Solving the Lunar Lander Problem with Deep Q-learning

Huiwen Duan

I. Problem Statement

Lunar Lander is the problem to solve as well as the environment to work in. It has a 8 dimensional state space and 4 actions. Each state dimension is continuous and the actions are discrete. The four actions include ”do nothing”, ”fire main engine”, ”fire left orientation engine”, and ”fire right orientation” engine. The goal is train the lander agent so that it can decide the optimal action to go for at each state, in order to land within the defined target area. This requires an algorithm that can generate the optimal value for each action in any given state.

II. Methods

A. Deep Q-Learning (DQN)

DQN uses a function approximation to represent the Q table in traditional Q-learning. While training the agent, we maintain and use two sets of networks: Q network and target Q network. The target Q network is a copy of the Q network that's updated every 5 steps. The Q network take in the state as input and outputs the action values. When training the neural network, we use the Q-learning formula, as detailed in Fig 1, to calculate the expected Q-value.

$$\text{Set } y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$$

Fig. 1. Figure 1

We take the output from the target Q network corresponding to the actual action taken as the actual Q-value. The loss function for training the neural network is defined as the squared error between the expected Q value and the actual Q value. The network is trained through gradient descent which optimizes toward minimizing the loss.

The neural network I used have two linear layers, the first one is 64*64 and second one is 64*32. Given our problem statement, the input to this neural network is the 8 dimensions of the state and the output 4 values corresponding to the 4 actions.

Here I will explain more about using two sets of Q networks instead of one. During each time we update the neural network parameters to optimize $Q(s, a)$, we inevitably will alter the Q value for the next state and several other states. However, the parameter optimization itself requires the value of next state. This causes the training to be unstable. Therefore, we used two sets of Q-network: one for parameters update and the other for accessing the Q value of the next state. For every 5 steps taken, the target Q network gets synchronized with the main Q network

B. Experience Replay

One caveat with using Neural Networks to learn the action values is the correlation among a series of actions. For example in our Lunar Lander context, if an agent has experienced the states to the right of the target a lot and learned the best actions when dwelling in such states, it will have zero knowledge regarding what to do when in states that are to the left of the target. The agent will then have a greatly imbalanced memory which bias toward the actions chosen when facing the states that it has experienced more. Eventually it will cause the reinforcement learning to diverge or fluctuate.

Experience replay, introduced by Mnih et al's paper addresses this issue by recording every transition and providing a pool of transitions to to be sampled from. Specifically, after performing an action, information about the current state, action, reward, next state, and whether the episode is done is stored in experience. Then we sample from experience to train the neural network. This also makes more efficient use of the experience as we get to relive the rare states.

C. Using DQN and Experience Replay to Train the Agent

Here I would like to briefly go over the algorithm I implemented for training the agent to converge on the Lunar Lander problem. It is similar to the one introduced by Mnih et. al's paper. Firstly, I initialized a few containers and parameters: the experiences list to store all the transitions in a named tuple form, mini-batch size, scores list to store the score for each episodes, and steps list to store the number of steps needed to converge for each episode, Q network as the main neural network, and target Q network as the target network. For each episode, I reset the environment the get the initial state. For each step within an episode, the agent takes an action based on epsilon greedy strategy, stores the state, action, reward, next state, and done as a tuple in experiences. Then we train the main network by sampling a mini batch from experiences. Details on training the neural networks are covered in the Section A. For every 5 steps, the algorithm synchronizes the main Q network and target Q network. Convergence is defined as achieving 200+ score on average over 100 consecutive episodes.

Here I will expand more on using epsilon greedy strategy to choose actions. The algorithm generates a random number between 0 and 1. If this number is smaller than epsilon, then the agent takes a random action. If the number is bigger than epsilon, then the agent takes an action as recommended by the main Q network. The Q network takes input of the state information and output the values corresponding to each action. The action with the highest Q value is used.

Epsilon is an important hyper parameter in the algorithm as it controls the balance between exploration and exploitation. Exploitation means to make the best decision according to the current Q network. Exploration means explore unknown areas such as actions that have not been executed before. Therefore, the purpose of exploitation and exploration is to obtain a strategy with the highest long-term profit. This process may have a loss in short-term reward. If there is too much exploitation, then the model is easier to be trapped into the local optimum, but with too much exploration, the model can converge really slowly. I used epsilon decay so that the algorithm can explore more during the early learning episodes and exploit the visited territories more during the later state of training. Due to run time, I limit run time to only 1000 episodes and decay epsilon by 0.5% after each episode. Therefore throughout the training, epsilon is in a range from 1 to $1 * 0.995^9 99 = 0.007$. In the future, I would like to implement Reward Based Epsilon Decay as introduced in Maroti's 2019 paper, which adjust the epsilon dynamically in accordance to the reward accumulated.

