Landing Lunar Lander

1st Yuan Yang
yyang998@gatech.edu
Deep Reinforcement Learning

***Abstract*—This work solves OpenAI Gym's Lunar Lander problem [1] by implementing Q-learning algorithm [2] with the DQN (Deep Q-Networks) [3] [4]. The performance of this setup on Lunar Lander environment is evaluated by fine-tuning hyperparameters. Both of the training and generalization (testing) results are assessed across various combinations of hyperparameters.**

I. Introduction

OpenAI Gym's Lunar Lander problem [1] is a rocket trajectory optimization problem to land a space ship at a specific position with optimal rewards. An agent is created and trained to take actions within the Lunar Lander environment, aiming to maximize the received score/reward upon landing. To solve this environment, an off-policy TD learning algorithm, Q-learning, was implemented with improvements to build the agent. The main modifications are to couple the reinforcement learning algorithm Q-learning with a deep neural network(DQN) [3] [4], also with Experience Replay to better utilize historical experiences during learning.

This paper begins by introducing the Lunar Lander environment, followed by a justification for the choice of employing DQN based on the environment's characteristics. Then, the implementation details of the agent are outlined. The training performance is evaluated using the optimal combination of hyperparameters, and the trained agent is tested to access generalization performance. Additionally, the impacts of different parameters on the outcomes are discussed. Some alternative approaches will also be discussed.

II. Lunar Lander [1]

In the Lunar Lander environment [1], which is similar to a toy game based around physics control, the agent need to perform different actions on a space ship to land it on the landing pad (between two flags). The ship departs at the center of the upper bound of the frame, and a random force is generated and applied to the ship to promote movement. The reward can be collected and accumulated based on different action and states reached during the landing process. This system contains four discrete actions that can be performed on the ship's engines: do nothing, fire left orientation engine, fire main engine, and fire right orientation engine.

An 8-dimensional vector (8 state variables) is used to represent a state of the ship:

- Positions: horizontal x, vertical y, angle θ
- Linear velocities: horizontal $\dot{x}$, vertical $\dot{y}$, angular $\dot{\theta}$
- Boolean variables: if left leg of ship touching ground leg_L, if right leg of ship touching ground leg_R.

The scoring mechanism is sophisticated and every step during landing incurs a reward. Within each episode, the rewards are granted based on the following rules:

- Ship crashing: -100 (body of the ship touching ground);
- Ship landing: +100;
- Each leg ground contact: +10 (based on leg_L and leg_R);
- Each main engine firing: -0.3; Each left/right engine firing: -0.03 (based on current action selected);
- Increases/decreases as ship moving towards/against pad (based on x and y);
- Decreases as ship tilting (based on θ);
- Increases/decreases as ship moving slower/faster (based on $\dot{x}$, $\dot{y}$, $\dot{\theta}$).

As introduced above, 6 of the state variables are continuous in the state space and the rewarding system is complicated, making the environment more sophisticated. A Reinforcement Learning agent needs to be properly selected to solve this problem.

III. Algorithm Explanation: Q-Learning & Why choose off-policy over on-policy

In this work, the algorithm used in training the agent is Q-learning with a deep neural network (DQN). This section introduces the algorithm and the reason why this algorithm is selected to perform the learning.

Since reinforcement learning is about teaching an agent optimal behavior in an environment to achieve maximum reward, it is the applicable approach in solving this Lunar Lander problem. Among the various approaches in reinforcement learning, Q-Learning is chosen as the base algorithm. This algorithm is an off-policy temporal difference (TD) control approach. (Please refer to my Project 1's paper for the introduction to TD methods).

A. *Why choose off-policy over on-policy*

During the learning process, a policy needs to be learned by an agent, and this agent also needs to interact (choose actions) with the environment based on a policy. For on-policy algorithms such as State-Action-Reward-State-Action (SARSA) [5], the agent acts based on the same policy it learned. For off-policy methods such as Q-learning [2], the agent uses a different policy for action selection and learning. Because of the separation of policies, off-policy methods could offer extra exploration capabilities and are more flexible. The agent can update values based on the best estimate of the Q-values for state-action pairs (maximizing possible actions in the next state), even if this action has not been selected and taken in the environment. In the homework, both SARSA and Q-learning are used for learning in different types of environments. For a simple environment like Frozen Lake [6], on-policy methods could be more efficient since the extra exploration ability provided by off-policy might not be necessary. For a more complex system like Taxi [7] with larger state and action spaces, off-policy methods can explore more diverse experiences. Therefore, for the Lunar Lander environment in this work with complex setups, Q-learning is selected. However, since the state space is mostly continuous, extra improvements need to be made to vanilla Q-learning to achieve better performance (which will be introduced in the next few sections).

