CS7642 Project 2

Lunar Lander

Wei En Chen (Git hash `2c71657e4d0e0e6f4b2fd1b14d054ef66a1c4340`)

***Abstract* – This paper demonstrates the theory, implementation, and hyperparameter analysis of deep Q network (DQN) on lunar lander challenge. It shows the agent learns well and is able to successfully land the lander with 200+ scores average over 100 episodes. Effects of 3 hyperparameters are also discussed in details.**

I. INTRODUCTION

In the Lunar Lander v2 challenge proposed by OpenAI gym [1], the goal is to land a lander between two flags on the ground properly without crashing. The environment is 2D and there are 4 actions in the action space: do nothing, left orientation thrust, main engine, and right orientation thrust. In the state/observation space, it consists of 6 scalar and 2 boolean values: x pos, y pos, x velocity, y velocity, angle, angular velocity, and the last two are boolean, which represent left and right legs touching the ground.

An important feature to help an agent learn in reinforcement learning (RL) is reward shaping [2]. It can be critical to how fast a good solution, if not optimal, converges. In the lunar lander challenge, there are positive rewards for navigating the lander from top to the landing area, legs in contact with the ground, etc. There are negative rewards for crashing, leaving the landing area, and firing any thrust. The Gym library website provides more quantitative details on this challenge [1].

In this report, I will devise a RL algorithm to solve this challenge and discuss the algorithm in details.

II. Q-LEARNING

Q-learning is known as one of the major breakthroughs in RL early days. It uses an off-policy TD control algorithm [3]. Its Q value update rule is defined as [3]:

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

Off-policy means the agent may act based on different policy from the updated policy from Eqn. 1. From the equation, the Q value is updated using maximum evaluated value from all actions in the next state, S_{t+1}, while the agent takes ε-greedy action derived from Q at state S_{t+1}.

On the other hand, on-policy like SARSA updates Q values based on agent's action at state S_{t+1} [3]. Both basic forms require a temporary storage (Q-table) for state-action pair space to update Q values. If there are 100 states (a discretized 10 by 10 grid) and 4 actions, there are 400 unique state-action pairs. The discretized world is an easy and effective method for Q-learning if the Q-table size is reasonable. The space complexity of the Q-table is O(n^m) where n is number of discretization per dimension (state), and m is number of dimensions. Actions are excluded here because it does not contribute as much as n^m. If we use the lunar lander states as an example to estimate our Q-table size, we can use n=10 which isn't much for each state dimension, and m is 6 (excluding the 2 boolean values and action space). We have 1 million states in the Q-table. If n=20, there are 64 million states. We can see how easy the size of Q-table can blow up and it will be very difficult for agents to optimize a solution because there are exponentially more state-action pairs to explore, if not limited by size of physical memory storage. To mitigate this downfall, a deep q-learning (DQN) was introduced by Mnih et al. [4]. DQN is the proposed algorithm to solve lunar lander challenge in this report, with some tweaks.

III. DEEP Q NETWORK (DQN)

Instead of a discretized Q-table, a neural network is utilized to take all elements in the state space as input and applies non-linear functions to output values that match the action space dimension. The algorithm described in [4] uses two Q-networks, one for updating/learning, and the other is treated as "stationary" (or target) network for loss functions and weights update in neural networks. The "stationary" network is required because in supervised learning, the loss function is evaluated against labels (ground truth) which are considered fixed. However, in RL there are no labeled ground truths so Bellman equation is applied to approximate Q*. Due to the iterative nature of Bellman's equation, approximated Q* is a moving target. This will essentially render supervised learning useless when labels keep changing. The "stationary" network is fixed for a number of steps for the learn q-network to evaluate loss functions and update weights. This tends to stabilize the weight update process [4]. From paper [4], the error function formula is essentially Eqn. 2

$$E_i(\omega_i) = Q_{target}(s', a', \omega_{i-1}) - Q_{learner}(s, a, \omega_i)$$

$$Q_{target}(s', a', \omega_{i-1}) = r + \Upsilon(\max_{a'} Q(s', a', \omega_{i-1})) \quad (2)$$

Subscript i indicates the time step, and ω is the weights. This formula is explaining that the error we want to minimize throughout the training process is the difference between target and learner q-networks. Q-target is evaluated with ω_{i-1}, meaning previous iteration weights which are stationary for at least a number of steps. Q-target is evaluated at state prime and a maximum value action is chosen.

Another important implementation in the paper [4] is the experience replay. Whenever the agent makes a step, it stores relevant information such as last state, current state, action taken, and reward for training later in batches. Since these are past experiences, experience replay works well with Q-learning, which uses off-policy approach. For on-policy approach like SARSA, Q-target has to be updated from taking ε-greedy action current policy. This prevents Q-target from using any other policy.

