Solving Lunar Lander Using DQN

Abstract This paper presents a solution to the Lunar Lander environment using DQN for training.

1. Introduction

The problem we are trying to solve is to successfully land a space craft on the surface of the moon within a given landing area. Conceptually, this seems quite trivial to us humans. But it can be very challenging to an RL agent which, unlike humans, must learn all necessary concepts from scratch: gravity, thrust, engines, goal area, directions, and so on. On the other hand, humans can bootstrap from the knowledge and experiences they have built over their lives from playing games and interacting with the world.

The state at each step is represented by an 8-deimensional vector: $(x, y, \dot{x}, \theta, \dot{\theta}, left_{leg}, right_{leg})$, whose components are the lander's coordinates, angle, angular velocity, and two booleans indicating if the left leg and right leg are touching the surface. There are 4 discrete actions available: do nothing, fire left engine, fire right engine, and fire main engine. We get rewards for getting closer to the landing pad, moving slower upon landing, and having both legs touch the ground. Firing the engine and crashing costs points. An episode ends if the lander crashes or make a successful landing. An episode is considered a solution if it scores at least 200 points. Full details about the environment can be found at [1].

Since no model of the environment is available, a model-free RL algorithm (such as Sarsa or Q-learning) must be used to train the agent to estimate the action-value function. $Q(s, a)$ gives the expected return from state s after taking action a. Since this environment does not have discrete and finite number of states, we must use a function approximator to estimate the action-value function, Q. In addition, since the environment is complex, we will use a non-linear function approximator: a neural network. We will model our solution on the celebrated DQN paper [2] from DeepMind, that popularized combining neural networks with Q-learning.

Q-learning is an off-policy algorithm that makes the following learning update at each step:

$$w_{t+1} = w_t + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a, w_t) - Q(S_t, A_t, w_t)]\nabla Q(S_t, A_t, w_t) \qquad (1)$$

where w_t is the vector of the network's weights, A_t is the action selected at time step t, and S_t and S_{t+1} are the state vectors at time t and t+1 respectively.

2. DQN Lunar Lander

A Deep Neural Network used to estimate Q-values is called a *Deep Q-Network (DQN)* [2] and using a DQN for Q-learning is called *Deep Q-Learning*. We use this approach to train the agent for Lunar Lander. The pseudocode for the algorithm used to train the agent is similar to that in the DQN paper [2]. An alternative way to train the agent would be to use policy gradient methods but they were not used due to the lack of experience and familiarity with those algorithms.

Like in the DQN paper, a few modifications to the basic Q-learning update procedure in Eq (1) are made:

1) We use a method called experience replay, that accumulates experiences in a replay memory. The above learning update is then made using a mini batch of experiences sampled uniformly from this buffer. This makes more efficient use of data, and makes the mini batch of samples more independent, and identically distributed making stochastic gradient descent more stable.
2) In the basic deep Q-learning algorithm, the model is used both to make predictions and to set its own targets. This can lead to oscillations and/or divergence. To address this problem, we use two DQNs instead of one: the first is the *online model*, which learns at each step and is used to move the agent around, and the other is a *target model* used only to define the targets. The target model is updated after every learning update towards the online model by doing a weighted average of its existing weights and the new online model weights. (target_model = τ online_model + (1- τ)target_model). This differs slightly from the DQN paper, where the target model was updated every C steps; we chose this instead since a continuous update ensures smoother learning as weights are smoothly updated, instead of making hard jumps.
3) Like in the DQN paper [2], we do a learning update every 4 steps to reduce computational load.

We now present the DQN and the hyperparameters used to train an agent that solves the environment:

- DQN: The neural network takes the state vector as input and has 4 outputs: an action-value estimate for each action. It has 3 fully-connected layers with 64, 64, and 4 neurons respectively. ReLU activations was used in the first two layers; no activation function was used in the last layer.
- Loss function: mean squared error.
- Batch size: 64
- Optimizer: Adam
- Learning rate: 5e-4
- τ: 0.001
- ϵ-greedy policy was used during training. $\epsilon = 1$ at the start of training and it decayed by 0.995 every episode.
- Replay buffer size = 50,000. It was implemented as a circular queue.
- Convergence criterion: when the average of the total reward per episode over the last 100 episodes is greater than 215.

