Lunar Lander and DQN

May 20, 2021

Abstract

Lunar lander is an interesting control problem in the well-known OpenAI gym library. In this paper, the lunar lander problem was solved by implementing the Deep Q-Networks (DQN) algorithm. After around 538 episodes, the approach reached the average score more than 200 points over 100 consecutive runs.

1 Introduction

Lunar lander is one of the problems or environments in the OpenAI gym library. In this problem, a rocket needs to be controlled in order to land safely on the landing pad. While it is possible to play it with the keyboard, an automatic solution is still attracting taking consideration of the complexity in the environment. Mnih et.al (2013)[1] introduced the Deep Q-Networks (DQN) approach to play the classical Atari games with better performance than human experts. The approach was supplemented in Mnih et al. (2015)[2] by separating the policy and target network to achieve higher stability in converging. Lillicrap et al. (2015)[3] further improved the algorithm by using a soft update approach in each learning instead of directly copying the weights to the target network. In this paper, the soft update approach was implemented to solve the lunar lander problem. More details will be shared in the following sections.

The article is organized as follows. In the second section, the environment of lunar lander is introduced with more details. In section 3, the design of experiment is explained with more details in the algorithm. In section 4, the results of the experiment are presented and discussed. The tuning of some hyper-parameters are also discussed. In the final section, I will talk about the pitfalls and difficulties during the project.

2 Environment - Lunar Lander

In the gym library, there are two kinds of lunar lander: *LunarLander* with discrete action space and *LunarLanderContinous* with continuous action space. In this experiment, the discrete lunar lander environment is used. The description of the environment is quoted from the project description as follows:

"The four discrete actions available are: do nothing, fire the left orientation engine, fire the main engine, fire the right orientation engine. The landing pad is always at coordinates (0,0). Coordinates consist of the first two numbers in the state vector. The base reward for moving from the top of the screen to the landing pad depends on a number of factors. If the lander moves away from the landing pad it is penalized the amount of reward that would be gained by moving towards the pad. An episode finishes if the lander crashes or comes to rest, receiving an additional -100 or $+100$ points, respectively, on top of the base reward. Each leg-ground contact is worth $+10$ points. Firing the main engine incurs a -0.3 point penalty and firing the orientation engines incur a -0.03 point penalty, for each occurrence. Landing outside of the landing pad is possible. Fuel is infinite, so, an agent could learn to fly and land on its first attempt."

3 Design of experiments

The experiment was implemented on my local machine with PyTorch. Before everything, we need to solve two problems that may cause the divergence of the learning. One of the difficulties that stop the application of neural networks on the reinforcement learning is the existence of correlation. In a control environment like lunar lander, the correlations in the state sequences are extremely high which may cause the divergence with non-linear operations. In order to remove such correlations, the idea of *experience replay* was introduced in the learning process. Instead of feeding the state sequences into the neural networks, we randomly sample historical data from previous episodes as the input for the network. To achieve this idea, during the training process, we need to memorize the observations. And to improve efficiency, in practice we only need the most recent observations whose size are controlled by a hyper-parameter.

The other problem is the instability of the Q-learning process. Because of the high correlations between action Q values and target Q values, the neural networks are actually very sensitive to the Q values, which may case the divergence. Mnih et al. (2015) improved the DQN algorithm by generating a separate target neural network to update the targets. What's more, this separate network is updated from the policy network by a certain frequency which is controlled by another parameter, and is then used to compute the target Q values. However, the *hard update* approach did not perform well for me. Instead, the *soft update* method successfully helped stable the learning curve. After each learning, the weights in the target network are updated by changing just a litter bit towards the policy Q table. The degree of the change is controlled by a hypter-parameter τ.

