Solving Open Gym Lunar Lander Problem using Deep Q Learning

Vibha Hegde
vhegde33@gatech.edu
Git Hash: `4eb36692d69ad14152d4aa54c7348b1e700fe4f0`

Abstract: Reinforcement learning is an area of machine learning that is used to solve general learning problems without the aid of a specific "teacher". Optimal control of any independent system is part of the learning problem where the system needs to navigate without any adverse effects optimally. Lunar Lander or Rocket trajectory optimization is one such optimal control problem. The lunar lander problem is unique in its design where the action space of the problem can be discrete, and the state space is continuous. Thus, the traditional reinforcement learning approaches like TD fail to solve the problem and reach an optimal solution. In this paper, a Deep Reinforcement Learning method called Deep Q learning (DQN) [1] is used to solve the Lunar Lander problem. The DQN methodology uses a non-linear function approximator to tackle the continuous state space. It also used the concept of experience replay buffer and target network to reduce the variance thus acting as a good choice to solve the Lunar Lander problem.

1. PROBLEM DESCRIPTION:

The Lunar Lander problem is an optimal control problem with the goal to land the rocket in a landing pad always located at the co-ordinates (0,0). The environment can have discrete actions as its considered optimal to fire the engine full throttle or turn it off. So, the actions available to the lander are do nothing, fire left, fire main engine and fire right engine. Each state is represented by the 8-tuple:

$(x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, legL, legR)$ -(1)

The horizontal and vertical positions of the lander is provided by x, y and the next two variables correspond to the horizontal and vertical speed. The $\theta, \dot{\theta}$ correspond to the angle and angular speed of the lander. Finally, the legL and legR are binary values to indicate whether the legs of the lander are touching the ground.

The reward system is configured in such a way that if the lander moves away from the pad, then it loses as much reward as it would have gained if it had moved towards the pad. If the lander lands accurately it gains +100 else -100 and for each leg-ground contact the lander is rewarded +10. If the contact is made and broken, then it is penalized -10. Firing main engine incurs -0.3 point while firing the orientation engine incurs -0.03 reward. Fuel is infinite and the game is considered as solved if the lander scores 200 or higher over 100 consecutive runs.

2. IMPLEMENTATION:

To solve the lunar lander problem, the algorithm used is Deep Q learning. Based on its performance over the continuous state space games like the Atari games, this seemed like a suitable choice to begin with.

2.1 Q – Learning:

it is a model- free algorithm in RL and uses an optimal policy from the dynamic environment to learn and take actions based on the optimal policy. In the Q learning the long-term reward for any state s and action a is given by the Q- function $Q(s, a) : S \times A \rightarrow R$ and the Q here defines the quality of the action taken from state s. Generally, in Q-Learning, the policy is to take the action that provides the maximum reward. Using the Q-learning method with the Bellman equation, i.e using the discounted long-term reward γ provides us the equation 2.

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a) \qquad \text{-(2)}$$

However, to enable the agent to converge over the new and old learning, we can use a learning rate α and reduce the noise from the environment. Thus, the Q- learning equation is shown in equation 3.

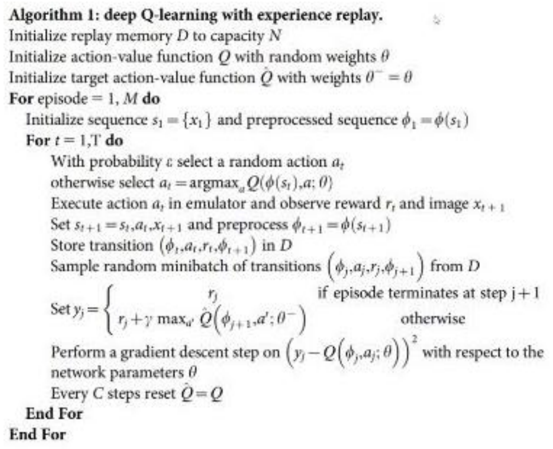
$$Q(s_t, a_t) = (1\text{-}\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a)) \qquad \text{-(3)}$$

This is known a Tabular – Q learning. The drawback of this method is that it handles continuous state spaces in the problem poorly. To overcome this issue a non-linear function approximation needs to be used and is provided in the Deep Q Learning algorithm.