PyTorch was used as the framework and training was done on Google Colab, with a CPU runtime. The agent converged in 800 episodes, and that took about 1 hour.

Figure 1 shows the total reward per episode during training. We can see that the results are quite noisy from one episode to the next, this seems to be a hallmark of using function approximation: when the agent learns for certain states, the weight changes negatively impact other states, causing wild fluctuations on consecutive episode. However, on average, there was continual improvement over the 800 episodes.

Figure 2 shows the total reward per episode using the fully trained agent with greedy policy. On most of the episodes (92 of the 100), the agent earned a total reward of at least 200, showing that the agent was near-optimal. There was however some stochasticity from one episode to the next, and this can be explained due to the random start state for each episode. The agent may not have gotten a reward of 200 or more on all episodes due to the fact it has sub-optimal values for some state-actions; particularly for states not visited or not visited often during training. Training the agent for much longer might have fixed this issue, but it is also hard to say how long to keep the agent training before hand.

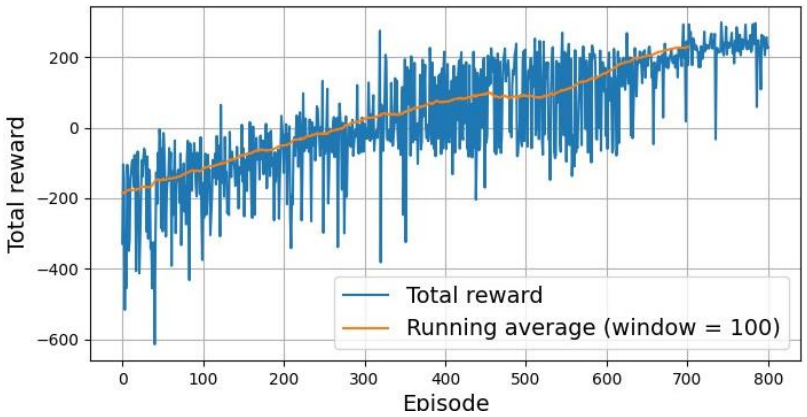

Figure 1: Total reward per episode during training. Agent converges on episode 800, when the average reward over last 100 episodes is greater than 215.

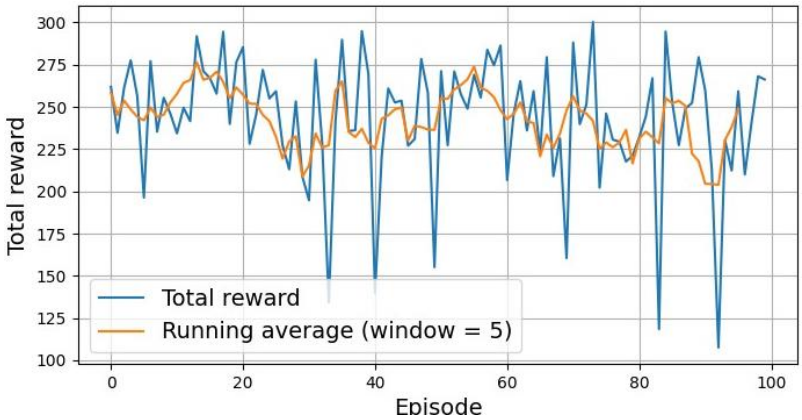

Figure 2: Total reward per episode over 100 episodes using the fully trained agent with optimal policy.

