CS7642 Project 2: Lunar Lander

Fernando Villamarin
fdiaz35@gatech.edu
Commit Hash: a0dc84073c415d728c190b8f814d789fedf78706

***Abstract*—This study investigates the impact of various hyperparameters on the performance of a Deep Q-Learning (DQL) agent in the Lunar Lander environment, focusing on the discount factor (γ), exploration rate (ϵ-decay), and batch size. Additionally, the performance of the Dueling Network architecture was compared with regular single-stream architectures.**

I. Lunar Lander

The Lunar Lander environment, part of the Box2D simulation environments in the Gymnasium framework, emulates a lunar lander module that requires control in order to achieve a safe landing on the Moon's surface. The discrete action space comprises four distinct actions: do nothing, fire left orientation engine, fire main engine, and fire right orientation engine. The observation space is an 8-dimensional vector containing information about the lander's position, velocities, angle, angular velocity, and leg contact with the ground, as shown in (1).

$$(x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, \text{leg}_\text{L}, \text{leg}_\text{R}) \tag{1}$$

Reward mechanisms within the environment allocate points for successfully navigating from the top of the screen to the landing pad and achieving a state of rest, while imposing penalties for deviating from the pad, crashing, or firing the engines. The environment is deemed solved when the agent amasses a total of 200 reward points. The lander module initiates at the top center of the viewport with a random initial force exerted on its center of mass, and an episode concludes if the lander experiences a crash, strays beyond the viewport, or enters a sleep state.

II. Q-Learning

Q-Learning, recognized as a pivotal advancement in the domain of reinforcement learning, is an off-policy Temporal-Difference (TD) control algorithm. As opposed to SARSA, Q-Learning is considered off-policy as it learns an optimal policy independently of the policy employed during exploration. This distinction arises from the manner in which Q-Learning updates its action-value function, denoted as $Q(s, a)$:

$$\begin{aligned} Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') \\ - Q(s_t, a_t) \Big] \end{aligned} \tag{2}$$

The critical difference in the update rule (2), compared to SARSA, is the usage of $\max_{a'} Q(s_{t+1}, a')$ instead of $Q(s_{t+1}, a_{t+1})$. This component of the update rule denotes the selection of the action with the highest Q-value in the next state, thereby learning the optimal policy irrespective of the current exploration policy.

However, a major limitation of Q-Learning is its inability to manage continuous state or action spaces. The standard Q-Learning algorithm relies on maintaining a tabular representation of the action-value function $Q(s, a)$ for all state-action pairs, rendering it impractical for continuous state or action spaces due to the infinite number of possible state-action combinations. As a result, Q-Learning can face scalability issues and may struggle to converge to an optimal policy in such environments.

III. Deep Q-Learning (DQL)

Deep Q-Learning (DQL) is a technique that synergistically integrates the principles of conventional Q-learning with the representational capabilities of deep neural networks. Pioneered by Mnih et al. [2], the DQL algorithm addresses the intrinsic limitations of classic Q-Learning techniques in handling high-dimensional and continuous state or action spaces effectively.

The fundamental concept underpinning DQL is the approximation of the Q-function, responsible for mapping state-action pairs to their respective Q-values, using a deep neural network as a function approximator. Neural networks trained through this method are referred to as Deep Q-Networks (DQN). By utilizing a DQN, the inherent structure of the Q-function can be effectively learned, enabling generalization from observed experiences to make predictions about previously unencountered state-action pairs. This capacity to learn from raw sensory input empowers DQN agents to establish intricate control policies even in the face of demanding environments.

Furthermore, the DQL algorithm incorporates experience replay, a technique that stores past experiences in a replay buffer and samples them randomly for training. This approach breaks the temporal correlations between experiences, promoting more stable and efficient learning.

A. Double Deep Q-Learning

Double Deep Q-Learning is a crucial advancement in the DQL algorithm, as it introduces a target network [4], which serves as a separate neural network used for stabilizing the learning process. By employing a fixed set of Q-value estimates during training, the target network mitigates the risk of oscillations and divergence that might arise from the use of a single, constantly updating network.