III. Results

A. Scores When Training the Agent

After experimenting with different combinations of hyper parameters, my algorithm converged the fastest with $\gamma = 0.98$, learning rate = 0.001 and mini batch size = 64. It used 470 episodes to converge. Figure 2 detailed the score for each training episode until converge. Although there's variance among individual episodes, we can observe an upward trend of the scores, indicating the improvement of the agent. The scores started off negative, lingering around -300, meaning the agent was performing poorly without much training. Gradually, the scores climb up to around 0 after 200+ episodes, and eventually achieved an average of 200 scores.

However, when I test this agent in a new environment using greedy strategy, the average score is 194, lower than 200. I realized that achieving an average score of 200 does not necessarily mean the agent has learned the optimal policy and will perform ideally in a testing setting. Therefore, I extended the training time to 1000 episodes and recorded the scores in Figure 3. The seed and hyper parameters used are the same as in Figure 2, meaning that the agent reached an average score of 200 for the first time after around 470 episodes as well. It had 530 additional episodes to fine-tune the neural network. As observed in Figure 3, episode 0 to 470 follow a similar pattern as Figure 2. After episode 470, the scores stabilizes at around 220.

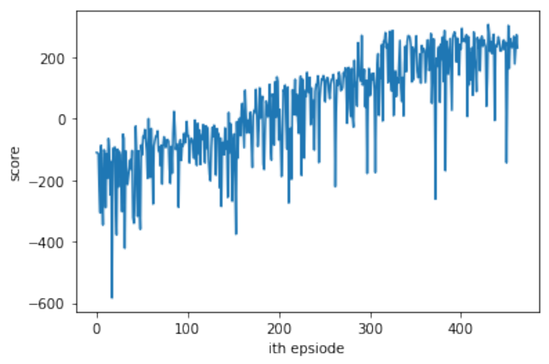

Fig. 2. Scores During Training, Stopping Immediately after Achieving an Average of 200 Over 100 Episodes

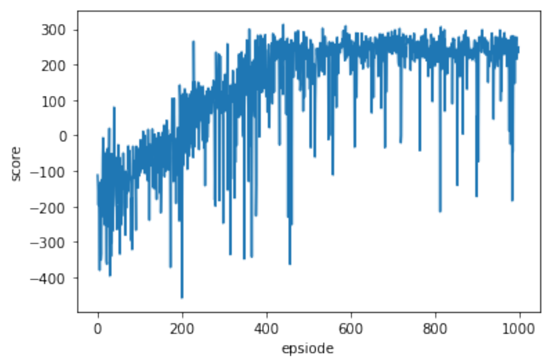

Fig. 3. Scores During Training, 1000 Episodes

B. Scores of a Trained Agent

Figure 4 shows the scores by the same agent after being trained 1000 episodes. It plays the same environment for 500 episodes using greedy strategy, meaning there are no more exploration and every action taken is based on the maximum action value. The trained agent is performing really well with all the scores above 50. The average score for the first 100 testing episodes is 204.

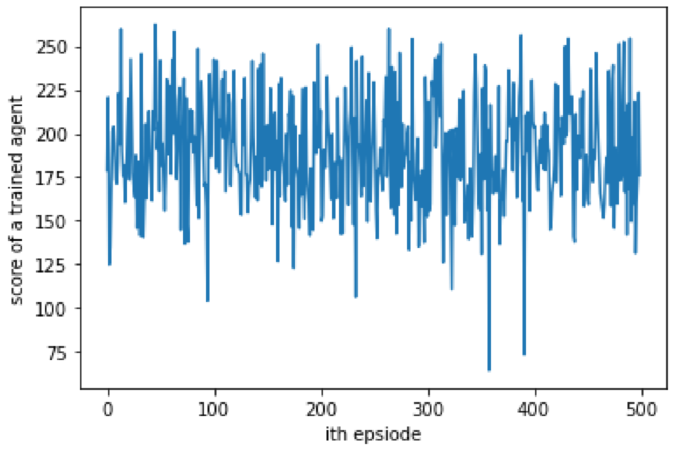

Fig. 4. Playing Trained Agent

C. Hyper parameters

There are several important hyper parameters when tuning the agent, as listed here:

