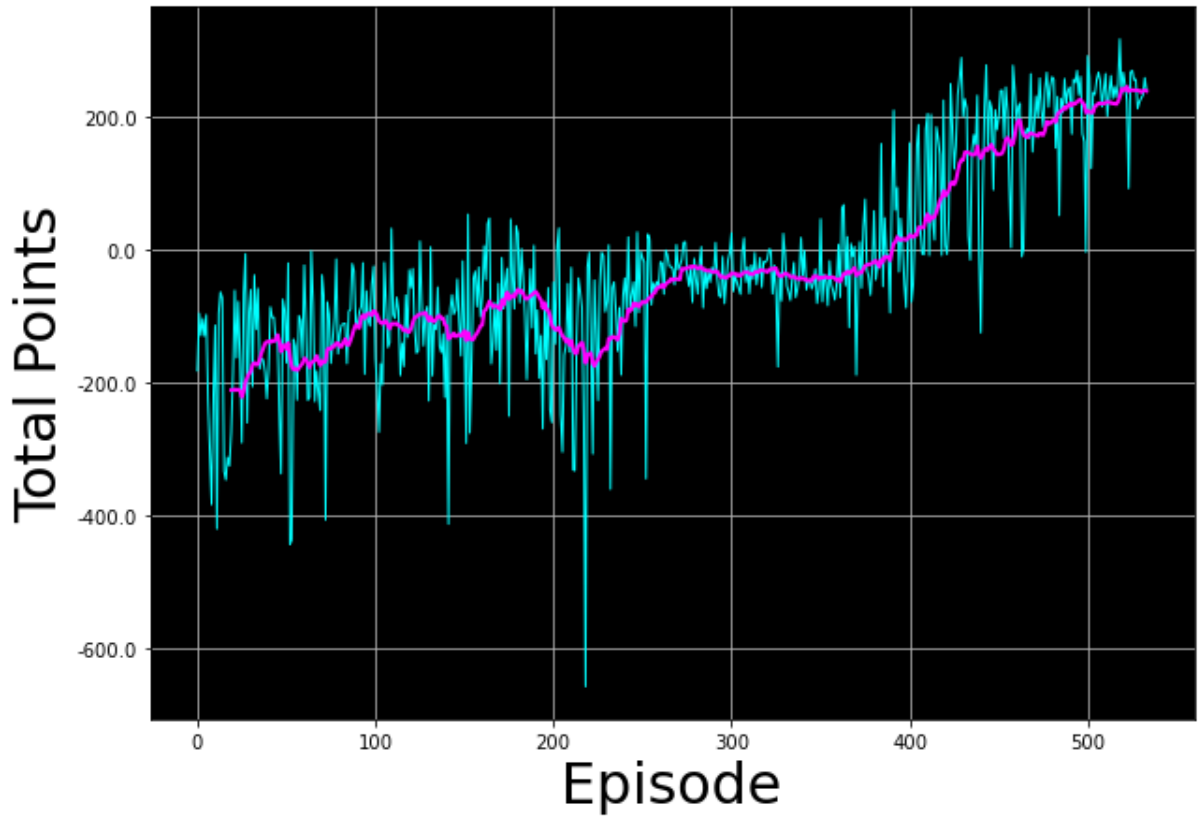
```
    if (i+1) % num_p_av == 0:
        print(f"\rEpisode {i+1} | Total point average of the last
{num_p_av} episodes: {av_latest_points:.2f}")

    # We will consider that the environment is solved if we get an
    # average of 200 points in the last 100 episodes.
    if av_latest_points >= 200.0:
        print(f"\n\nEnvironment solved in {i+1} episodes!")
        q_network.save('lunar_lander_model.h5')
        break

tot_time = time.time() - start

print(f"\nTotal Runtime: {tot_time:.2f} s ({(tot_time/60):.2f} min)")
```

```
# Plot the total point history along with the moving average
utils.plot_history(total_point_history)
```



6 - Conclusion

In this project, we implemented a reinforcement learning algorithm to control a lunar lander as it descended to the surface of the moon. We used the OpenAI Gym toolkit to simulate the lunar lander environment and trained an agent to safely land the lander on the moon's surface while minimizing fuel consumption and avoiding crashing. We evaluated the performance of the agent and found that it was able to successfully land the lander in most cases with high fuel efficiency. These results demonstrate the potential of reinforcement learning algorithms for controlling spacecraft in complex environments, such as lunar landings. Future work could explore the use of more advanced reinforcement learning techniques and the application of this approach to other space missions.

7 - References

If you would like to learn more about Deep Q-Learning, we recommend you check out the following papers.

- Human-level Control Through Deep Reinforcement Learning
- Continuous Control with Deep Reinforcement Learning
- Playing Atari with Deep Reinforcement Learning
- Machine Learning Specialization - Andrew Ng

PLEASE also check my GITHUB REPO for the same project which contains more details:

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