Algorithm 1: DQN

```
Input: state batches
Initialize memory $D$ with size $N$;
Initialize action-value Q table $Q$;
Initialize target Q table $\hat{Q}$;
for episode=1 to M do
    Initialize environment;
    for step t=1 to T do
        with probability $\epsilon$ randomly select $a_t$,
          otherwise update $Q(s_t, a_t)$ and select
          $a_t = \text{argmax}_a Q$;
        select action $a_t$ with $\epsilon$-greedy algorithm;
        execute action $a_t$;
        observe state $s_t$, reward $r_t$ and next
          state $s_{t+1}$;
        store the transition pair $(s_t, a_t, r_t, s_{t+1})$
          into the memory $D$;
        set $s_t = s_{t+1}$;
        for Every C step do
            sample random batches
              $(s_j, a_j, r_j, s_{j+1})$ from $D$ with size
              $B$; update table $\hat{Q}(s_{j+1}, a')$;
            set $y_j = r_j + \gamma \max_{a'} \hat{Q}$ for
              non-terminal states otherwise
              $y_j = r_j$;
            perform gradient descent step on
              the loss function;
            soft update $\hat{Q} = \tau * Q + (1 - \tau) * \hat{Q}$
        end
    end
end
```

As presented in Algorithm 1, the deep Q-learning network is implemented as follows. Firstly, the neural network architecture contains two hidden layers with 64 units each. The input layer and output layer has the size of state size and action size, respectively. The input of the network is observed states which are randomly sampled from the memory. And the output is the values corresponding to each action. Two networks are built as a policy net and a target net. In each step when we select the action, the ϵ-greedy algorithm is performed. During the training process, the ϵ decays with a certain rate from 1.0 to minimum value 0.01. Such settings encourage exploration at the beginning and exploitation near end. After action execution and observation, the transition is stored in a *deque* which is a container type in Python with fixed-length. For every C step, the agent will learn from the memories. In each learning process, the memories are randomly sampled as batches. The target Q values are then updated from the target net by feeding the state batch and then applying the Bellman equation. The gradient descent steps are then performed to minimize the loss function. In this implementation, the loss function is the mean squared errors (MSE) between policy action-value Q and expected value y from the bellman equation.

$$L = \frac{1}{B} \sum_{1}^{B} (y_j - Q(s_j, a_j))^2$$

The solver for the gradient descent is *Adam*. Finally, to improve the stability of the algorithm, the weights in the target net was updated with the soft update approach:

$$w_{\hat{Q}} = \tau * w_Q + (1 - \tau) * w_{\hat{Q}}$$

where τ is far less than 1. As the goal of the project, the training should end when achieving a score of 200 points or higher on average over 100 consecutive runs. In the implementation, however, the agent will run more episodes to demonstrate the performance and also improve the stability of the trained agent.

4 Result

4.1 Training agent

In this agent, the reward achieved 200 points after 538 episodes. The size of the batch is 64 and the memory has maximum size of 10000. The learning rate of Adam solver is set to be 0.001. For the Q learning, the discount factor γ is set to be 0.99 which is the same to the value in previous studies. The agent will learn from the memories every 2 steps. The soft update parameter τ equals 0.001. The value of ϵ in the action selection step decays during the whole training process with the decaying factor equaling to 0.995. The reward curve is presented in Figure 1.

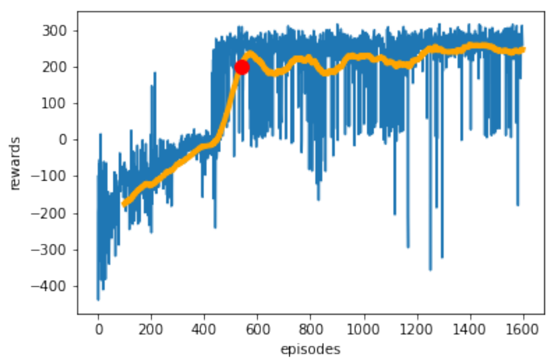

Figure 1: Reward for each training episode. The orange line represents the moving average of rewards in the past 100 episodes. The red point is the problem-solved point.