At the end of Algorithm 1 in Mnih's paper [4], it updates Q-target network weights to the same as Q-learner network weights every C steps. The reason is explained above that "stationary" target is important for neural network weights to follow gradient descent in a stable manner.

IV. IMPLEMENTATION

The core algorithm to solve lunar lander challenge is DQN with experience replay [4][5], plus some tweaks. Pytorch is imported to handle neural network forward/backward propagations including loss function, optimizer and weight updates [6]. The algorithms are first written below, and explanation follows:

Algorithm 1: Initialization

```
Initialize $Q_{critic}$ network
Initialize $Q_{target}$ network
Initialize M for replay memory with fixed capacity C
Initialize ε, τ, ϒ
```

Algorithm 2: ε-greedy action

```
**func** epsilon_greedy(s, ε)
  **if** random_number() < ε **then**
    **return** random action
  **else**
    **return** greedy action from $Q_{critic}$ network
  **end if**
**end func**
```

Algorithm 3: Custom DQN with Experience Replay

```
**for** episode in EPISODES:
  Reset env ψ to initial condition
  **for** t in MAX_STEPS:
    action a = epsilon_greedy(s, ε)
    Agent step forward in ψ with action → (s', r, done)
    Agent adds experience tuple to M: (s, a, r, s', done)
    **if** size(M) ≥ nn training batch size **then**
      grab batch $(s_b, a_b, r_b, s'_b, done_b)$ from M
      $Val_{critic}$= inference on $Q_{critic}(s_b, a_b)$
      $Val_{max\_target}$=max(inference on $Q_{target}(s'_b)$)
      $Val_{target} = r_b + \Upsilon \times Val_{max\_target} \times (1 - done_b)$
      $L_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Val_{critic_i} - Val_{target_i})^2$
      Perform back propagation on $L_{MSE}$
      Adam optimizer to update $Q_{critic}$ weights (θ)
      $\theta_{target} \leftarrow \tau \times \theta_{critic} + (1 - \tau) \times \theta_{target}$
    **end if**
    s = s'
    $\varepsilon \leftarrow \max(\varepsilon \times \varepsilon_{decay}, \varepsilon_{min})$
    **if** done **then**
      exit this loop
    **end if**
  **end for**
  $\tau \leftarrow \max(\tau \times \tau_{decay}, \tau_{min})$
**end for**
```

Algorithm 1 is simply initializing the two Q networks, replay memory bank, and some parameters for the subsequent algorithms. Each Q network consists of 3 hidden layers. The input layer takes the state space, row

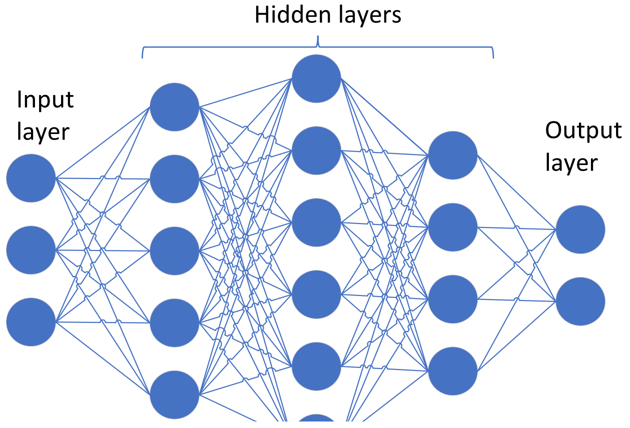

Figure 1 - Schematic of a neural network

vector of 8 elements, as the input. The output layer is in the shape of action space, row vector of 4 elements, a ranking of 4 actions with real numbers. The neural network is fully connected between any adjacent layers. The schematic neural network is shown in Figure 1.

Algorithm 2 is a function to determine if the agent chooses the best available action or a random one. The decision is based on the value of ε. This value decays as agent moves and tries to learn from the environment. To find greedy action, simply do an inference on Q_{critic}, and find the argmax.

Algorithm 3 is the core of this project, custom DQN with experience replay. The concept follows Mnih's paper [4] and pytorch DQN tutorial [5]. There are 2 for loops in this algorithm. The outer one controls the number of times the lunar lander agent will attempt to play the game by landing. The inner loop controls the maximum number of steps the agent takes to land the lander. In each episode, epsilon greedy action is evaluated at current state s. The agent then steps forward with the epsilon greedy action, which then results in s' and reward r. Every step is recorded in the replay memory M. If M is full, the oldest experience is popped.

