CSN-382: Machine Learning
Lunar Lander – Project Report

Team Members:

Anand Chourasia

20114008

Neerav Chittora

20114062

Ujjawal Kumar Dubey

20115160

Submitted to – Prof. R. Bala Subramaniam

Problem Statement

The objective of the lunar lander task is to safely land a small spacecraft between two designated flags. The spacecraft can take one of four actions at each time step: firing the main engine, left engine, right engine, or doing nothing. If the spacecraft hits the ground at an incorrect angle or too fast, it results in a crash and a significant penalty. Additionally, a small penalty is incurred for firing the main engine. Rewards are given for approaching the desired landing position and successfully touching the ground. The task is considered successfully solved if the agent achieves an average score of 200 or higher over the last 100 episodes.

This environment presents a classic problem of optimizing the trajectory of a rocket. Based on Pontryagin's maximum principle, it is determined that the optimal action for the engine is to either fire at full throttle or turn it off completely. This is why the environment offers discrete actions, specifically engine on or off. The landing pad is always located at coordinates (0,0), which are the first two numbers in the state vector. It is possible for the rocket to land outside of the designated landing pad. Additionally, the fuel supply for the rocket is considered infinite, allowing the agent to learn to fly and successfully land on its first attempt.

Action Space:

1. do nothing
2. fire left orientation engine
3. fire main engine
4. fire right orientation engine

Observation Space:

The state in this scenario is represented as an 8-dimensional vector, which includes the coordinates of the lander in the x and y axes, its linear velocities in the x and y directions, its angle, its angular velocity, as well as two boolean values that indicate whether each leg of the lander is in contact with the ground or not.

Rewards:

After every step a reward is granted. The total reward of an episode is the sum of the rewards for all the steps within that episode.

For each step, the reward:

- is increased/decreased the closer/further the lander is to the landing pad.
- is increased/decreased the slower/faster the lander is moving.

- is decreased the more the lander is tilted (angle not horizontal).
- is increased by 10 points for each leg that is in contact with the ground.
- is decreased by 0.03 points each frame a side engine is firing.
- is decreased by 0.3 points each frame the main engine is firing.

The episode receives an additional reward of -100 or +100 points for crashing or landing safely respectively.

An episode is considered a solution if it scores at least 200 points.

Other Details:

- The episode finishes, if the lander is not awake, if the lander crashes or the lander gets out of the frame.
- The lander starts at the top center of the viewport with a random initial force applied to its center of mass.

Algorithms

1. Q Learning

First, let's provide an overview of the algorithm we will be using, which is called Q-learning. Q-learning is a type of reinforcement learning algorithm where an agent operates within an environment. At each time step, the

agent is able to observe the current state of the environment and take an action. Following the agent's action, the environment undergoes a change, and the agent is then able to observe the new state of the environment. Throughout these state transitions, the agent receives feedback in the form of rewards or

penalties. In traditional Q-learning, a Q-table is used to store and update the Q-values for all possible state-action pairs.

However, in complex environments with high-dimensional state spaces, such as images or continuous state spaces, using a Q-table becomes impractical due to the large number of possible states and actions. So, instead of traditional Q learning, we will be using another algorithm known as deep Q learning.

2. Deep-Q Learning

Deep Q-learning is a variant of the Q-learning algorithm that utilizes deep neural networks to approximate the Q-values, which represent the expected cumulative rewards for different actions in a reinforcement learning environment. The Deep Q-learning algorithm would involve training a Q-network, which is a deep neural network, to approximate the Q-values for different actions that the spacecraft can take, such as firing the main engine, left engine, right engine, or doing nothing.

The input to the Q-network would be the current state of the spacecraft, which would include its coordinates in x and y, linear velocities in x and y, angle, angular velocity, and the status of the landing legs (i.e., whether they are in contact with the ground or not). The output of the Q-network would be a set of Q-values, one for each possible action, representing the expected cumulative rewards for taking those actions

During training, the agent would interact with the lunar lander environment, taking actions based on the Q-values predicted by the Q-network, and receiving feedback in the form of rewards or penalties depending on the success or failure of the actions taken. The agent would collect experiences, including the current state, action, reward, and next state, and store them in a replay buffer. The Q-network would then be trained on batches of randomly sampled experiences from the replay buffer, using techniques such as gradient descent to minimize the difference between the predicted Q-values and the target Q-values, which are computed using the Bellman equation.