2.2 Deep Q Learning:

To overcome the issue presented by Q-Learning, a neural network function approximator is used, thus modifying the Q Learning algorithm into Deep Q Learning. In addition to having the general Q Function into the form of neural network, the algorithm introduces a variety of parameters making the algorithm a little bit complex. Since the goal of the Lander is to gain more rewards, the weights of Q function need to be calculated accurately. To measure the accuracy of the network, a loss function named MSE loss is used to calculate the difference between the predicted weight in the Q network and Target Q network, making this problem like that of a regression problem.

The significance of the DQN algorithm is the usage of Experience Replay buffer or the memory buffer. This is introduced to stabilize the neural network. Using the traditional approach of using current state, action pair to predict the future Q function value leads to large variance and it is observed that the error gets propagated quickly into the further steps. To reduce this a list called the replay buffer is used to store the historical values, which is then sampled as a batch and introduced for training. This immediately reduces the unwanted feedback loops and reduces the divergence. This complements the use of Q-learning as the usage of experience replay requires the use of off policy algorithm. Sampling randomly from the last N historical values from the buffer reduces the oscillations and thus increases the stability of the model. The algorithm from the paper is provided for reference in Figure 1.



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$
        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 $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

Figure 1: DQN Algorithm [1]

The second important modification in DQN is the use of a target network. Two different networks are used during the execution. The first is a local network on which the updates are carried out. After every C step the weights from the local network is updated to the target network. This makes the oscillations and divergence more unlikely.

3. Experimental Setup:

For the experiment, A feed forward neural network with the two linear hidden layers with the dimensions 256/128. The input dimensions are the state values which at any given point is 8 from the problem description. The output required is the value of each action given the state. Thus, the output would be 4. Hence, the model is of the dimensions 256 × 8 in the input layer, with a dropout of 0.5. The dropout is used to prevent the overfitting for the training data. The hidden layer is of the dimension: 256 × 128 and the output layer is of the dimension 128 × 4, for the four actions allowed in the problem.

A non- linear activation function is used at the input and first hidden layer named RELU. This function returns the maximum positive value of the input as shown in equation 4:

$f(x) = relu(x) = max(0, x)$ -(4)

Note: To arrive to the conclusion of using a simple feed forward network a bit of trial and error method was used. Adding more hidden layers did not produce any significant change in the outcome. This is because the network does not do any complex computation or use convolution or pooling layers. However, the depth of the network provided a significant improvement where the value 64 × 32-dimension hidden layer would provide faster execution and quick divergence. For the Loss function selection, the MSE loss seemed the most appropriate selection due to the requirement mentioned in the DQN paper [1]. As for the optimizer, Adam was used as the default optimizer as it performed well even with very small α value.

4. INITIAL EXPERIMENTATION AND RESULTS:

Using the default parameters from the paper [1], initial tests were run for 1000 episodes and 1000-time steps each. Each trained agent is tested for 200 episodes.

Epsilon Values: From the concept of epsilon greedy policy[2] for Q- Learning, the ϵ_{start} , ϵ_{End} , ϵ_{Decay} parameters need to be selected. The greedy policy selects a random action every ϵ time and the action that produces maximum value for every $(1 - \epsilon)$ times. This is to ensure that the agent in the beginning of the game tries to explore more of the possible actions rather than learn. This leads to three basic intuitions:

- Initially the ϵ_{start} should be high, so that the agent can explore more.
- Gradually, the ϵ should decrease, enabling the agent to exploit the best possible actions more. Each time epsilon is decreased by the value ϵ_{Decay}.
- Finally, the agent should continue using the action that is learnt by the agent making the ϵ_{End} very small.

Thus, for the initial experimentation, ϵ_{start} is set to 1, ϵ_{End} is set to 0 and decay value, ϵ_{Decay} is set to 0.99.

Learning Rate: Learning rate is another important parameter in Q- learning as shown in equation 3. In this experiment, the learning rate is utilized by the optimizer in the network (Adam). To start with, α is set to 0.00025

Other parameters: There are few parameters specific to the DQN algorithm, such as Replay buffer size, C (Update frequency), and mini batch size which are set to 1000000, 20 and 32 from the paper.

4.1 Results:

The initial results weren't as good, because the agent never learnt to cross over to the positive rewards. The reward per episode in training is shown in the Figure 2. This shows that initially the agent executes the randomly and slowly learns to take optimal actions. However, after a certain score of +72, the agent stops learning and remains in that position.