Once M has enough experience for at least a batch, a random batch is extracted from M for two Q networks to learn. Q_{critic} is the network that learns to optimize. Q_{target} is the "stationary" network to calculate loss. Val_{critic} represents the value returned from making inference on Q_{critic} with current state and action. Val_{target} is the Q value update by finding maximum action value from Q_{target} at state prime. Once the two values Val_{critic} and Val_{target} are calculated, I use MSE to calculate the loss between the two values, and back propagation is performed on the MSE loss function. Adam optimizer is utilized instead of the simple SGD because Adam includes adaptive learning rates which helps a lot in this optimization task. Q_{critic} weights are then updated from the Adam optimizer.

For the Q_{target} update, instead of updating the weights every certain intervals as proposed in Mnih's paper [4], I decided to try a soft update from machine learning paper [7]. The concept is to use weighted average between Q_{critic} and Q_{target} weights with a variable τ. The lower the τ, the more weights on Q_{critic}, meaning Q_{target} weights are updated slower. The soft update concept is similar to a regular update at certain interval steps: to reduce the target movement, which reduces the learning variance.

At the end of the inner loop, ε is multiplied by a decay value. The purpose is to slowly reduce the stochastic behaviour of the agent through out the training. The agent knows nothing about the environment at the beginning so it needs to explore more unknown states by having stochastic actions. As the agent knows more, executing more greedy actions will help maximize the reward.

At the end of the outer loop, I added a custom τ decay. τ slowly decays to a minimum value during training. This helps speed up training while providing stability. More discussion and experimentation is provided in the next section.

V. EXPERIMENT AND RESULTS

With the hyperparameters below, I was able to achieve an average of >200 scores with 100 episodes:

$\Upsilon = 0.99$, $\tau_{start} = 0.1$, $\tau_{min} = 0.001$, $\tau_{decay} = 0.99$, $\varepsilon_{min} = 0.01$, $\varepsilon_{decay} = 0.999$, $\text{batch}_{size} = 64$, $\text{M}_{size} = 500{,}000$, $EPISODES = 5000$, $MAX\ STEPS = 1000$

Neural network: 3 hidden layers: layer 1: 32 units, layer 2: 32 units, layer 3: 16 units. ReLU applied after each layer except for the output layer. Adam optimizer is used. MSE loss is used.

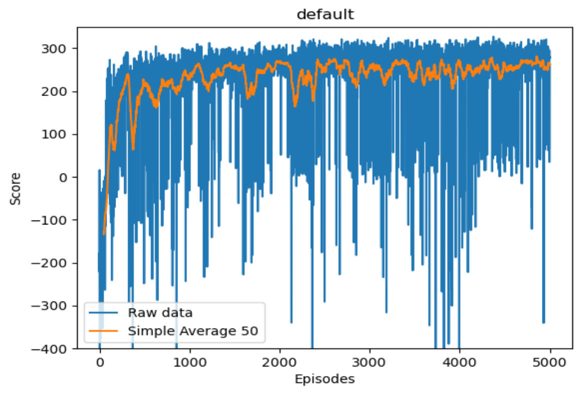

Figure 2 - Default Hyperparameters Training Cycle

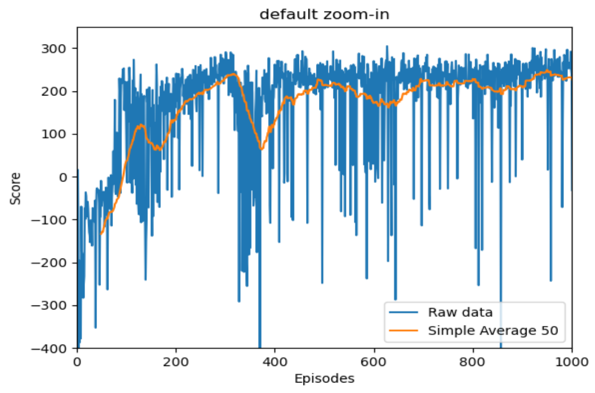

Figure 3 - Default Training Zoom-in

A typical training process takes a couple hours because there are 5000 episodes. The algorithm typically takes fewer episodes to converge. 5000 is used to see if there is possible unstable behaviour as training processes. Figure 2 shows the training process with default hyperparameter settings. The raw data varies a lot from episode to episode, so Figure 3 shows zoomed in version. The agent's scores go from negative a few hundreds to 200 in about 500 episodes. The simple moving average helps smoothen the curves. The agent keeps learning at a slower pace until around 2000 episodes. The best agent is saved along the training process. It is determined at every 100th episode that if the mean score minus 2 standard deviation is higher than the previous agent. Figure 4 shows the histogram of the trained agent over 100 episodes. The mean score is 270.64, and standard deviation is 34.58. As we can see, most of the episodes end up with scores higher than 200. Those end up with scores lower than 200 have uncommon initial states (e.g. x, y velocities) which lead the lander to unexplored states early on.