As the training progresses, the Q-network would learn to make optimal decisions on when and how to fire the spacecraft's engines to safely navigate and land the spacecraft between the designated flags. Once the agent achieves a certain level of proficiency, as determined by a predefined success criteria, it can be deployed to autonomously control the spacecraft and land it on the moon's surface in a safe and efficient manner.

Implementation

```
env = gym.make("LunarLander-v2", render_mode="rgb_array_list")


state_size = env.observation_space.shape
num_actions = env.action_space.n
```

Creating the lunar lander environment using make function of the gym module. Initiliazing state_size and num_actions.

```
q_network = Sequential([

    Input(state_size) ,
    Dense(64 , activation="relu") ,
    Dense(64 , activation="relu") ,
    Dense(num_actions  , activation="linear") ,
]) ;
target_q_network = Sequential([

    Input(state_size) ,
    Dense(64 , activation="relu") ,
    Dense(64 , activation="relu") ,
    Dense(num_actions  , activation="linear") ,

]) ;
target_q_network.set_weights(q_network.get_weights()) ;
optimizer = Adam(learning_rate=ALPHA);
```

Creating two deep neural networks using keras, each having four layers. The first is the input layer, followed by two hidden layers with relu as activation function, the last is the output layer which has a linear activation function. The first neural network is the policy network, the second is the target network used to estimate

the cost function without introducing instability in the optimization process.

```
def compute_cost(expriences , gamma , q_network , target_q_network) :

    states , actions , rewards ,  next_states , done_vals = expriences
    max_qsa = tf.reduce_max(target_q_network(next_states) , axis=-1)
    y_targets = rewards + gamma * ( 1 - done_vals ) * max_qsa
    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))
    loss = MSE(y_targets , q_values)
    return loss
```

Final results

It took the deep q learning network about 680 episodes to have a score of +200 over the last 100 episodes.

Defining the cost function, that computes the cost between the estimated q value and the target q value. The cost is taken as the mean squared difference between the estimated and the target q values.

```
epsilon = 1.0
total_points_history = []
memory_buffer = deque(maxlen=MEM_SIZE)

for i in range(NUM_EPISODES) :

    state = env.reset()
#     print(state)
    state = state[0]
    points = 0
    for t in range(MAX_NUM_TIMESTEPS) :

        q_vals = q_network(state.reshape(1 , -1))
        action = utils.get_action(q_vals, epsilon)
        next_state , reward , terminated, truncated , _ = env.step(action)
        done = terminated|truncated
        memory_buffer.append(experience(state , action , reward , next_state , done))

        if utils.check_update_conditions(t,NUM_STEPS_PER_UPDATE,
                                         len(memory_buffer)) :

            experiences = utils.get_experiences(memory_buffer)
            update_networks(experiences , GAMMA)

        state = next_state.copy()
        points += reward
        if done :
            break
```

Training the neural network using a random batch of experiences taken from the replay memory to avoid correlation bias.

