Project 2: Lunar Lander

Ran Chen
email: ranchen@gatech.edu

Abstract

This report presents a solution to the LunarLander-v2 problem provided by OpenAI Gym (`https://gym.openai.com/envs/LunarLander-v2/`) using deep reinforcement learning methods. Code used to run experiments described in this report is hosted here: `https://github.gatech.edu/rchen350/cs7642summer2018p2`, which is also linked in the header as required

1 Game: LunarLander-v2

Game The game is comprised of a 2D Newtonian mechanical environment, a small rocket is initialized at the top middle position of the 2D environment, with certain random but bounded initial state. The state vector of the rocket is $S = \{x_1, x_2, \dot{x_1}, \dot{x_2}, \theta, \dot{\theta}, L, R\}$, where x_1, x_2, are location coordinates in the 2D environment, and θ is the 2D rotational angle of the rocket, their first order derivatives are the velocities respectively. L and R are binary values indicating whether the rocket's left or right leg is touching the ground. The actions space is discrete, comprised of $\{$*firing left engine, firing right engine, firing main engine, do nothing*$\}$. An episode finishes if the lander crashes or comes to rest, receiving additional -100 or $+100$ points respectively. Each leg ground contact is worth $+10$ points. Firing main engine incurs a -0.3 point penalty for each occurrence. And the goal is to land the rocket within designated location marked by two flags and maximize the total reward.

Approach The Lunar Lander game can be formalized as an Markov decision process (MDP), in which, given any state vector S, an action a needs to be chosen. Clearly this can be solved using Q-learning, where policies are generated based on Q function values for a particular state action pair $Q(S, a)$. Since this game has a large and continuous state space, the naturally suitable solution would be using function approximation to model the Q-function $Q(S, a)$, also because little information is known on what type of function this might be, it would be ideal to use an artificial neural net (ANN) to model it. In this paper, deep Q-networks (DQN)[1,2] are built using PyTorch as part of a Q learning process for solving this game.

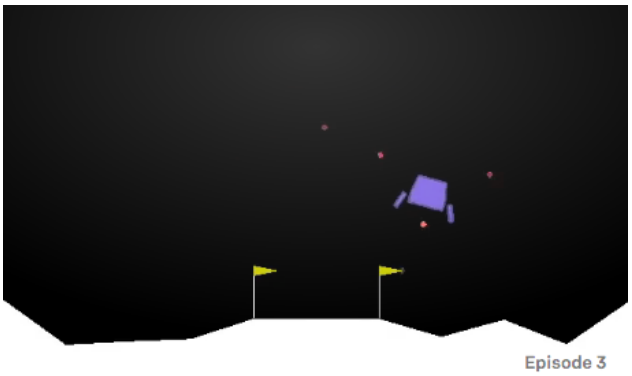


Figure 1: An illustration of the LunarLander-v2 environment from `https://gym.openai.com/envs/LunarLander-v2/`.

2 Experiments and Results

Experiments were run with different algorithms and/or different hyperparameters on the LunarLander game. The training sessions were performed until the rolling average of reward over 100 episode reach 215, and

capped at total 2500 episodes. The testing of the trained networks were run 100 times, and the rewards were plotted.

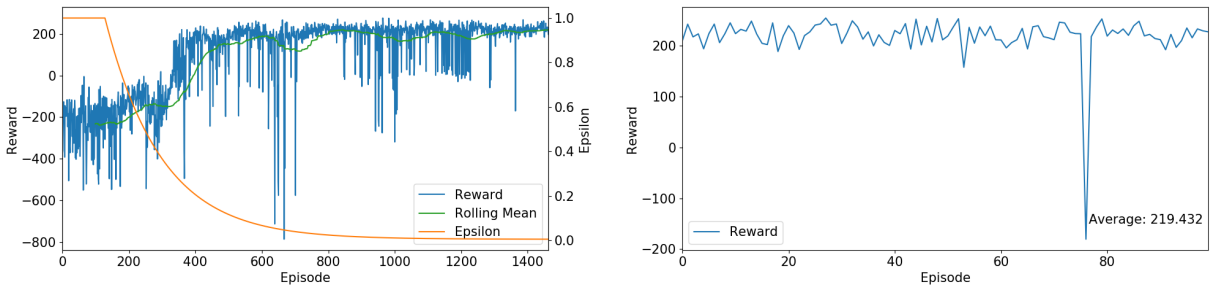
2.1 Experiment 1

First, deep Q networks (DQN) is tested. Here DQN refers to bare-bone DQN with fixed targets and experience replay.[1,2]