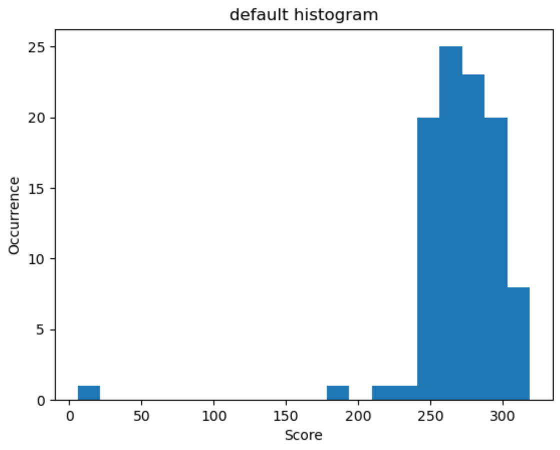

Figure 4 - Trained agent scores over 100 episodes

Hyperparameter: τ

The first hyperparameter to look at is τ, which is used in soft target update. As described in earlier section, it stabilizes the training process by "fixing" the target network weights. From algorithm 3, the soft update is described as:

$$\theta_{target} \leftarrow \tau \times \theta_{critic} + (1 - \tau) \times \theta_{target} \qquad (3)$$

In this experiment, we will have τ = [1.0, 0.5, 0.1, 0.03, 0.01, 0.001, 0.0001] while the rest of the hyperparameters stay fixed (default). In algorithm 3, τ also goes through a decay throughout the training. In this experiment, τ is fixed as a constant. From intuition, smaller τ will result in slower learning because θ_{target} weights change smaller from Eqn. 3. On the other hand, bigger τ will result in faster learning but can become unstable because of "moving target" which is difficult for neural network to do gradient descent. Figure 5 shows the training scores of 7 different τ values. We can observe that when τ = [1.0, 0.5, 0.1, 0.03], there is huge variation in the final scores during the learning process. At τ = 0.01, the scores start to stabilize with some volatility. For τ = [0.001, 0.0001], the training process is quite stable. There is always some zig zag but the curves stay above scores of 200 once the curves plateaus. For τ = [0.5, 0.1, 0.03, 0.01, 0.001], the agent learns quicker than the agent with τ = 0.0001.

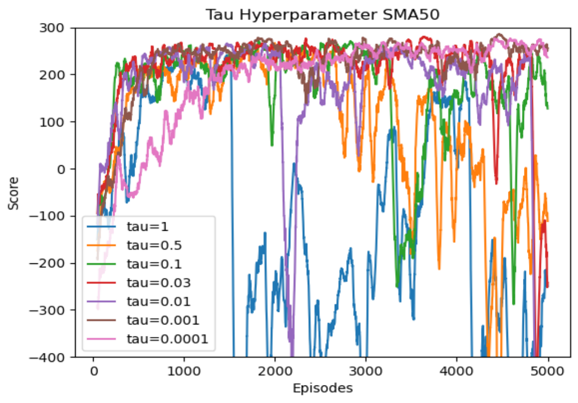

Figure 5 - Hyperparameter Tau Experiment

The observation confirms my hypothesis. Also, the idea of decaying τ comes from this experiment. A higher τ at the beginning for the agent to learn faster, and a lower τ at the end for stable learning.

Hyperparameter: ϒ

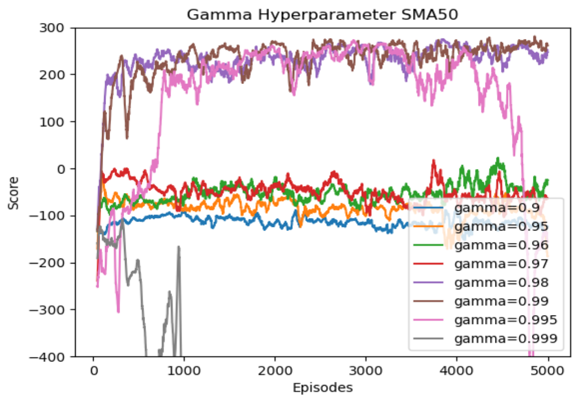

Figure 6 - Hyperparameter Gamma Experiment

The second hyperparameter is ϒ, which controls the discount reward. In this experiment, ϒ = [0.9, 0.95, 0.96, 0.97, 0.98, 0.99, 0.995, 0.999]. When the discount factor is low, future rewards are less important and the agent becomes more short-sighted. The solution is easier to converge but the policy may not be a desirable one. On the other hand, when ϒ is high, future rewards are more important and the agent will weigh more on future rewards.