Algorithm 1: Double Deep Q-Learning with Experience Replay

Data: Learning rate α, discount factor γ, soft update factor τ, exploration probability ϵ

Initialize primary network Q with random weights θ;
Initialize target network $\hat{Q}$ with weights $\theta^- = \theta$;
Initialize experience replay buffer D to capacity N;
for *episode* $e = 1, M$ **do**
  Initialize state s;
  for *time step* $t = 1, T$ **do**
    Choose action a based on exploration strategy ϵ and $Q(s, a; \theta)$;
    Execute action a, observe reward r and next state s';
    Store transition (s, a, r, s') in replay buffer D;
    Set $s = s'$;
    Sample a random minibatch of transitions from D;
    foreach *transition* (s_i, a_i, r_i, s'_i) *in minibatch* **do**
      Compute $a'_Q = \arg\max_{a'} Q(s'_i, a'; \theta)$;
      Compute target value y_i:

$$y_i = \begin{cases} r_i, & \text{for terminal } s'_i \\ r_i + \gamma \hat{Q}(s'_i, a'_Q; \theta^-), & \text{otherwise} \end{cases}$$

      Update primary network weights θ using gradient descent on $(y_i - Q(s_i, a_i; \theta))^2$;
    Perform soft update on target network weights: $\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$;

B. Dueling Deep Q Networks

Dueling Deep Q-Networks (DDQN), introduced by Wang et al. (2016), represent a significant advancement in the field of reinforcement learning by proposing a novel neural network architecture tailored specifically for model-free RL. The central idea of DDQN is the explicit separation of the representation of state values, denoted as $V(s)$, and state-dependent action advantages, denoted as $A(s, a)$. The proposed dueling architecture consists of two distinct streams that represent the value and advantage functions, while sharing a common feature extraction backbone. Formally, the DDQN estimates the action-value function $Q(s, a)$ as shown in (3).

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \alpha) + \left(A(s, a; \theta, \beta) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \beta) \right), \tag{3}$$

where θ represents the shared parameters of the feature extraction backbone, and α and β denote the parameters of the value and advantage streams, respectively. $\mathcal{A}$ is the set of possible actions, and $|\mathcal{A}|$ is the cardinality of the action set.

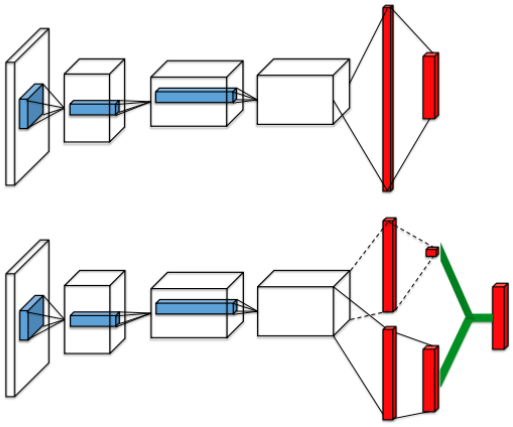

Fig. 1. A common single-stream Q-network (top) and the dueling Q-network (bottom) are depicted. The dueling network employs two distinct streams to separately estimate the scalar state-value and the action advantages; the green output module incorporates Equation (3) to integrate these estimates. Both networks produce Q-values for every action. [5]

In order to assess the performance of different network architectures, we compared the conventional single-stream network architecture with the dueling network architecture, in addition to tuning hyperparameters. The dueling architecture's ability to separate the estimation of state values and action advantages allows the network to learn valuable states without the need to explicitly consider the impact of each action in every state. This characteristic proves to be particularly advantageous in the Lunar Lander environment, where certain instances may render the lander's actions to have minimal or negligible effects on its position or orientation.

In such situations, the dueling architecture is capable of rapidly identifying the most appropriate actions for the lander without being burdened by the necessity to evaluate the consequences of every possible action in a specific state. Wang et al. [5] demonstrated the efficacy of this approach in the Atari game Enduro, wherein the agent had to navigate a car down a road while avoiding collisions with obstacles, a problem bearing resemblance to the Lunar Lander environment.

TABLE I
NETWORK ARCHITECTURES

Network Architecture	Hidden Layers
Network 1	64 x 64
Network 2	64 x 128 x 64
Network 3	128 x 256 x 128
Dueling Network	64 x 64 x 32; 16; 16

The networks examined in this study are presented in Table I. ReLU activation functions are employed in all layers for each of these networks.