Algorithm 1 DQN with fixed Q-targets and experience replay

input: D - empty replay buffer; θ - eval network parameters, θ^- - copy of θ (target network parameters)
input : N_r - replay buffer maximum size; N_b - training batch size; N^- - target network replacement freq
repeat
  reset game: $S \leftarrow env()$
  $R \leftarrow 0$
  for $t \in \{0, 1, ...\}$ **do**
    generate action using ϵ-greedy policy $a = \pi(S)$, observe new state from env: $\{S', r\} = env(S, a)$
    add $\{S, a, r, S'\}$ to D, bump oldest memory from D if $|D| > N_r$
    sample a minibatch of N_b tuples $\{S, a, r, S'\}$, construct eval and target Q values for each of N_b tuples:
    $Q_{\text{eval}} = Q(S, a; \theta)$ and $Q_{\text{target}} = \begin{cases} r & \text{if } S' \text{ is terminal} \\ r + \gamma \max_{a'}(Q(S', a'; \theta^-)) & \text{otherwise.} \end{cases}$
    one step optimization on θ with loss $\|Q_{\text{target}} - Q_{\text{eval}}\|$
    $R \leftarrow R + r$
    $\theta^- \leftarrow \theta$ if $t \mod N^- = 0$
  end for
until no significant improvement in R

As shown in **Fig** 2a, with the right combination of hyperparameters ($\alpha = 5 \times 10^{-5}$, $\gamma = 0.99$, and target net is update every 500 optimizations to the eval net). The rolling average (green line) comes close to convergence when ϵ falls below 0.2. The testing result (**Fig** 2b) is also satisfactory, scoring 219 on average. However the learning curve demonstrated certain degree of instability after convergence. According to literature, using "soft updates", i.e. incrementally updating target network every time eval net is optimized, could help stabilize the learning curve by avoiding making drastic changes to target networks when outlier samples are incurred.



(a) (b)

Figure 2: DQN with fixed Q-targets (updating interval: 500): earning curve (a) and testing result (b)

Here the optimal learning rate ($\alpha = 5 \times 10^{-5}$) was obtained by experimenting. Nearby values was also tested for performance. As shown in **Fig** 3a ($\alpha = 2.5 \times 10^{-4}$) and 3b ($\alpha = 2 \times 10^{-4}$), the results indicate that large learning rate tends to generate extremely unstable learning curve, although for many episodes, the learning reward can reach above 200. Further changing α away from optimal seems deteriorate the performance more significantly, as shown in **Fig** 3c ($\alpha = 5 \times 10^{-2}$) and 6a ($\alpha = 5 \times 10^{-6}$), the agent is not learning anything in those situations and hardly reaching any positive reward.