B. *Q-learning [2] and its limitations*

In Q-learning algorithm, the update rule for learned action-value at time step t is

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

where Q is the action-value function represented by a table: each value in the table is the expected rewards for an action taken in a given state. And S, A, R are state, action and reward respectively.

Moreover, at the beginning of each time step, the action is selected based on the state using a policy derived from Q using an approach called ϵ-greedy (introduced later in this paper). In equation (1), the $max_a Q(S_{t+1}, a)$ part indicates off-policy behavior: maximizing all possible actions in the next state.

Q-Learning works well for environments with discrete action and state spaces that are relatively small. However, for Lunar Lander with continuous states, using a table to represent Q is computationally expensive. The reason behind this is that to use Q-learning, the continuous state variables need to be discretized, which could lead to an explosion of the state space. Then, the Q-table could grow exponentially, and it could be very challenging to explore uncountable combinations of state and actions in a limited amount of time. Moreover, the discretization could potentially reduce the quality of learning.

To overcome these challenges, Minh et al. [3] proposed an improvement of Q-learning with a deep neural network (DQN). The DQN is the approach used in this work and is introduced in the next section.

IV. Algorithm Explanation: Deep Q-Learning & Network Architecture [3] [8]

Deep Q-learning with Experience Replay is an algorithm designed by DeepMind based on the raw Q-learning. Several improvements are adapted to achieve this algorithm.

A. *Deep Q-Learning & DQN*

As introduced in Section III-B, the q-learning algorithm's performance on a complex system with continuous states could be relatively low due to the explosion of the Q-table. Therefore, the state-action function Q in regular Q-Learning is replaced with a neural network in Deep Q-Learning. As shown in Fig. 1, the neural network is trained to provide **all possible actions and their corresponding Q-values** based on a state (input). This neural network is called a Deep Q-Network (DQN).

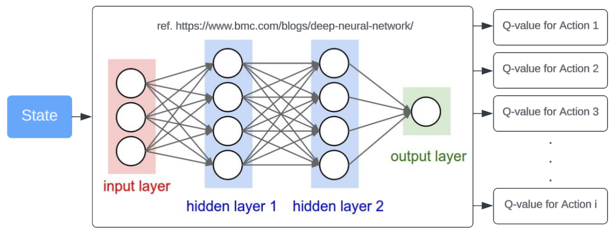

Fig. 1: Deep Q-learning Illustration: this network is only a sketch, refer to Section IV-C for actual architecture used.

B. *Experience Replay and Random Sampling*

In order to train the neural network, an experience replay mechanism needs to be adapted. In this algorithm, the experience (state, action, reward, next state) at each time step is stored in a replay memory buffer. The neural network is trained using randomly sampled mini-batches from all the past experiences stored in the replay memory. Random sampling from the replay memory is also an important strategy in this work. Consecutive samples could be strongly correlated with each other, further impeding the efficiency of training. Adapting a random sampling strategy can help mitigate this correlation.

C. *Neural Network Architecture and Training Process*

The neural network used in this work is with four layers. As shown in Fig. 1, since the input of the network is state, the input layer has 8 nodes (number of state variables in Lunar Lander). The output from network should be Q-values for each action, then the output layer has 4 nodes (number of actions). The dimensions of the two hidden layers is treated as hyperparameters for tuning. Moreover, ReLU (rectified linear unit) function is used in network to calculate the output of the node to provide better gradient propagation.

To train, after obtaining mini-batch from replay memory, Bellman equation is applied to calculate optimal Q-values:

$$Q^*(s, a) \leftarrow [r + \gamma max_{a'} Q^*(s', a')|s, a] \tag{2}$$

where optimal strategy is at the action that could maximize the expected value of $r + \gamma Q^*(s', a')$.