3. Hyperparameter exploration

We will now discuss how we arrived at the hyperparameters presented above for the agent that solved the environment. Hyperparameter selection was the most challenging part of the project. The biggest challenges were: (1) not knowing what values to begin with when starting exploration; (2) managing the subtle interactions between the hyperparameters, for example, an optimal learning rate for one optimizer was not the same for another optimizer; (3) the instability of deep reinforcement learning compared to supervised learning.

For some hyperparameters, I used intuition and general best-practices, and for others I simply did a grid search until the optimal agent was found. This process did take a lot of time, patience, perseverance, and luck. When testing each hyperparameter, the stopping criterion was either: the average of the total reward per episode over the last 100 episodes was greater than 215, or the number of episodes during training reached 2000 (in which case the given hyperparameter under test was not optimal). On average, each run for testing an hyperparameter lasted about 1 to 2 hours and was the biggest limiting factor.

Due to compute constraints, I used intuition to choose and fix a few hyperparameters during search:

- DQN model architecture: I started and remained with a 3-layer fully connected network with 64, 64, and 4 neurons respectively. While the number of neurons in the last layer was fixed at 4, I arbitrarily chose 64 for the number of neurons in the first two layers. My other option was using 32 neurons, but after tuning other hyperparameters with the number of neurons fixed to 64, I did not experiment with 32 due to limited compute.
- Activation function, batch size, and optimizer: For these three hyperparameters, I relied on best practice, popularity, and my experience with supervised learning to choose ReLU, 64, and Adam respectively.
- τ: 0.001. Intuitively, a τ of 1, would update the target model to be equivalent to the online model after every step. Whereas a τ of $\frac{1}{1000}$ would update the target model to be equivalent to the online model (assuming a static online model) after 1000 steps. Each episode is on average 600 steps, and updates were done every 4 steps, so 150 times per episode; thus, updating the target model to be equivalent to the online model every 5 to 6 episodes seemed reasonable.

Figure 3 shows the running average of total reward per episode over the last 100 episodes for different values of the discounting-factor, γ. The running average is used to remove the noise and thus making it easier to see the trends. For lower values of γ (0.90, 0.95, 0. 97), the agent failed to converge over 2000 episodes. The agent converged for both γ equal to 0.98 and 0.99, though it performed slightly better for γ=0.98. I arbitrarily chose γ=0.99 for the final agent. For lower values of γ, I suspect that the agent did not converge as, it prioritizes longer-term rewards less, and optimized for short-term rewards. In the Lunar Lander environment, the long-term reward is landing successfully. However, there are many short-term rewards the agent can get before landing, such as minimizing fuel usage or staying upright, and the agent may have optimized for these instead of learning how to land successfully.

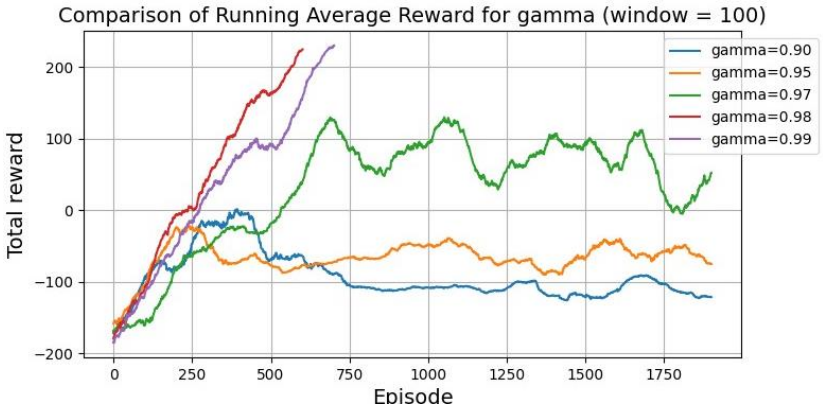

Figure 3: The running average of total reward per episode, with window size of 100, for various values of γ. α=5e-4, buffer_size=50,000, ϵ-decay=0.995

Figure 4 shows a similar graph, but this time comparing the learning rate, α. The largest value, α=5e-2 did not lead to convergence after 2000 episodes. This makes sense as with a larger α, the weights of the neural network can be updated too much in a single training step. This can cause the training process to oscillate and make it difficult for the agent to converge on a good solution. For the smallest value, α=1e-4, the agent converged but after 1800 episodes, as the learning made slow and steady progress. The best values of α, were intermediate values: α=5e-4 and α=5e-3. α=5e-4 led to slightly faster convergence and was used for the final agent.