- γ, Discount factor: A factor of 0 will make the agent consider only current rewards, while a factor closer to 1 means valuing past states.
- ϵ, Exploration constant: This is explained more in the method section.
- α, Learning rate: It is used to update neural network parameters. It dictates the importance of each gradient descent on moving the network parameters. A small learning rate will lead to a slow convergence while a big learning rate may cause the algorithm to miss the optimal parameters that can best minimize the loss and lead to a overshoot.
- Minibatch size: The size of the mini batch being used to train the neural network at each step. As Keskar et. al mentioned in their 2016 paper, a larger batch can cause a degradation in the quality of the model and ability to generalize, because "large batch trainins tend to converge to sharp minimizers of the training and testing functions". However, a smaller batch can take a long time to converge.
- Shape of the neural network. I fixed the network layers and empirically used 64*64 for the first layer and 64*32 for the second layer, since the office hour guest speaker recommended them to be close to the input and output dimension.
- C steps: it used for synchronizing the main network and the target network. The target network is updated after every c steps. It has a fixed value of 5.

I chose γ, α, and mini batch size to dive deeper into.

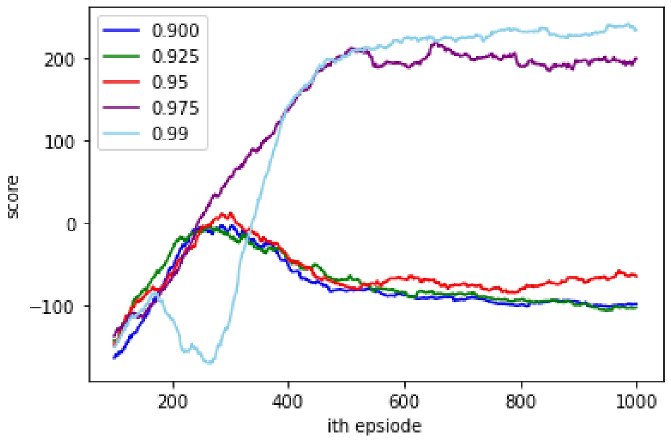

Fig. 5. Scores for Different Gamma Values

1) γ: I trained the agent with multiple gamma values including 0.9, 0.925, 0.950, 0.975, and 0.99. All other hyper parameters are held the same. The moving average for every 100 episodes for each of the gamma values are plotted in Figure 5. For γ values of 0.9, 0.925, and 0.95, the algorithm failed to converge within 1000 episodes. When γ equal to 0.975, the scores improves throughout and stabilized after approximately 550 episodes with an average score of around 200. The scores for γ = 0.99 is interesting. The score increased from episode 1 to episode 200 but sharply decreased afterwards until episode

260. From episode 260 to 450, the agent is picking up the speed of learning and the scores steadily increases. After episode 450, the score continues to improve but at a much slower speed. Having a large γ means putting a lot of emphasizes on past experiences when doing the Q learning. In our case, when $\gamma = 0.99$, if the agent has experienced some extremely bad states, these bad states will then be sampled for training but won't have meaningful contributions to parameters optimization. When the agent experiences new states that are not adjacent to these learned states, there hasn't been any learning around these states and therefore the agent will perform poorly, explaining the dip in figure 5 skyblue line.