The network can be trained with a loss function (Mean Square Error - MSE). The MSE should be minimized between the predicted and the target values. The target value is obtained using network and Equation(2), while the predicted value is obtained only using the network at current step. Then the weights parameters in the neural network are updated through propagation and Stochastic Gradient Descent(SGD). The implementation details of the network are introduced in Experiments part.

V. Algorithm Explanation: Algorithm Details - Modifications and Why

This section introduces details of the actual algorithm used in the work for solving Lunar Lander. The Deep Q-learning algorithm with neural network introduced in SectionIV-C is adapted with slight modifications (refer to **Algorithm 1**).

One of the modifications is to use the **ϵ-greedy approach with ϵ decay** for the agent to select actions, instead of only using ϵ-greedy. The reason for employing ϵ-greedy is that if the agent only takes actions greedily by choosing the action that can provide the maximum Q-value at the current point, then it is possible that the agent will get stuck in a local sub-optimal solution and lose its capacity for exploration. The agent uses a probability ϵ at each step to help determine if it will take actions greedily or randomly select from the action space. In this way, the exploration and exploitation capabilities of the agent could be balanced. To further aid in this balance, on top of ϵ-greedy, the probability value of ϵ can be decreased over time to achieve ϵ decay. At the beginning of the training, we aim to explore the environment more with a higher ϵ to promote more random action selection and enhance exploration. Then, ϵ decreases to prioritize exploitation and select the optimal action greedily. Additionally, the minimum value for ϵ needs to be set to a fixed value to avoid excessive exploitation, and the amount of decrease at each time step should also be properly set. These two values serve as hyperparameters for tuning and are introduced in the Experiment section.

Other modifications include that the replay memory D has no capacity in my Algorithm since setting a capacity noticeably affects performance and efficiency. Also, according to the project requirement, training finishes when the average accumulated reward for the last 100 episodes reaches 200. The details of parameter tuning is presented in next section.

Algorithm 1 Algorithm used in this work: Deep Q-Learning

```
Initialize replay memory buffer $D$
Initialize deep neural network model as $Q$ function with random weights $\theta$
Initialize $\epsilon$ value to be 1
for episode = 1, NUM_EPISODES do
    Initialize state $s_1$ by resetting environment
    Initialize accumulated reward $total\_rw$ to be 0
    for t = 1, TIME_STEPS do
        n ← uniform random number between 0 and 1
        if $n < \epsilon$ then
            Select random action $a_t$
        else
            Select action $a_t = max_a Q(s_t, a; \theta)$
        end if
        Execute $a_t$ in environment and observe reward $r_t$, next state $s_{t+1}$, and a binary variable $terminated_t$ indicating if reaching terminal
        Store experience ($s_t$, $a_t$, $r_t$, $s_{t+1}$, $terminated_t$) in $D$
        Sample a random minibatch of experiences from $D$ with size of BATCH_SIZE
        for each ($s_i$, $a_i$, $r_i$, $s_{i+1}$, $terminated_i$) in batch do
            if $terminated_i$ then
                $y_i = r_i$
            else
                $y_i = r_i + \gamma max_{a'} Q(s_{i+1}, a'; \theta)$
            end if
            Calculate loss: MSE between $y_i$ and $Q(s_i, a_i; \theta)$
            Perform gradient descent step on loss through back-propagation with a learning rate $lr$
        end for
        Update state with $s_{t+1}$
        Add $r_t$ to $total\_rw$
        Break current for loop if $terminated_t$
    end for
    if $\epsilon$ >MIN_EPSILON then
        $\epsilon = \epsilon$ * EPSILON_DECAY_FACTOR
    else
        $\epsilon =$ MIN_EPSILON
    end if
    Finish training if average of last 100 episodes' $total\_rw \geq 200$
end for =0
```

VI. Experiment and Tuned Parameters for Optimal Performance

The experiment is separated into two parts: training and testing. The training part follows the **Algorithm 1**, and the testing part only uses the trained agent to get optimal solution (omit the part of network training with minibatches).