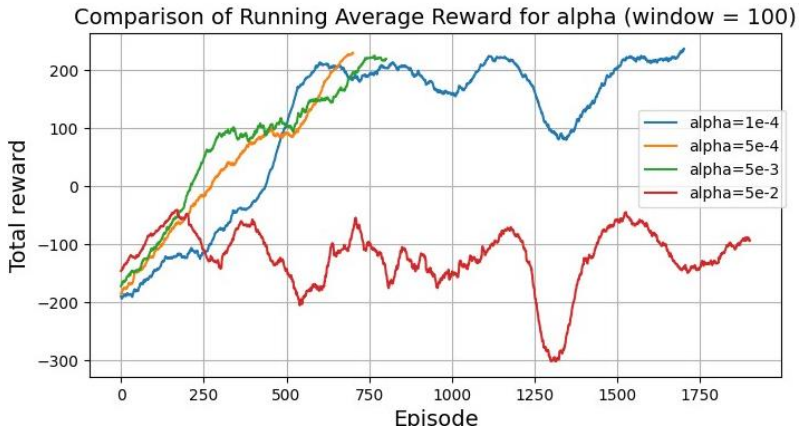

Figure 4: The running average of total reward per episode, with window size of 100, for various values of α. γ=0.99, buffer_size=50,000, ϵ-decay=0.995

Another reason smaller learning rates performed better may be due to better stability during exploration: when an agent is first exploring its environment, it takes a lot of random actions (due to high ϵ) and experiences a lot of different rewards. A large learning rate can be too sensitive to these initial rewards and lead to the agent learning sub-optimal actions. A smaller learning rate helps to stabilize the learning process during this exploration phase.

Figure 5 shows a similar graph as above, but this time comparing the size of the replay memory buffer. For the smallest buffer size (1000 experiences), the agent took the longest to converge, converging after 1100 episodes. The agent converged the fastest (700 episodes) for the largest buffer size of 100,000. And the general trend is larger the buffer size, faster the convergence. However, the gain in speed from increasing the buffer size from 10,000 to 100,000 is only about 200 episodes faster. For the final agent, I chose an intermediate value of 50,000, considering computational costs.

In the context of the DQN algorithm, it makes sense that a larger buffer leads to faster convergence for two reasons: (1) larger diversity of experiences: with a larger buffer, the agent samples from a wider range of past experiences ensuring it learns at more states; (2) reduced correlation between experiences: more i.i.d samples helps gradient descent converge faster and increases stabiltly.

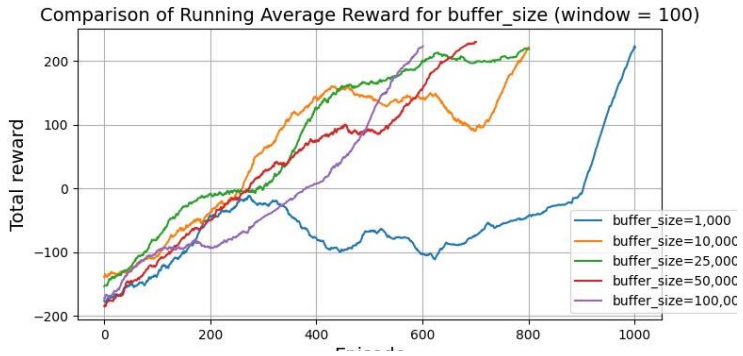

Figure 5: The running average of total reward per episode, with window size of 100, for various values buffer sizes. γ=0.99, α=5e-4, ϵ-decay=0.995

Figure 6 shows comparison of different ϵ-decay rates; ϵ was decayed by multiplying it by a number less than 1 after every episode. Different values of decay did not cause a significant difference in the convergence rate. Overall, the trend was that the faster ϵ decayed, the slower the convergence. This is because, the faster decay mean that the agent spent less time exploring and thus experienced less states, and consequently took longer to learn an optimal policy. The fastest ϵ-decay was 0.90, which made the agent converge on episode 1100; and the slowest ϵ-decay at 0.995 made the agent to converge fastest in 800 episodes.