4.2 Trained agent

The agent is then tested for 100 consecutive runs with the network weights after training. For testing, the agent simply execute actions and make observations. The actions are still selected based on the previous ϵ-greedy method with ϵ fixed to 0.01. The total rewards for testing episodes are presented in Figure 2. The average reward in the 100 runs surpassed 200 points (the orange line).

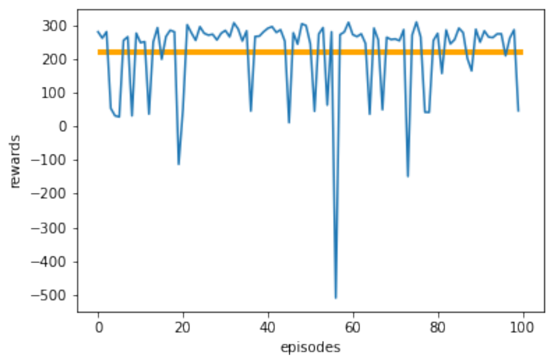

Figure 2: Reward per episode with trained agent ($\epsilon = 0.01$)

4.3 Tuning hyper-parameters

4.3.1 update frequency C

The update frequency C decides how frequent the agent will learn from the stored memories and update the Q tables. As introduced in the first section, the DQN algorithm has been improved for several times. One of the major improvement is about the frequency update and update method. A larger C value will lead to fewer learnings in the training process and possibly worse performance within the same number of episodes. However, a small C, which indicates a high frequency of learning and updating of Q tables, may also lead to the divergence because of higher correlations. From Figure 3, with the same random seed, $C = 0$ achieves the best performance with the fastest convergence. In the $C = 10$ and $C = 100$ cases, the agents showed poor learning result due to the lack of learning.

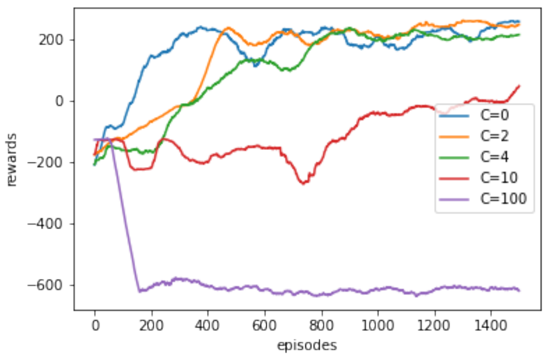

Figure 3: Moving average reward for 100 consecutive training episodes with different C

However, in this experiment, $C = 2$ is still chosen as the final value because of lower volatility and higher time efficiency. Figure 4 shows the rewards in each episodes during the training for both $C = 0$ and $C = 2$ scenarios. With all steps engaged in learning from memories ($C = 0$), the agent only need around 385 episodes to get more than 200 points over 100 runs and solve the problem. However, it was rather time consuming for the agent to learn on each step.

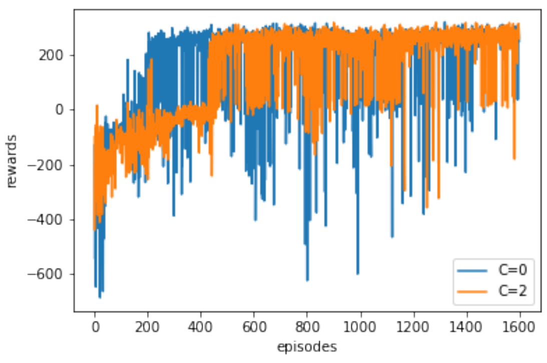

Figure 4: Reward for each training episode with different C

If we update the Q tables every two steps instead, the agent took much less time to learn and the result is still acceptable. What's more, it is also found that the orange curve has a lower volatility, which suggests a more stable performance and less risk of failing. Such tradeoff can be more well balanced in the future if the neural network can work faster with more tweaks.