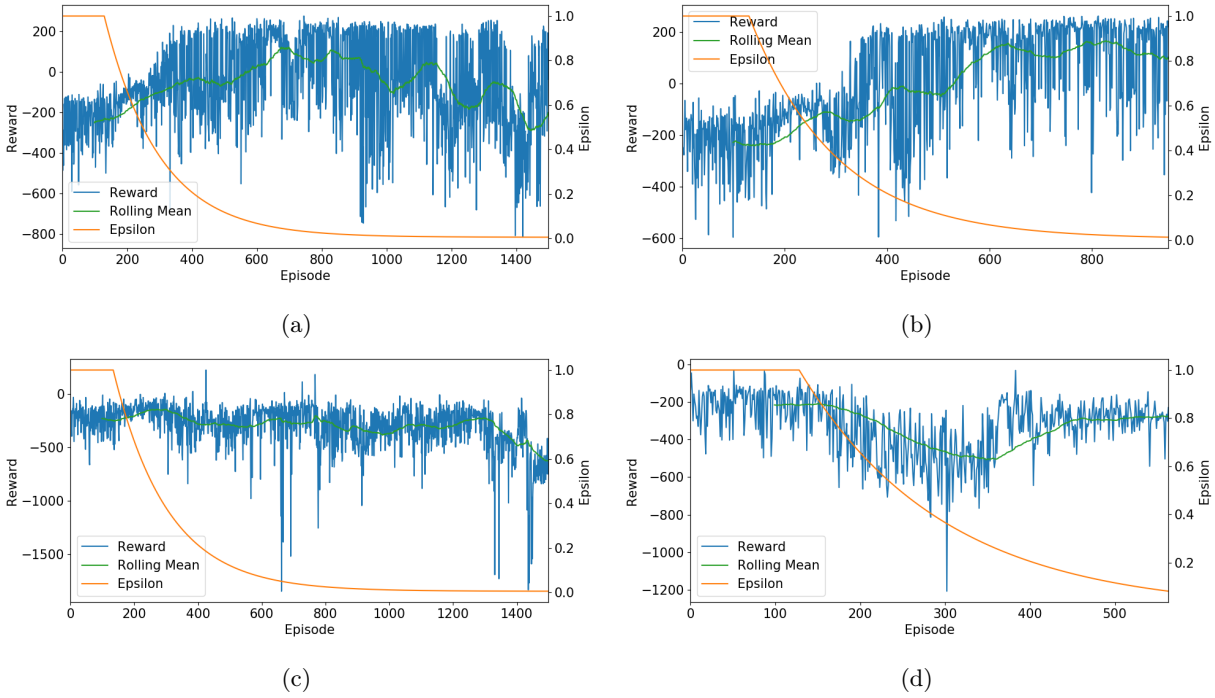


Figure 3: DQN with fixed Q-targets (updating interval: 500): $\alpha = 2.5 \times 10^{-4}$ (a), $\alpha = 2 \times 10^{-4}$ (b), $\alpha = 5 \times 10^{-2}$ (c), and $\alpha = 5 \times 10^{-6}$ (d).

2.2 Experiment 2

Now we replace fixed Q-targets which updates target network every 500 optimizations to eval network with a new strategy called "soft update", which update target network every time the eval network is optimized, but only by a fraction. Namely the update to target network is replaced by $\theta^- \leftarrow (1-\tau)\theta^- + \tau\theta$, instead of $\theta^- \leftarrow \theta$ if $t \mod N^- = 0$.[3]

DQN with "soft update" ($\tau = 0.02$) generated much more stable learning curve compared to fixed Q-targets (**Fig** 4a) and converged as early as around episode 650. This algorithm also achieved better testing score at 227 on average (**Fig** 4b).