The hyperparameters introduced in **Algorithm 1** and **Section IV-C** are tuned based on several criteria: convergence during agent network training, training efficiency, potential underfitting or overfitting during training, and whether the agent gets stuck at a local sub-optimal during training. Additionally, the performance of the trained agent during testing (generalization) is taken into account.

As shown in Table I, there three types of parameters:

- General parameters for the algorithm (maximum number of episodes, maximum number of time steps in one episode, size of sampled minibatch, and γ in Bellman equation)
- Parameters for ϵ-greedy decay method (start value of ϵ, minimum value of ϵ, and decay factor of ϵ along episodes)
- Parameters for neural network (four layers as introduced in **Section IV-C**) that served as Q-function (learning rate, size of the two hidden layers, loss function used, optimizer used for gradient descending)

VII. Results and Effects of Hyperparameters

A. Training and Testing Results

Maximum number of episodes and maximum number of time steps in one episode only serve as hard stop points. During experiments, the training processes usually terminates before 1000 episodes since the average reward for 100 consecutive episodes reaches 200 prior to reaching 1000 episodes

TABLE I: Tuned values for hyperparameters

Hyperparameter	Tuned Value
maximum NUM_EPISODES	1000
maximum TIME_STEPS	2000
minibatch size BATCH_SIZE	64
γ	0.99
start ϵ	1.0
MIN_EPSILON	0.02
EPSILON_ DECAY_FACTOR	0.995
Network learning rate lr	0.001
Network hidden layer 1 size	128
Network hidden layer 2 size	64
Network loss function	MSE-Mean Squared Error
Network optimizer	Adam

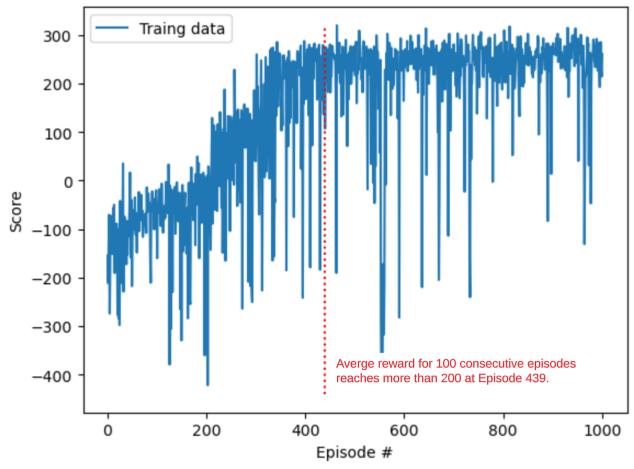

Fig. 2: Training Result: Reward vs. Episode

at most times. And for number of steps within episode, an episode could be terminated before the maximum step as long as the environment indicates that the lander has landed.

As shown in Fig. 2, the training processes took around 7 minutes for 1000 episodes when using this set of parameter values presented in Table I, and **the average accumulated reward for 100 consecutive episodes reached 201.67 at episode 439**. And the training converges efficiently within an acceptable amount of time and no over-fitting or under-fitting was identified.

The trained agent from Fig. 2 was tested and the result is presented in Fig. 3. The testing/generalization performance is satisfactory for 100 episode, and the average reward is 266.07. Even with some fluctuations involved, all the rewards are positive, and most of them are around 200 to 300.

During tuning, I mainly studied the impacts of network optimizer, mini-batch size, and γ on the performance. The discussion of these are presented in sections below.

B. *Effect of Network Optimizer Type*

For network optimizer, comparing to classic SGD method with Adam, **Adam can adaptively change the learning rate during training** and could be beneficial to training. As shown