4.3.2 soft update parameter τ

This parameter controls how much the weights of target network will change toward the policy network. The existence of the parameter is to avoid the divergence because of the correlations. Otherwise the updated Q table from policy net would also be used to compute the target net.

The tuning result of parameter τ makes the importance of soft update more clear. A larger value of τ indicates more movement toward weights in policy net, which makes the performance and stability of the agent getting worse. When τ is too small, however, the updating of target network would become slow, which heavily delays the problem-solved point.

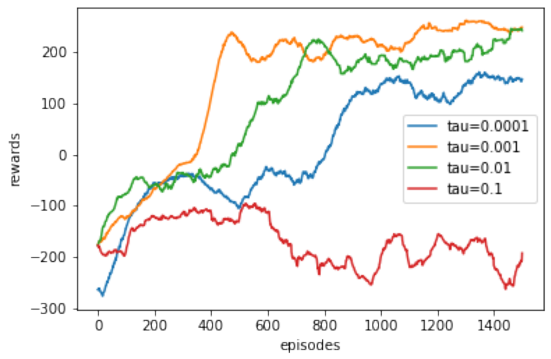

Figure 5: Moving average reward for 100 consecutive training episodes with different τ. Both too large and too small values show poor performance.

4.3.3 learning rate

The learning rate determines the step size of the Adam optimization solver which was proposed by Kingma and Ba (2014)[4]. In gradient descent optimization, learning rate will affect how fast the optimizer find the optimum and also its performance. For example, a larger learning rate will seek the solution faster, but potentially lead to worse performance due to the large step. A smaller learning rate, on the other hand, will eventually find a better result by sacrificing time efficiency. Since time is of course a significant factor to consider in reinforcement learning, the value of learning rate should be examined with caution. As shown in Figure 6, the learning rate can effectively affect the result of the training agent and the result is consistent with our hypothesis.

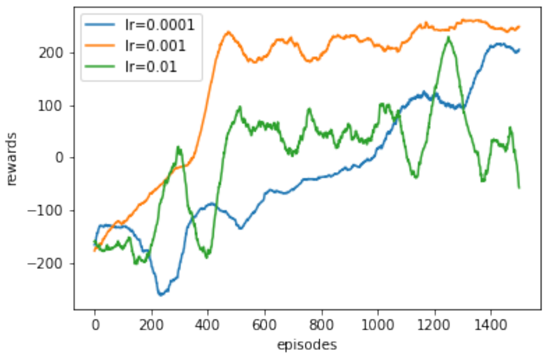

Figure 6: Moving average reward for 100 consecutive training episodes with different learning rate of optimization solver.

5 Discussion on the Difficulties

The project is challenging because I was not very familiar with deep learning models or PyTorch previously. Therefore, it cost me a huge amount of time to understand how to employ the neural network into the Q learning process. It was not straightforward to me at first to understand why we should add a separate target network.

Another problem I faced was the divergence. As noted by Mnih et al. (2013), neural networks generally do not match well with reinforcement learning because of the correlations. In fact, removal of the correlations is more difficult than theories. My first implementation was the direct application of the first DQN algorithm. However, despite tuning the parameters for days I failed to reach the 200 points for 100 runs and the reward curve shows little signal of learning. After adding a separate target network and adding update frequency to slow the learning, the agent becomes smarter by reaching a higher average score around 100 points but it's still not stable in the performance. The results generally suggest that the correlation still exists even with hard updating method and a separate target network. The review of literatures saved me from wasting too much time on tuning parameters in a bad agent. The soft update method in Lillicrap et al. (2015) is similar to the hard update one but bring more stability to the performance and higher average scores.

Due the time limit and too much time spent on tuning parameters on the bad agents, the performance of the current agent still has much space to improve. For example, the update frequency parameter C stills need to be carefully selected by balancing the tradeoff between the training speed and the performance. The agent can also possibly be improved in speed in the future with some tactics in PyTorch.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[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 preprint arXiv:1509.02971*, 2015.

[4] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.