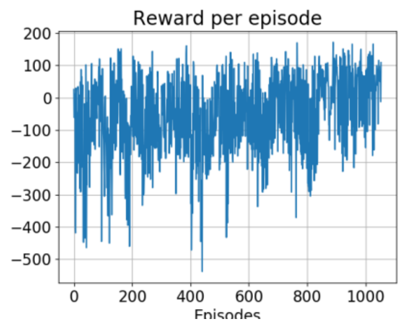


Figure 2: Initial training

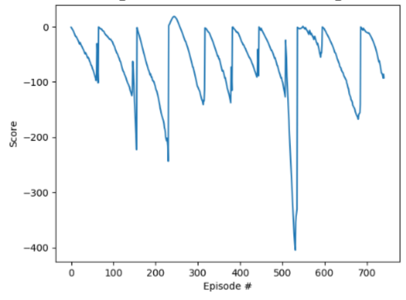


Figure 3: Testing a trained agent

The test results of the trained agent are also low, where each time the agent crashes rather than land as shown in Figure 3.

4.2 Hyperparameter Tuning:

For the improvement of the agent, through experimentation the following parameters were chosen along with increase in the episodes to 2000.

4.2.1 Update frequency:

It is clear from the beginning that the agent learns and updates the target values to train. So the update frequency acts as an important feature for optimization. To experiment 4 different frequencies were selected: 4, 10, 15, 20. The Update frequency in the paper is quite high and this is because the original paper deals with very high input value (frames about a million). Thus, it makes sense to keep it high.

Analysis: The assumption that the higher value of update frequency yields better results is wrong from the experiments this is because the learning happens over a different set of batches for larger C.

Disadvantage: A problem with the larger C is that a lot of randomness from the initial episodes would not be considered for the execution and skipping good runs become an issue here. Meanwhile a lower C increases the execution time and oscillations increase, resulting in divergence rapidly.

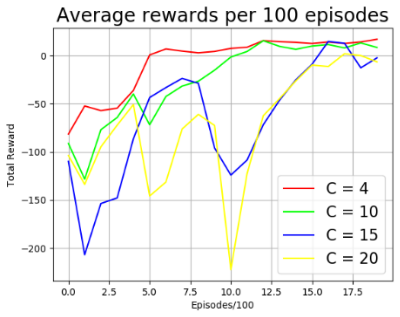


Figure 4: Total rewards in training for the different update frequencies C.

Remark: A moderately high C with prioritized Experience Replay would solve the issue of completely skipping the initial runs and would make the system more robust.

4.2.2 Discount Factor - Gamma

Discount Factor (γ) is the long-term rewards that need to be considered for the present state. Generally, it is supposed to be closer to 1. Since the model is highly sensitive to the parameter, a test was carried out with different values for gamma: 1, 0.999, 0.998, 0.995, 0.99 as shown in Figure 5.

Analysis: The initial assumption that the γ=0.999 should perform better does not hold good as γ = 0.99 provides a better result. Intuitively, this is correct because providing γ=1 or very close to 1 would mean that all the long-terms rewards are as important as the current reward and thus the current reward loses the optimality that it provides. Also, γ = 0.99 is quite stable and converges well.

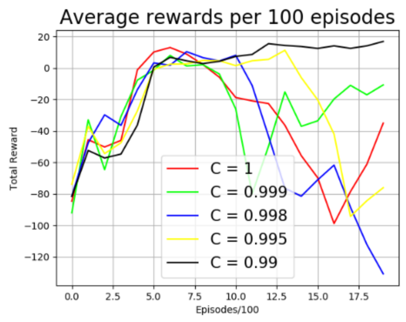


Figure 5: Discount Factor Tuning

4.2.3 Epsilon Greedy Parameters:

Epsilon start: From section 4, the experiment needs to have a high value of ϵ_{start}. So various values of $\epsilon_{start}$are used for training and testing as shown in Figure 6:

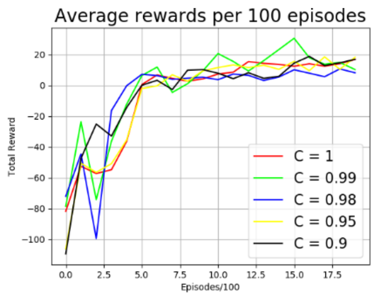


Figure 6: ϵ_{start} Training values

Analysis: Having a higher ϵ_{start} makes sense and the best mean is obtained for ϵ_{start}=1. However, in terms of stability and quicker convergence, ϵ_{start} = 0.9 serves best. This is because the decay is calculated based on the initial value. Thus, if the initial value is high, then the decay is slower, allowing more randomness.