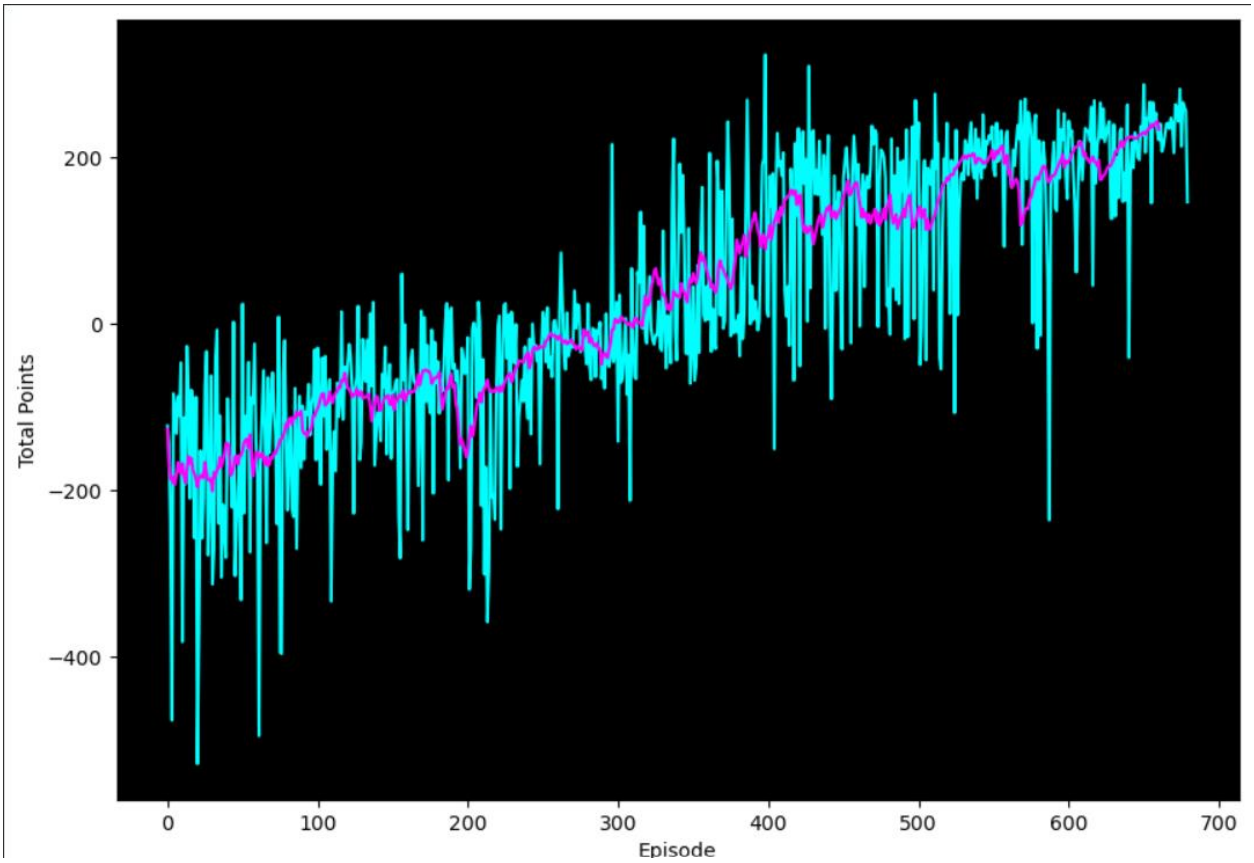
```
def update_target_network(q_network, target_q_network) :
    for q_weight , target_q_weight in zip(q_network.weights , target_q_network.weights)
        target_q_weight.assign(TAU * q_weight + (1 - TAU) * target_q_weight)

def get_action(q_values , epsilon) :

    if np.random.rand() < epsilon :
        return np.random.randint(q_values.shape[1])
    return np.argmax(q_values.numpy()[0])
```

Creating the function to update the target network using the policy network, and the function to select action at any given state using the epsilon value from the epsilon greedy policy.

```
def get_experiences(memory_buffer) :

    experiences = random.sample(memory_buffer , MINIBATCH_SIZE)
    states = tf.convert_to_tensor(np.array([e.state for e in experiences if e is not None]),dtype=tf.float32)
    actions = tf.convert_to_tensor(np.array([e.action for e in experiences if e is not None]), dtype=tf.float32)
    rewards = tf.convert_to_tensor(np.array([e.reward for e in experiences if e is not None]), dtype=tf.float32)
    next_states = tf.convert_to_tensor(np.array([e.next_state for e in experiences if e is not None]),dtype=tf.float32)
    done_vals = tf.convert_to_tensor(np.array([e.done for e in experiences if e is not None]).astype(np.uint8),
                                     dtype=tf.float32)

    return (states, actions, rewards, next_states, done_vals)

def get_new_epsilon(epsilon) :
    return max(MIN_EPSILON , EPSILON_DECAY_RATE * epsilon)
```

Defining the function get_experiences() to get a random sample of experiences from the replay memory. The function get_new_epsilon() updates the epsilon value using the epsilon decay rate.

Final result



Rewards vs Episodes graph

As we can see, the average reward increases as the model gets progressively better.

Tools used

- OpenAI Gymnasium for the Lunar Lander environment
- Tensorflow and Keras for neural network
- Matplotlib library for creating the Rewards vs Episodes graph.
- ImageIO for building the final video output
- Jupyter Notebook as the development environment.