Training the Lunar Lander Agent With Deep Q-Learning and Optimizing Hyperparameters

Yufeng Wang
Georgia Institute of Technology
8b1f5b878a5ac9f1285b7d454ba40aca3a1d4555

***Abstract*—Recently, reinforcement learning has been successfully applied in different problems like self-driving cars, trading and finance, and playing video games. In this paper, we solve a well-known robotic control problem - the lunar lander problem using Deep Q-Learning under OpenAI Gym's LunarLander-v2 Environment. The winning agent can achieve over 266 average rewards for 100 test episodes. The paper will also show that different hyper-parameters, like batch size, learning rate and update size, affect both training speed in episodes and performance in rewards.**

I. Introduction

Lunar Lander problem is the task to control the fire orientation engine to help the lander land in the landing pad. LunarLander-v2 is a simplied version of the problem under OpenAI Gym environment[1], which requires the agent to move in 8-dimensional state space,with six continuous state variables and two discrete ones, using 4 actions to land on pad: 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). The coordinates are the first two numbers in the state vector.If the lander moves away from the landing pad it loses reward. The episode finishes if the lander crashes or comes to rest, receiving an additional -100 or +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. To get a stable and successful agent, landing safely on the pad with an average rewards over 200 in 100 successive episodes is needed.

In this paper, We implemented the Deep Q-Learning algorithm to solve the problem with over 266 average rewards in 100 test episodes. The paper is structured as follows: In section 2, we will describe the winning solution and discuss the results. In section 3, we will review how different parameters for batch size, target network update steps, and learning rate will influence the agent's learning process. We will next discuss some failed attempts or pitfalls we tried with disappointing results in section 4. Lastly, we will conclude paper with further improvements.

II. Deep Q-Learning with Target Network

Deep Q-learning is an improved algorithms over Q-learning. In Q-learning, a lookup table with the rewards of each pair of (state, action) will be updated during training. However, when states are continuous or the number of states is very large, it is memory-expensive to maintain a large table to save the rewards. To solve this problem, a value function approximators is introduced to provide value prediction given the state, and deep neural network is one option for generalization after many training episodes.

An efficient Deep Q-learning uses experience replay and target network with Epsilon-greedy strategy to select actions to interact with the environment, and update the weights of the neural network from the difference between the maximum possible rewards for next state and the predicted value for current state from the Q-network. "Figure 1" shows the mathematical representation of the Deep Q-Learning Algorithm.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Fig. 1. Deep Q-Learning, Source[2]

The experience replay randomly generates a batch of training samples from a memory buffer with a predefined maximum size, where we store all the history experience. This method utilizes history experience more efficiently by learning with it multiple times, and tackles the problem of autocorrelation leading to unstable training by making the problem more like a supervised learning problem. When we use a large enough buffer to store the history experience and randomly select a batch of experience, it can also avoid correlation between continuous samples.

Deep Q-Learning uses not one but two neural networks: behavior network and target network. As we know, in the standard Q-Learning, we update exactly one state-action pair at each timestamp, whereas with Deep Q-Learning we update many pairs, which can affect the action values for the very next state instead of guaranteeing to be stable as in Q-Learning. Using a stable target network can solve this problem. The target network copy the weights from the behavior networks at not one but every few steps.

To make a balance of exploration and exploitation, we use Epsilon-greedy strategy to select the actions in training phase.

We try both epsilon decay in each iteration and epsilon decay in different stages, and we find the second method gives us better performance. Specifically, the epsilon is set based on the following rules: epsilon = 0.5 for first 50 iterations, epsilon = 0.3 when iteration is from 50 to 100, epsilon = 0.2 when iteration is from 100 to 300, epsilon = 0.1 when iteration is from 300 to 500, epsilon = 0.05 when iteration is from 500 to 1,000, and epsilon = 0.01 when iteration is over 1,000.

As for the Reproducibility, we set a fixed random seed for python libraries like *numpy*, *tensorflow*, and *random*.