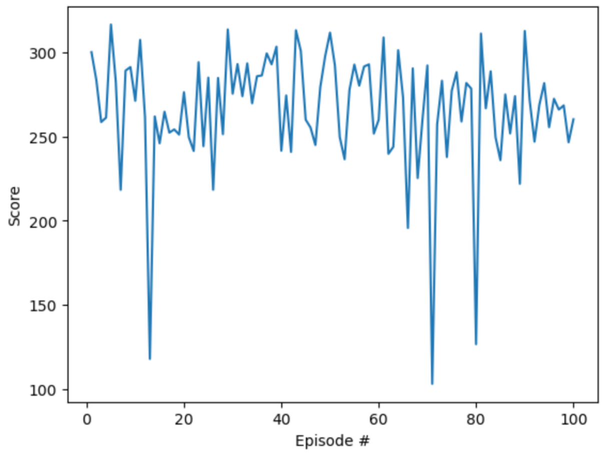

Fig. 3: Testing Result on Trained Agent: Reward vs. Episode

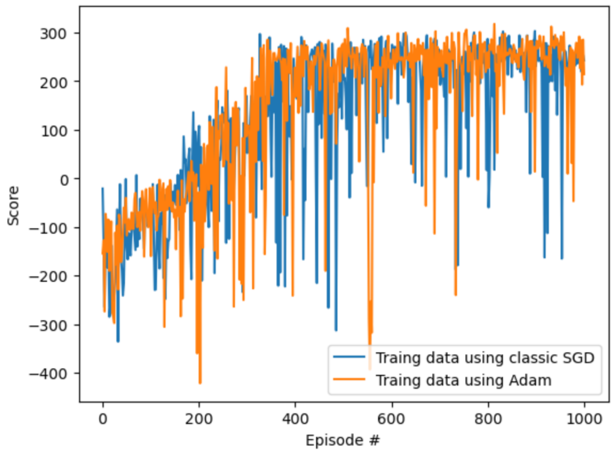

Fig. 4: Compare of Adam and classic SGD as Neural network optimizer: both train for 1000 episodes

in Fig. 4, changing from SGD to Adam **slightly enhanced the convergence**. With the same other parameter values, SGD reached average reward of 200 for 100 consecutive episodes at around episode 485, while Adam reached at episode 454. Therefore, even the testing performances are similar, I still chose to use Adam as the optimizer.

C. *Effect of Mini-batch Size*

As shown in Fig. 5, three sizes of minibatch are used for comparison. Batch size of 64 outperforms the other two. At each size, the average reward for 100 consecutive episodes reached 200 at: episode 619 for size 16, episode 439 for size 64, and episode 721 for size 128. Experiment with 64 as batch size (orange line) converges faster than both 16 and 128, and batch size of 128 performs worst in terms of convergence.

At a smaller size of 16, the model may start to overfit to the specific noise or pattern in the mini-batch itself. And during gradient descending, the updates to weight might not be large enough, this could slowing down the convergence process.

For a large size 128, beside the slow convergence, the model starts to overfit at episode 846. A large batch size could cause converge to a suboptimal, leading to instability in the training

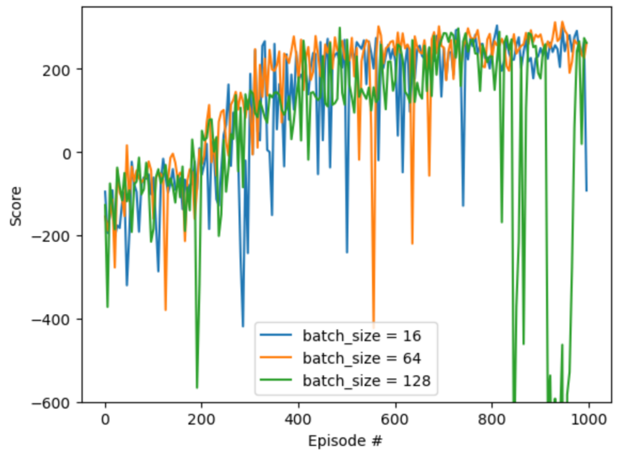

Fig. 5: Effect of Minibatch Size on Training (the data were sparsed to better identify trends)