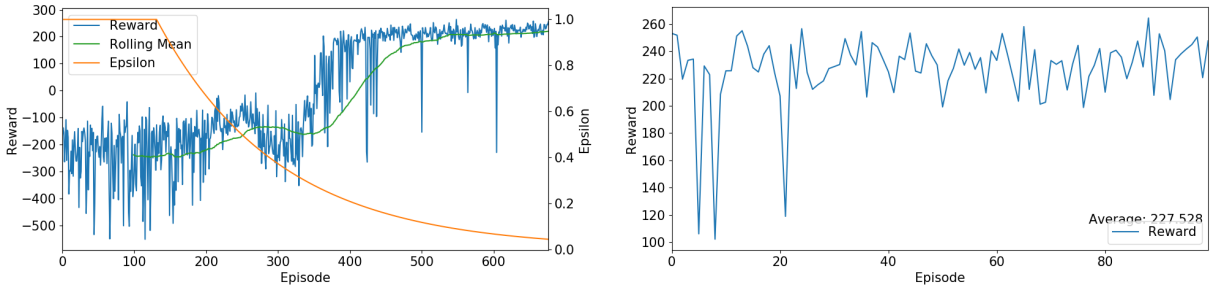
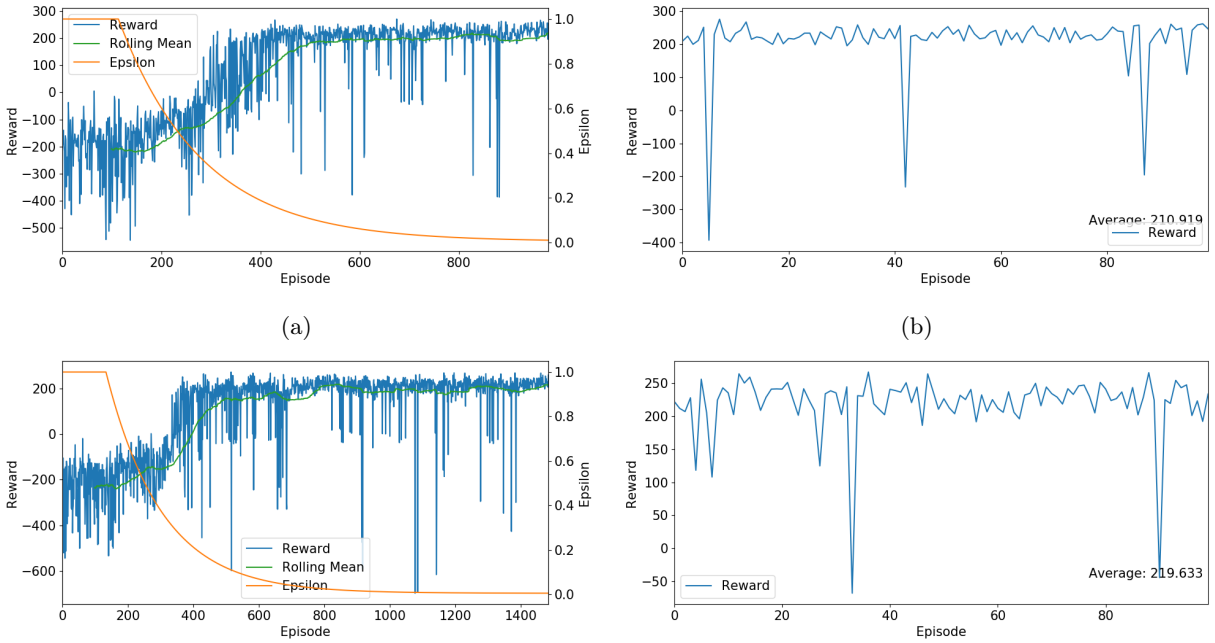


Figure 4: DQN with "soft update" ($\tau = 0.02$): learning curve (a) and testing result (b)

The most import hyperparameter for soft updated DQN is τ which determines how much the target network is updated by the eval network. Too large or too small update could both cause the two networks to drift away, resulting in increased loss, and in turn lead to instability in learning curves. As shown in **Fig** 5a and 5c, when $\tau = 0.01$ and $\tau = 0.05$ the learning curves were less stable and required more episodes to

converge, although the testing results are both still satisfactory, scoring 210 and 219 on average respectively.



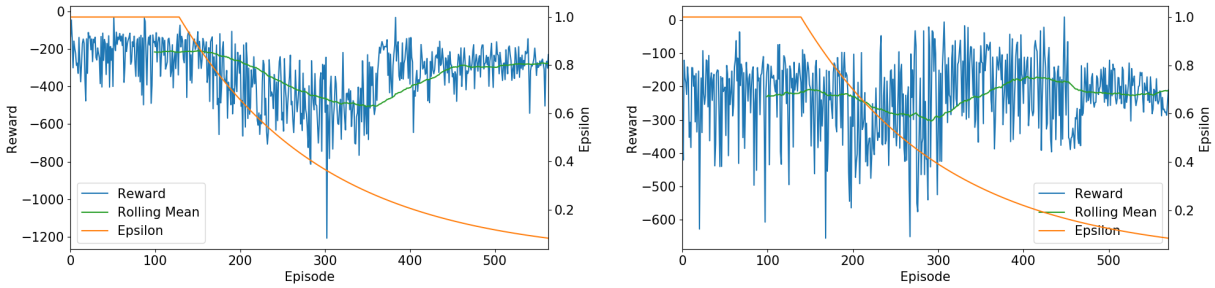
(a) (b)