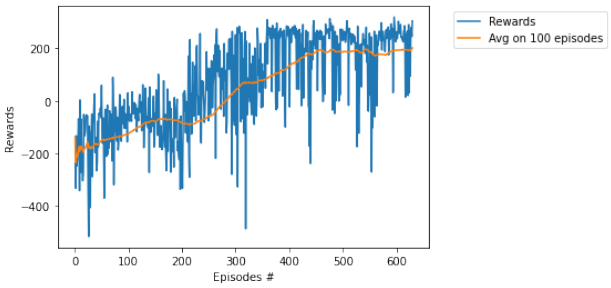

Fig. 2. Rewards for each episode in training

In our winning agent, after much efforts in the parameter tuning, we use two hidden layers with 32 and 64 dimensions respectively. We update the target network every 20 steps, and optimize the network with Adam optimizer with 0.001 learning rate and 64 as the batch size. During the training, we monitor the rolling mean of 100 episodes and if the average reward is over 200, we stop training the agent. Every episode will be finished either when the steps reach predefined maximum steps, 1,000 here, or when done=True from the gym environment.

"Figure 2" shows the reward values per experience at the time of training. Blue lines denote the reward for each training episode and the orange line shows the rolling mean of the last 100 episodes. The agent keeps learning with the time and the value of the rolling mean increases with the training episodes. The average started to be positive after around 300 episodes, and turn to over 200 after 600 episodes. Even the lowest reward in the last few sliding windows is above 0.

"Figure 3" shows the performance of the trained model for 100 episodes in the Lunar Lander environment. Instead of using epsilon-greed to explore the environment in training phase, in testing we use the prediction from the trained model to selection the action with maximum probability. The trained model is performing well in the environment with all the rewards being positive. The average reward for 100 testing episodes is 266, where even the lowest reward is over 120. The results show that the agent is also very stable, guiding the lander to land on the pad safely. In only three trials the rewards are below 200. You can check the annotated GIF for this trained agent in [3].

Fig. 3. Rewards per testing episode

III. Effect of Hyperparameters

It is critical to analyze how hyperparameters affect the training process and the performance of the agent. In this section, we test batch size, update size for target network, and learning rate for the neural network in Deep Q-Learning. For each hyperparameter, we choose three values and compare the minimum episodes needed to achieve average rewards over 200 in successive 100 episodes. We found that different hyperparameters have significant impacts on the learning process of the agent.

A. Batch Size

We use different batch sizes to check how it influences the training process. As we can see in "Figure 4", the agent with batch size 64 is converged faster than another two agents with batch size 32 and 128 respectively. If the batch size is too small(32), it needs nearly 2,000 iterations to get enough data to converge. However, if the batch size is too large(128), it also performs bad because larger batch size brings some difficulties in numerical optimization of network weights[4], but it performs still better than the agent with 32 batch size.

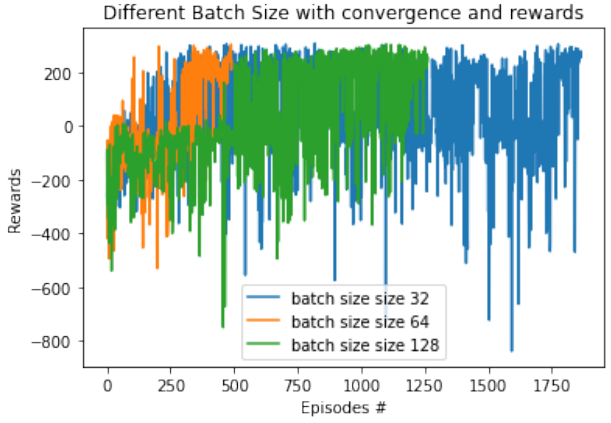

Fig. 4. The Effect of Batch Size

B. Update Size

As shown in "Figure 5", we evaluate the frequency of updating the target network and its influence on training.