IV. Methodology

To identify appropriate hyperparameters, the networks were initialized with a set of parameters, trained over 2,000 episodes, and subsequently tested across 100 episodes. The hyperparameters investigated through a grid-search approach are presented in Table II. A total of 24 agents were trained and tested in this step, encompassing 4 network architectures and 3 hyperparameters, each with 2 possible values.

Throughout the training and testing process, agents were restricted to a maximum of 500 steps. If they were unable to land the aircraft within this limit, the environment would terminate without a reward.

TABLE II
HYPERPARAMETERS EXPLORED

Hyperparameter	Values
	[0.001, 0.01]
	[0.005, 0.05]
Batch Size	[64, 128]

The agent that delivered the most outstanding performance utilized the Dueling Network, attaining an average total reward of 250.56 across 100 consecutive episodes. As illustrated in Figure 2, the agent experienced a crash in only one episode, specifically episode 75.

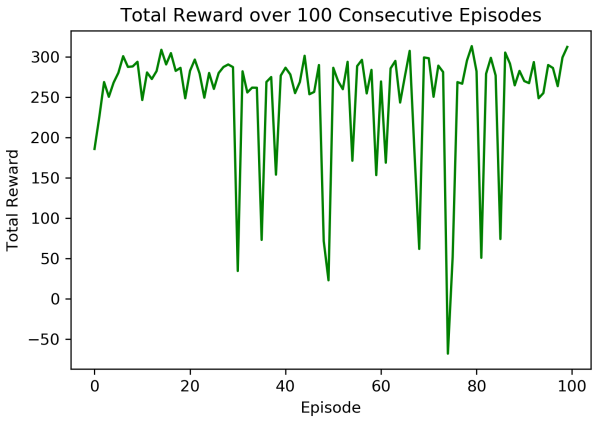

Fig. 2. Test performance of the Dueling Network following 2,000 episodes of training. The average total reward is 250.56, and the agent encountered a single crash.

The ultimate selection of hyperparameters employed in the experiments is presented in Table III, showcasing the optimal configuration discovered after the hyperparameter search. It is important to mention that for the ϵ-decay, we have employed a decay schedule, similar to the one proposed by Morales [1], which logarithmically spaces the values over the entire range of training episodes.

As illustrated in Figure 3, the agent was able to successfully clear the environment at around episode 1,250 in the training environment, where it followed an ϵ-greedy strategy,

TABLE III
FINAL HYPERPARAMETERS

Hyperparameter	Value
Network Architecture	Dueling Network
Number of Episodes	2,000
	0.99
	0.001
	0.005
-starting	1.0
-ending	0.1
-decay	0.9
Batch Size	64
Buffer Size	100,000
Optimizer	Adam
Loss Function	Smooth L1 Loss

choosing random, sub-optimal actions with a probability of ϵ. As mentioned earlier, after training, the agent achieved an average total reward of 250.56. We considered implementing early stopping to halt training once the agent could clear the environment; however, since the objective of this project is to explore the effects of different hyperparameters, we decided against this approach.

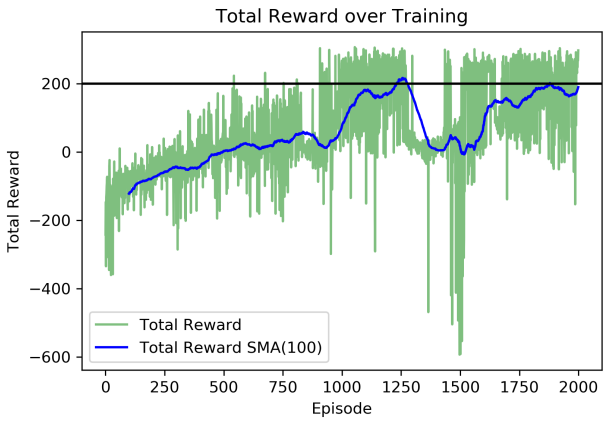

Fig. 3. The training performance of the Dueling Network was observed over 2,000 training episodes. As illustrated by the blue line, the agent achieved a total reward of over 200 across 100 consecutive episodes around episode 1,250. This accomplishment is particularly impressive considering that, at that episode and according to the schedule, the agent had a 12.79% probability of selecting a random action.