(c) (d)

Figure 5: DQN with "soft update" ($\tau = 0.01$): learning curve (a) and testing result (b); and ($\tau = 0.05$): learning curve (c) and testing result (d)

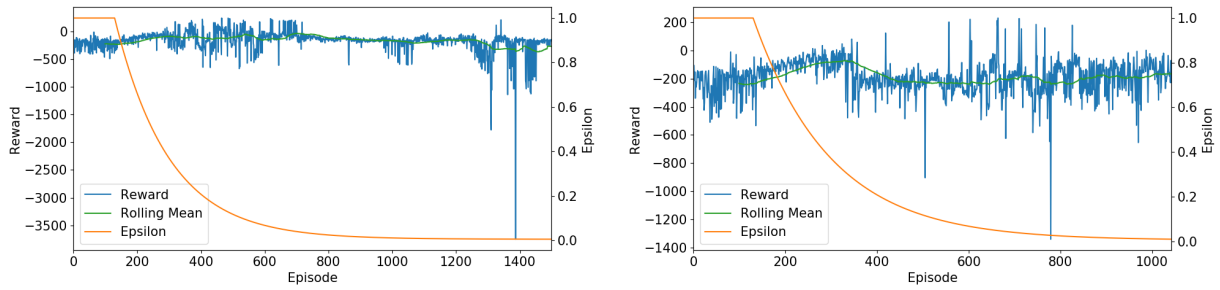
2.3 Experiment 3

Some preliminary tests were run using Double DQN (DDQN) algorithm,[4] in which Q value of the target network is chosen by the eval network: $Q_{\text{target}} = r + \gamma Q(S', \text{argmax}_{a'} Q(S', a'; \theta); \theta^-)$, instead of $Q_{\text{target}} = r + \gamma \max_{a'}(Q(S', a'; \theta^-))$. The rest of the algorithm is identical to DQN, either with fixed Q-targets or "soft update". DDQN is supposed to be able to further stabilize the learning by preventing the two networks from drifting away from each other. However, I ran out of time searching for the ideal hyperparameter combinations for DDQN on LunarLander-v2. Here are the results: for DDQN with fixed Q-targets, the agent does not to be able to learn at all (shown in **Fig** 6a and 6b); for DDQN with soft update, the learning curve seems to converges to some sub-optimal value, with huge variances (shown in **Fig** 7a and 7b).



(a) (b)

Figure 6: DDQN with fixed Q targets (update interval: 500) at $\alpha = 5 \times 10^{-4}$ (a), and $\alpha = 5 \times 10^{-2}$ (b).



(a) (b)

Figure 7: DDQN with soft update ($\alpha = 5 \times 10^{-4}$) at $\tau = 0.05$ (c), and $\tau = 0.02$ (d).

3 Difficulties

- DQN with fixed Q-targets
  - Extremely large continuous hyperparameter space.
  - Extremely high dimensional hyperparameter space.
  - No intuition or theoretical support on the range of hyperparameters. I ended up using a grid search script to cover very large space
  - Ambiguity in algorithmic specifics and naming conventions: is fixed Q targets and memory replay considered part of a "bare-bone" DQN, etc.
- DQN with soft update
  - Introduced yet another hyperparameter τ to tune.
- DDQN
  - Introduced yet another hyperparameter (a boolean to control if DDQN is used) to tune, making the totally number of parameters up to 20.
  - Again, the lack of intuition or theoretical support on the range of hyperparameters makes it extremely difficult to find a successful combination of hyperparameters.
  - If I had more time, I would like to search for hyperparameter combinations for DDQN and run tests to see if it is indeed able to stabilize learning process.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602 [cs]*, December 2013.

[2] Melrose Roderick, James MacGlashan, and Stefanie Tellex. Implementing the Deep Q-Network. *arXiv:1711.07478 [cs]*, November 2017.

[3] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*, September 2015.

[4] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-learning. *arXiv:1509.06461 [cs]*, September 2015.