When the update size is 10, the agent is trained with over 1000 episodes to be converged. The best agent is trained with update size 20, which need around 450 episodes to converge. The reason can be that if we update the target network too often(e.g., update size 10), it will affect the stability of the network, while if we update the target network less often (e.g., update size 30), although the training is stable, the model learns slowly due to delayed weights update. We also trained an agent without target network, and it turned out that the agent learns over 10,000 episodes and still can't solve the problem. In conclusion, the necessary and the right update size are very important to the success of the Deep Q-Learning agent.

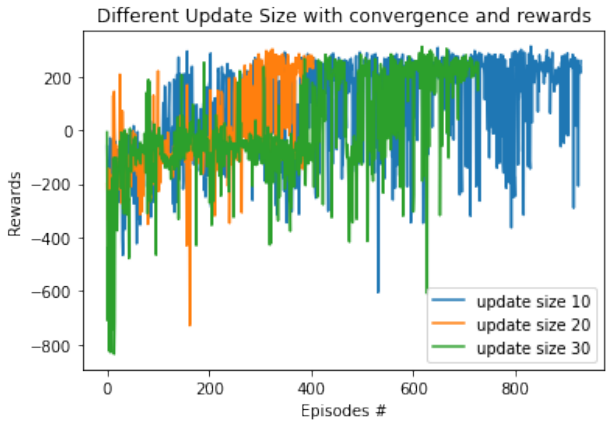

Fig. 5. The Effect of Update Size

C. Learning Rate

Learning Rate also plays an important role in training the agent. We test three learning rate: 0.001,0.0015,0.002. The "Figure 6" shows that when learning rate is 0.001 the performance is better and the model is converged with around 450 episodes. As learning rate increases, the neural network weights update more from gradient descent, causing more training steps to find the global minimal for the loss function.

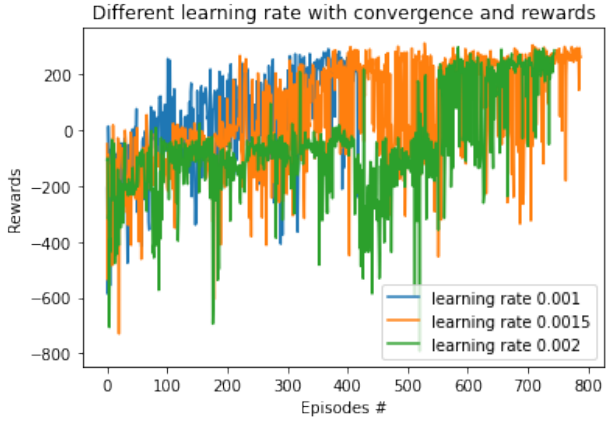

Fig. 6. The Effect of Learning Rate

IV. Failed Attempts and Pitfalls

During the project, we faced several failed attempts when we tried to solve the problem. In the beginning, we chose a vanilla Deep Q-Learning without target network, which took many long hours and the model did not learn well. Then we used wrong values for learning rate and chose improper optimizer for the neural network, and got very bad performance. Luckily, we solved all of them with many experiments. We used Google Colab to run multiple experiments at the same time to find the optimal values to solve the Lunar Lander problem.

A. Vanilla Deep Q-Learning

We start to solve the lunar lander problem with the Deep Q-Learning without target network, and it is painful to watch the agent to learn in over 10,000 training episodes without even one single positive reward, although we test different parameters. Later we used target network to solve this problem and get the agent converged very fast, with few hundred episodes. The target network provides a more stable learning environment, where the weights of the network are updated every so many steps with a copy of the behavior network.

B. Optimizer

To train the neural network for the Deep Q-learning, different optimizers, like Adam, SGD, AdaDelta, and RMSProp have been used to compare the performance. It turned out that only Adam optimizer provides fast convergence to the model, while others need more than 2,000 episodes for training. Given limited time, we did not spend much time on the parameter tuning for other optimizers, but that's definitely worth trying in the future.

C. Improper learning rate

In the beginning, we set the improper learning rate as 0.01 or 0.005 and it took more than 8 hours to make the agent learn the policies. Later we used 0.001 and the model made to solve the problem in 40 minutes. Proper learning rate can provide better generalization ability to the neural network in Deep Q-Learning. As shown in "Figure 7", the neural network uses gradient descent to update the weights for the hidden units, and if the learning rate is to large, the loss function will be stuck in a local minimal and need more and more training data to find the global minimal. If the learning rate is too small, the weights need more steps to find the local minimal.

