README

Zeyang Lyu

July 28, 2023

1 Introduction

This is a readme file for the Deep Q-Networks(DQN) project to solve the lunar lander problem. The aim is to use deep reinforcement learning algorithms to train a neural network to make a successful landing on the moon. This project includes three .py files: `DQN_agent.py`, `DQN_problem.py` and `DQN_check_solution.py`. To train the neural network, you should run `DQN_problem.py`. After the training, you will get a neural network file called `neural-network-1.pth`. You can run `DQN_check_solution.py` to check the solution.

Click here to view the project in github

2 Environment

2.1 Description

This project uses the environment from OpenAI gym : lunar lander. To import the environment, you can use: `import gymnasium.make("LunarLander-v2")`

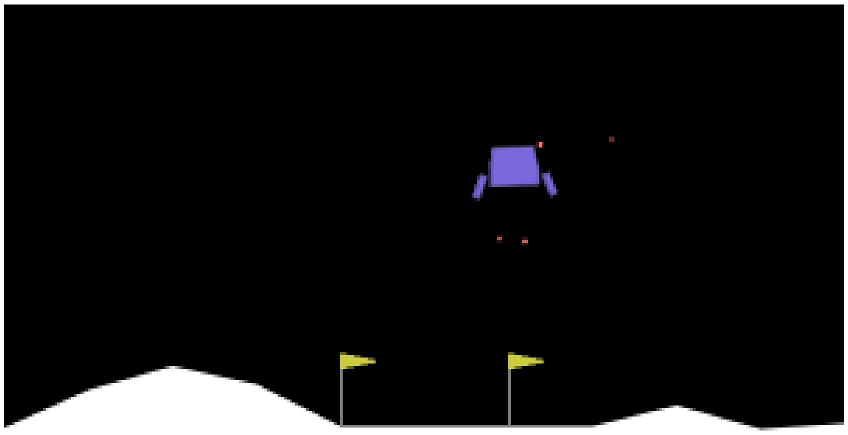

Figure 1: Lunar lander in OpenGym

2.2 Action Space

There are four discrete actions available:

- 0: do nothing
- 1: fire left orientation engine
- 2: fire main engine
- 3: fire right orientation engine

2.3 State

The state is an 8-dimensional vector: the coordinates of the lander in x & y, its linear velocities in x & y, its angle, its angular velocity, and two booleans that represent whether each leg is in contact with the ground or not.

2.4 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.

For more information, you can check the website of OpenGym.

3 Structure

3.1 Network Scheme

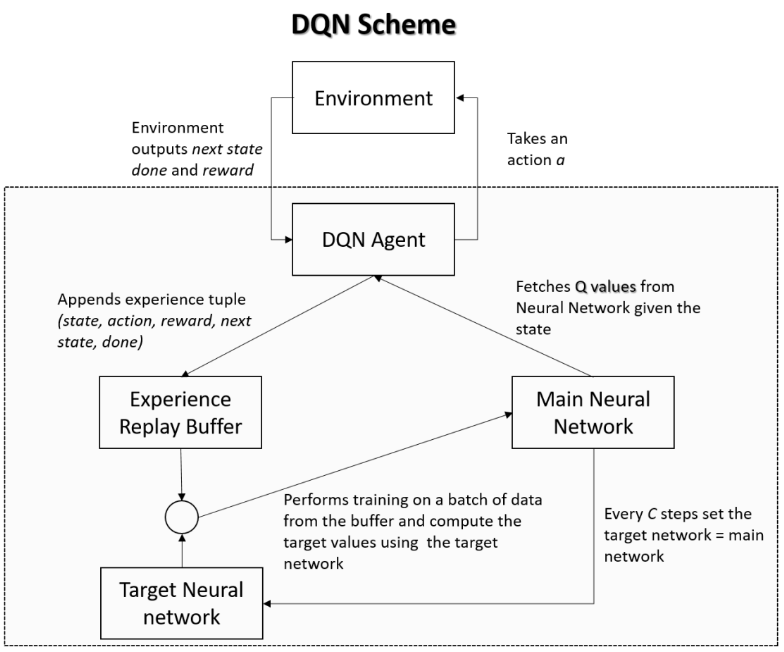
DQN Scheme
Environment
Environment outputs *next state* *done* and *reward*
Takes an action *a*
DQN Agent
Appends experience tuple *(state, action, reward, next state, done)*
Fetches **Q values** from Neural Network given the state
Experience Replay Buffer
Main Neural Network
Performs training on a batch of data from the buffer and compute the target values using the target network
Every *C* steps set the target network = main network
Target Neural network

Figure 2: DQN Scheme

This DQN scheme includes the interaction of the DQN agent and the environment. DQN agent gets the Q values from the neural network and selects the action based on the linear annealing algorithm, ensuring the agent can explore the environment with the possibility of $1 - \epsilon$ and select the action with the biggest rewards with the possibility of ϵ. ϵ decreases linearly with the increase of episodes. After the agent takes the action, the environment will return next state, done, and reward. Then append the experience tuple to the replay buffer. After the replay buffer gets enough experience(larger than batch size), each time the DQN agent takes action, the main neural network will be updated according to the Bellman Equation:

$$Q(s, a) = R(s, a) + \gamma \cdot \max[Q(s', a')] \tag{1}$$

γ is the discount factor and we can get $\max[Q(s', a')]$ through target neural network. Every C steps the target neural network will be updated to the main network. So the main neural network can learn the Q value through each step's reward.

3.2 Result

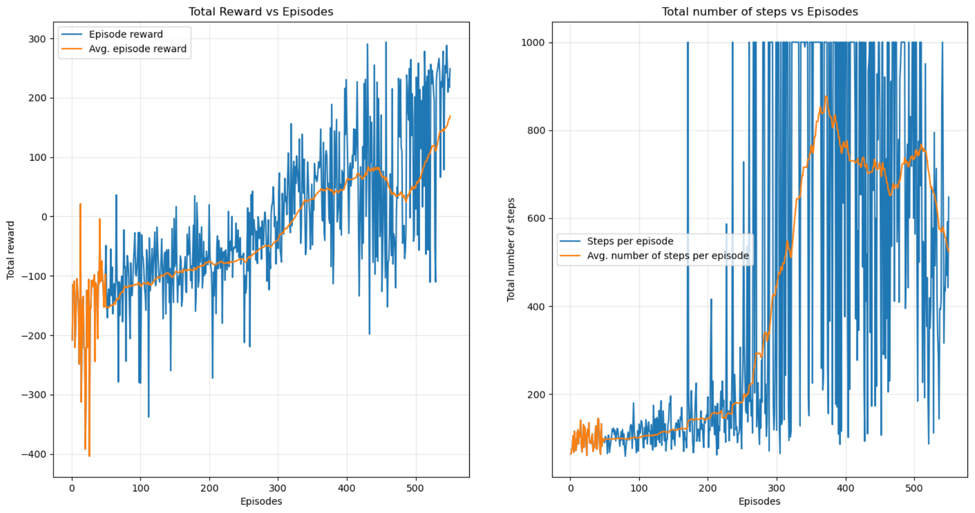

Figure 3: Result

This figure shows how the episodes affect the total rewards and the steps. overall the total rewards go up with the increase in episodes. But we should not set the episode too large because it will cause overfitting and make the result worse.