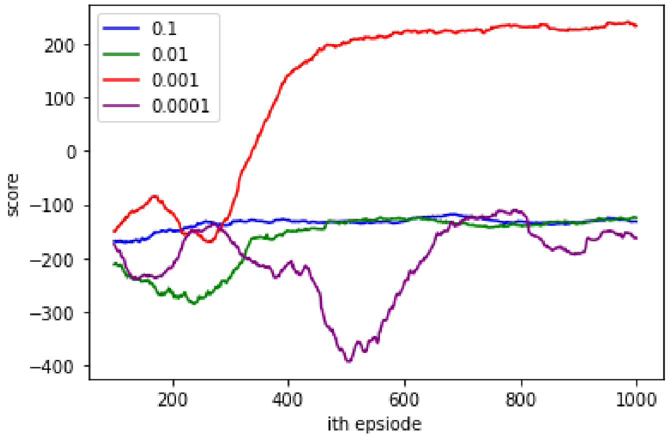

Fig. 6. Scores for Different Learning Rates

2) Learning Rate: Among the four learning rates- 0.1, 0.01, 0.001, 0.0001- I experimented with, only 0.001 resulted in algorithm convergence. The results are plotted in Figure 7. All other learning rates failed to converge. When doing this experiment, γ is fixed to be 0.99 whereas in the γ experiment the learning rate was fixed at 0.001, and therefore the red line in figure 6 is the same as the blue line in Figure 5. When γ is smaller than 0.001, there seems to be some learning taking place, causing the purple line to fluctuate and there is reason to believe that it will continue to learn after 1000 episodes until convergence. However, looking at the green and blue lines, where γ is bigger than 0.001 by order of magnitude, they quickly plateau to around score -150. This suggests that these learning rates may be too big, causing the gradient descent to overshoot and miss the optimal point. In the future, I want to experiment learning rates that are of the same order of magnitude as 0.001, meaning values such as 0.002, 0.003, etc.

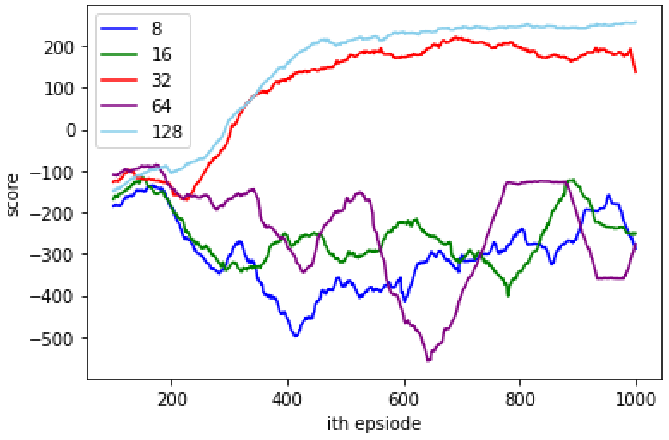

Fig. 7. Scores for Different Batch Sizes

3) Mini-batch Size: Batch size is the amount of transitions used to train the neural network at each step. The learning rate and γ used in this experiment are the best ones from the previous two experiments - 0.001 and 0.99 respectively. The results are plotted in Figure 7. With a batch size of 8, 16, 64, the agent failed to converge within 1000 episodes. However, batch size of 32 and 128 lead to successful convergence. There's no obvious linear relationship between batch size and convergence. This hyper parameters might be one of those which needs experimentation over random values to de decided upon.

IV. Discussions and Reflections

A. More on Hyper Parameters

As defined by Bergstra et al's 2012 paper, hyper parameter optimization is to identify a good value of hyper-parameters that can minimize generalization error. With limited prior of the hyper parameters themselves, it needs trial and error to identify the best ones. Moreover, hyper parameters often have interaction effects. For example, when α is 0.1, the best γ might be 0.8, but when α is 0.01, the best γ could be 0.9. Therefore, if run time permits, my next step is to experiment with every combination of hyper parameters and identify the best one.

B. Run Time Matters

The first agent I coded up with sub-optimal hyper parameters did not converge even after 10000 episodes with no signs of improvement, yet on Piazza multiple posts have reported convergence within 1000 episodes, which led me to suspect the validity of my implementation. I spent numerous hours trying to debug until I saw an inspiration on Piazza whose main idea is that not all algorithm converges at the same speed and my situation where it takes unreasonably long to converge could happen. Therefore, I shifted my focus from debugging to optimizing the hyper parameters and soon got a converging agent.

C. Fake Terminal States

Since I'm using a wrapped environment of lunar lander, for each episode it automatically terminates after 1000 steps and the boolean value marking whether the agent has finished will be set to True. This is a fake terminal state because without the artificial termination, the agent should continue to fire until landing. As mentioned in Figure 1, we treat terminal states differently when calculating the expected Q value to be used for gradient descent. With a fake terminal state, the algorithm will falsely zero out the value of the next state. To address this, I reversed the "done" flag whenever the agent unwillingly stops at a non-terminal state.

References

[1] Bergstra J, Bengio Y. Random search for hyper-parameter optimization[J]. Journal of Machine Learning Research, 2012, 13(1):281-305.

[2] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W., 2016. Openai gym. arXiv preprint arXiv:1606.01540.

[3] Keskar, N.S., Mudigere, D., Nocedal, J., Smelyanskiy, M. and Tang, P.T.P., 2016. On large-batch training for deep learning: Generalization gap and sharp minima. arXiv preprint arXiv:1609.04836.

[4] Maroti, A. RBED: Reward Based Epsilon Decay. arXiv preprint arXiv:1910.13701. 2019 Oct 30.

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. nature, 518(7540), pp.529-533.