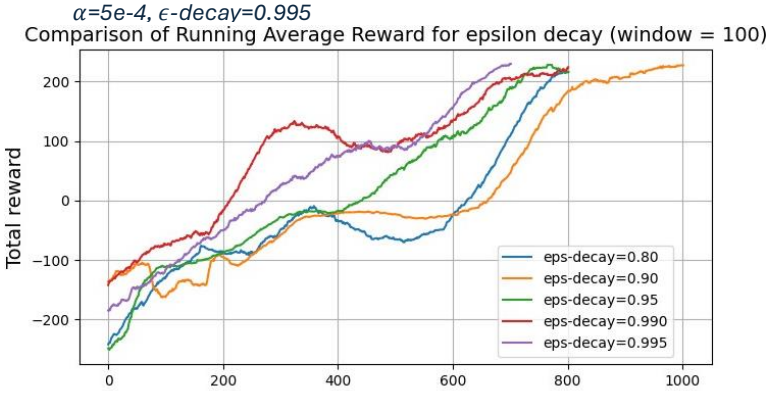

Figure 6: The running average of total reward per episode, with window size of 100, for various ϵ-decay values. γ=0.99, α=5e-4

Figure 7 shows the agent with the optimal hyperparameters trained using three different environment random seeds. Treating the random seed as a hyperparameter, we can see that Deep RL is relatively more unstable than supervised learning. The three different runs, with all the other hyperparameters fixed, took 800, 950, and 1050 episodes respectively to converge. This may also be due to inherent stochsticity of the environment, so the agent never sees the same training data every run.

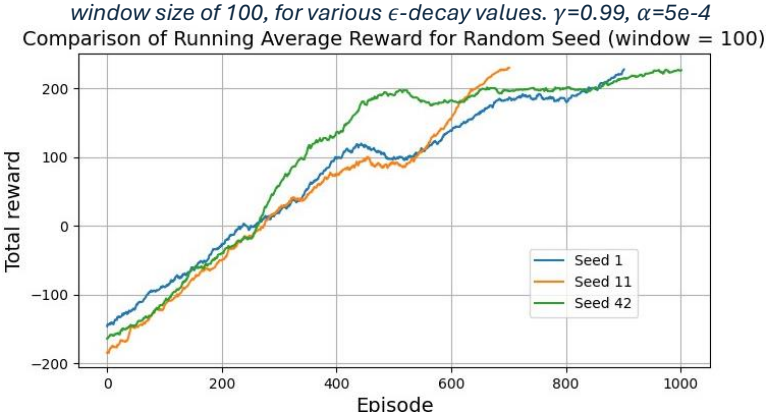

Figure 7: The agent with the optimal hyperparameters trained with three different hyperparameter seeds.

4. Conclusion

In this project, I have successfully trained an RL agent to solve the Lunar Lander environment using a modified version Q-learning as the algorithm and DQN as the Q-function approximator. I have also explored the effect of various hyperparameters on the convergence time. If I had more time, I would have done the following: (1) repeat each run several times with the same hyperparameters and average the results to reduce the effect of stochasticity; (2) explore the effect of other hyperparameter on convergence, i.e., the batch size, optimizer, and neural network size.

References

[1] Greg Brockman, et al. *OpenAI Gym*. (2016)

[2] V. Mnih, et al. *Playing Atari with deep reinforcement learning*. (2013)

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2020