The downfall is the solution may be harder to converge since the agent's horizon may be several times further away. Figure 6 shows the training results of 6 different ϒ.

When ϒ = [0.9, 0.95, 0.96, 0.97], the agent is short-sighted enough that it does not want the final landing reward, and the simple average score is low throughout the training. For ϒ = [0.98, 0.99, 0.995], the agent weighs more on the future rewards so it actually starts learning to achieve 200+ scores. ϒ = 0.995 starts to diverge near the end of the training cycle probably because some random initial conditions throw off the neural network weights, and it is at the edge of convergence difficulty. For ϒ = 0.999, the agent is simply unable to learn because the horizon is much further away. Intuitively, ϒ = [0.98, 0.99, 0.995] works because on average, the agent takes 200 to 300 steps to land successfully. $0.99^{300} = 0.049$ and the value is not too small for the agent to consider. Consider $0.95^{300} = 2.07 \times 10^{-7}$, and the value is likely negligible for the agent to include.

Hyperparameter: NN layer size

The last hyperparameter to test is the neural network layer size while the number of hidden layers is fixed at 3. Four configurations are considered: [16, 16, 8], [32, 32, 16] (default), [64, 64, 32], and [128, 128, 64]. Like any fully connected neural network application, it is likely there is some range of size that works well with a given problem. Outside the range, if the size is small, the solution will underfit, which means the network is not sufficient to learn the underlying patterns. If the size is big, overfit can occur and the results may become worse after reaching certain plateau during training cycle.

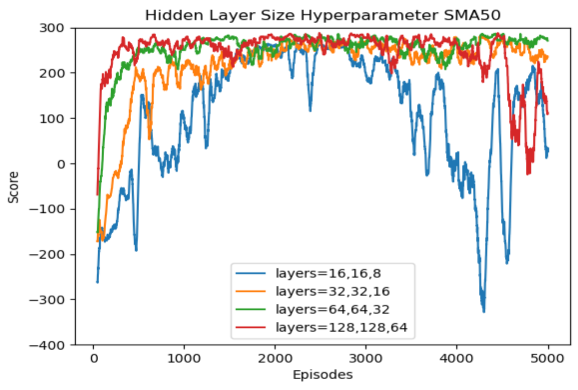

Figure 7 - Hyperparameter NN Layer Size Experiment

Figure 7 shows the learning cycles of different neural network layer sizes. We can see for [16,16,8], the NN is kind of struggling to learn to land the lander successfully. It eventually goes to 200+ around 2000 episodes briefly, but the simple average curve still shows volatility. It eventually gives up and fails. As the NN layer size grows, we can observe the slope of learning at the very beginning has a positive correlation. It does seem the largest NN layer size, [128,128,64], starts to struggle at the end of the training cycle. It could be by random that the initial condition of the lander was far off from the training parameters and made the NN weights derailed. Another possibility was the NN started to overfit.

VI. CONCLUSION

I demonstrated the theory and implementation of deep Q network (DQN) to tackle the lunar lander challenged posted by OpenAI. Regular Q-table would not work well in this case because of the sheer number of state-action pairs. I was able to train the model to land successfully and score 200+ average over 100 episodes. Throughout all runs there are still always failed attempts of landing from time to time, even after thousands of episodes. I believe one reason is the lander always starts at the top centre, which makes it very difficult to explore all states with equal probability. When some rare states are visited and the Q networks have not seen these states, the solution can spiral out of control.

VII. REFERENCES

1. "Lunar Lander," https://www.gymlibrary.dev/environments/box2d/lunar_lander/
2. Ng, A. Y., Harada, D., Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In I. Bratko and S. Dzeroski (Eds.), Proceedings of the 16th International Conference on Machine Learning, pp. 278–287.
3. Sutton, R. S., Barto, A. G. (2018). Reinforcement learning: An introduction 2nd edition. MIT press, Cambridge, Massachusetts.
4. Mnih, V., et al., "Human-level control through deep reinforcement learning", Nature, vol. 518, pp. 529–533, Feb. 2015.
5. "Reinforcement Learning (DQN) Tutorial", link: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
6. "PyTorch documentation", link: https://pytorch.org/docs/stable/index.html
7. Lillicrap, T., et al., "CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING", arXiv:1509.02971v6, link: https://doi.org/10.48550/arXiv.1509.02971