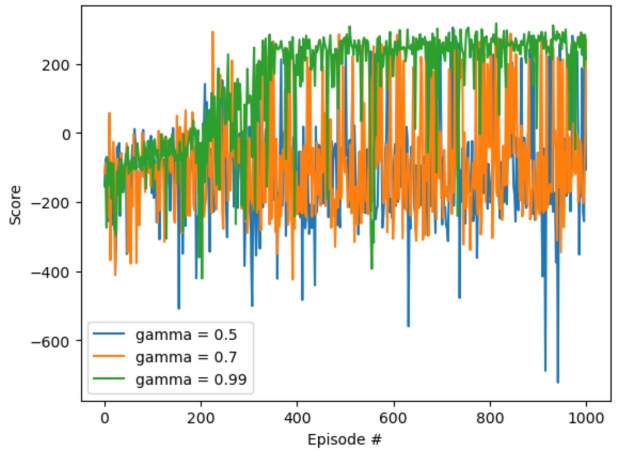

Fig. 6: Effect of γ on Training (the data were sparsed to better identify trends)

process. Also, the reason why the convergence is slow that we only train the model and updates weight when the number of experiences in Replay Buffer is larger than batch size. Therefore, with large batch size, the frequency for training and updating weight could be lower. Even training model with larger size could help perform better in generalization because the model can be trained with more diversed and smoothed data, to balance the convergence efficiency and generalization, I chose batch size of 64 during the tuning.

D. *Effect of γ in Bellman*

As introduced in Section 3 and 4, γ serves as the discount factor to determine the importance of future rewards. Tuning this hyperparameter could potentially affecting the training performance to a high extent. As shown in Fig. 6, three γ values are used for comparison. When gamma is smaller at values of 0.5 and 0.7, the model converged to suboptimals. When comparing the average reward from episode 900 to 1000, the average score is -113.11 for $\gamma = 0.5$, -50.23 for $\gamma = 0.7$, and 246.26 for $\gamma = 0.99$.

According to Bellman equation, when γ value is large such as larger than 0.9, the agent takes actions that could generate higher reward in the future. When γ value decreases, the agent would focus more on maximizing the current reward. With small discount factor, the training process could be converged fast but to a local suboptimal value because the agent does not focus much on the potential higher rewards in the future and it only focus on the short-term gain. Therefore, the lower γ like 0.5 and 0.7 converges to suboptimal (negative rewards) quickly. So I used a value larger than 0.9 (0.99) for the training to maximize the long-term reward.

E. *Effect of EPSILON_ DECAY_FACTOR*

I also studied the effect of EPSILON_ DECAY_FACTOR by changing it from 0.992 to 0.998. Due to length limitation of the paper, figures are not presented here (please refer to .ipynb notebook for the figure). When factor is 0.998, the average reward only reached 179.77 at episode 1000 and the agent converges much slower than agent with factor of 0.995. At higher factor, based on mechanism of ϵ-greedy introduced in Section 5, the ϵ is relatively large for the episodes at the beginning of the training. The algorithm randomly selects actions at a higher frequency, and it explores too much further reduce the exploitation capability.

VIII. Pitfalls and Future Work

The biggest pitfall I encountered is the implementation of ϵ-greedy approach. At first, I tried to use a fixed value of ϵ or change the ϵ value only within each episode across the time steps, but not changing it across the episodes. In my previous approach, every episode learns with same exploration and exploitation capabilities. This caused the training stuck at around -100 of the reward no matter how I tune the hyperparameters. Finally, ϵ decay approach largely improved the performance of my algorithm.

In the future, the algorithm could be improved by adapting the Double-q Learning [8] [9]: use two separate neural networks to select and to evaluate an action. One network for training to minimize the loss between predicted and target, and another network for getting the target Q-values. According to [9], the latter network can be updated along the process to mitigate overestimation of Q-values, and further improve the performance.

References

[1] https://gymnasium.farama.org/environments/box2d/lunar_lander/
[2] Chapter 6.5 - Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2nd Ed. MIT press, 2020. url: http://incompleteideas.net/book/the-book-2nd.html.
[3] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning,” NIPS Deep Learning Workshop 2013.
[4] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning,” Nature volume 518, pages529–533 (2015).
[5] Chapter 6.4 - Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2nd Ed. MIT press, 2020. url: http://incompleteideas.net/book/the-book-2nd.html.
[6] https://gymnasium.farama.org/environments/toy_text/frozen_lake/
[7] https://gymnasium.farama.org/environments/toy_text/taxi/
[8] https://www.baeldung.com/cs/q-learning-vs-deep-q-learning-vs-deep-q-network
[9] Hasselt et al. “Deep Reinforcement Learning with Double Q-learning,” AAAI 2016.