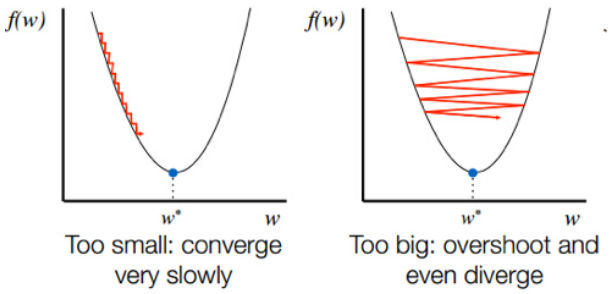

Fig. 7. Learning Rate: Too large or Too small, source[5]

D. Double Deep Q-Learning

Deep Q-Learning use the highest Q-value to calculate TD error. However, is the assumption that the best action for the next state is the action with the highest Q-value is true? We know that the Q values depends on the actions we take and the neighborhood states we explored. If we take the highest Q-value, it will bring us the problem of the overestimation of Q-values. Double Deep Q-Learning is the method to solve this problem.

When we implemented the algorithm, we made a mistake which took us some time to detect. In Double Deep Q-Learning, the model uses the behavior network to predict the values for next state, and select the action based on the Q values. Then the model uses the target network to calculate the value for next state given the action from behavior network. Finally, the model used the behavior network again on the current state, and use the values from the target network to update the Q values for current state. In the beginning it is very confusing to think which network to use in which step. We need to use the behavior network twice: one for predicting values for current state to update the value, another for predicting the values for next state to select actions with maximum rewards. We made mistake here and it took long time for the agent to learn, much worse than the winning Deep Q-Learning solution. After checking the paper[6] over and over again we finally implemented the Double Deep Q-Learning correctly and successfully.

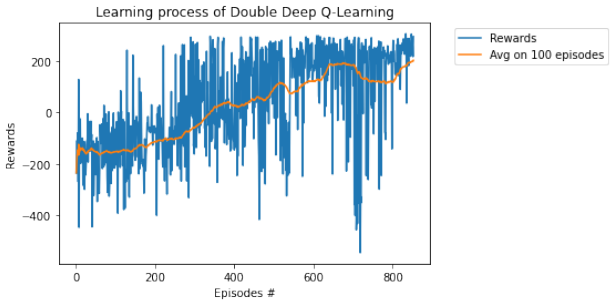

Fig. 8. Learning process of Double Deep Q-Learning

V. Conclusion

In conclusion, we observed that Deep Q-Learning is a powerful algorithms that can efficiently solve challenging problems such as Lunar Lander. However, selecting the proper parameters and networks, and using target network and experience replay significantly affects its performance. It is critical to tune the parameters like learning rate for the main network and update size for target network.

For future work, we would like to try other models like Sarsa, DDDQN, PPO and others. This will provide a more clear comparison of the efficiency of different algorithms. Besides the model side, we can change the structure of the neural networks as well, for example, add Dropout layer or BatchNormalization, or use more layers with different hidden units to the network.

References

[1] . [Online]. Available: https://gym.openai.com/envs/LunarLander-v2/.

[2] V. M. K. K. D. S. A. G. I. A. D. W. M. Riedmiller, "Playing atari with deep reinforcement learning," [Online]. Available: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf.

[3] . [Online]. Available: https://github.gatech.edu/gt-omscs-rldm/7642Spring2021ywang3886/tree/master/Project_2.

[4] P. A. Adam Stooke, "Accelerated methods for deep reinforcement learning," Jan. 2019.

[5] . [Online]. Available: https://srdas.github.io/DLBook/GradientDescentTechniques.html.

[6] D. S. Hado van Hasselt Arthur Guez, "Deep reinforcement learning with double q-learning," Dec. 2015.