Epsilon End: The minimum Epsilon value that can be used to for the policy after which the agent should select the optimal action is determined by ϵ_{End}.

Analysis: Having ϵ_{End}= 0 seems like the better solution initially and 0.01 is suggested from the paper [1]. However, ϵ_{End} = 0.05 provides quicker convergence and is stable over the increased training episodes as well.

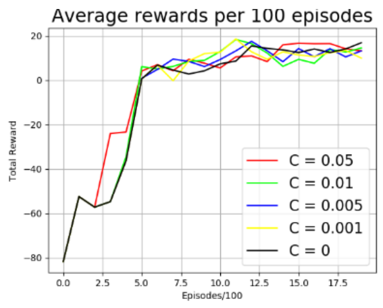


Figure 7: ϵ_{End}tuning

Epsilon Decay: The decay parameter decides how quickly the agent moves from exploration to exploitation of optimal actions. If the exploration is more, then the number of episodes required for the convergence is more. If the exploitation is more, then the agent doesn't learn various actions and remains in a sub-optimal state. Hence an experiment was conducted with different parameters for the decay as shown in Figure 8. From the experiment it is clear that the best value is 0.995.

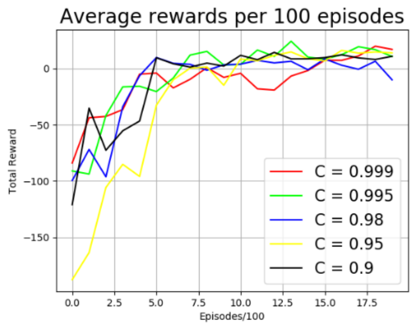


Figure 8: Epsilon Decay Tuning

4.2.4 Final Results:

With the modifications implemented from the above parameters, the agent was trained and tested again to obtain the following results shown in Figure 9a and 9b. From the results, the agent initially explores randomly and later learns to fetch positive rewards.

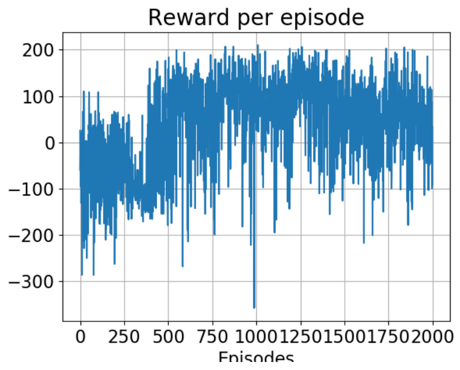


Figure 9a: Training modified agent

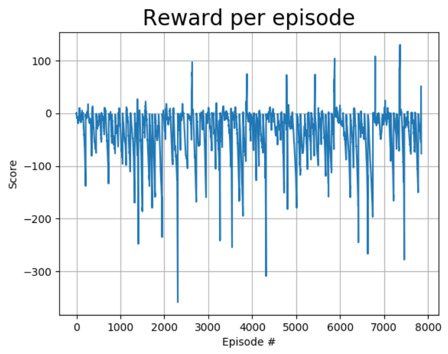


Figure 9b: Testing Modified agent

However, there are two problems:

- The agent "Forgets" everything it has learnt at one point and choses to randomly crash.
- The agent after learning seems to choose a sub-optimal approach of hovering (get minimal positive reward), rather than land properly.

5. CONCLUSION AND FUTURE WORK:

From the above results the agent is learning to play the Lunar Lander, however, the agent is unable to solve the environment. Two major things that can be explored to solve the issue would be: 1. Prioritized Experience replay buffer. 2. A Different epsilon policy to break from the constant hovering.

6. REFERENCES

[1] Mnih, V., Kavukcuoglu, K., Silver, D. *et al.* Human-level control through deep reinforcement learning. *Nature* **518,** 529–533 (2015). https://doi.org/10.1038/nature14236

[2] Sutton, Richard S., and Andrew G. Barto. *Introduction to reinforcement learning*. Vol. 135. Cambridge: MIT press, 1998.