V. Experiments & Results

To investigate the impact of varying hyperparameter values, we utilized the agent outlined in Table III and examined the effects of different values for γ, ϵ-decay, and batch size. To facilitate clearer comparisons, we employed a rolling average of the total reward from the last 100 training episodes.

A. Gamma

As shown in Figure 4, the agent with a γ of 0.99 is the top performer. However, this model exhibits more variance than the others, which are much more stable. Except for the dip in performance that the agent in red ($\gamma = 0.9$) experiences in the

later episodes, it can be observed that models with a higher value of γ generally perform better.

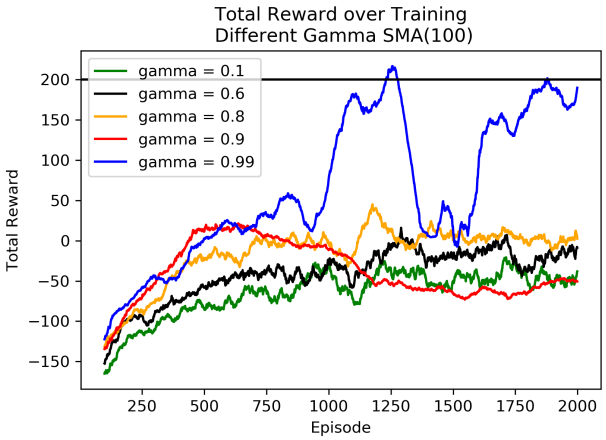

Fig. 4. Total reward received while training for different values of .

In the Lunar Lander environment, it is crucial for the agent to learn a series of decisions that lead to a safe landing on the moon's surface. A higher value of γ prioritizes long-term rewards over immediate ones, which in turn encourages the agent to consider the implications of its actions over an extended temporal horizon.

In this environment, landing safely requires the agent to carefully control the lander's velocity, angle, and fuel consumption. By prioritizing long-term rewards, the agent can better learn to optimize these factors in a coordinated manner, ultimately leading to more successful landings.

In contrast, agents with lower values of γ might focus more on immediate rewards, potentially neglecting important aspects of the task or making shortsighted decisions that could lead to crashes or suboptimal landings. This is why models with a higher value of γ generally perform better in the Lunar Lander environment.

B. Epsilon

To examine the impact of different ϵ values on the agent's performance, the hyperparameter ϵ-decay was adjusted while keeping the ϵ-ending at 0.1 and the ϵ-starting at 1.0. The resulting decay schedules can be observed in Figure 5.

Examining Figure 6, it is particularly noteworthy that the agent with the lowest ϵ-decay, 0.1, can achieve a total reward of over 200 as early as episode 250. It can be observed that agents with lower ϵ-decay values, which are more prone to exploitation, reach the 200-point threshold earlier. This is due to the fact that these agents explore the environment less aggressively compared to others with higher ϵ-decay rates.

This slower rate of exploration means that the agent focuses more on exploiting the learned knowledge, which may allow it to achieve higher rewards more quickly. However, this might also result in a less diverse exploration of the environment, potentially leading to suboptimal policies.

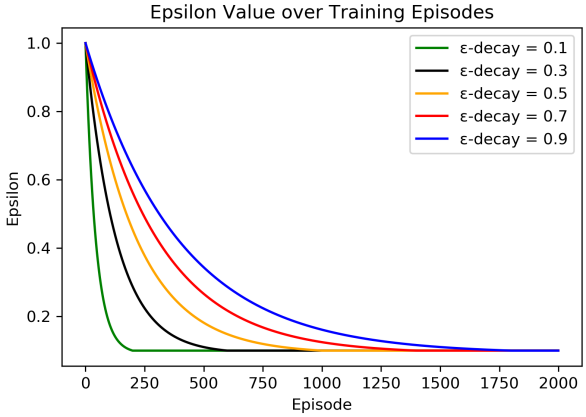

Fig. 5. Different decay schedules. The schedules are logarithmically spaced according to -decay and offset by -ending.

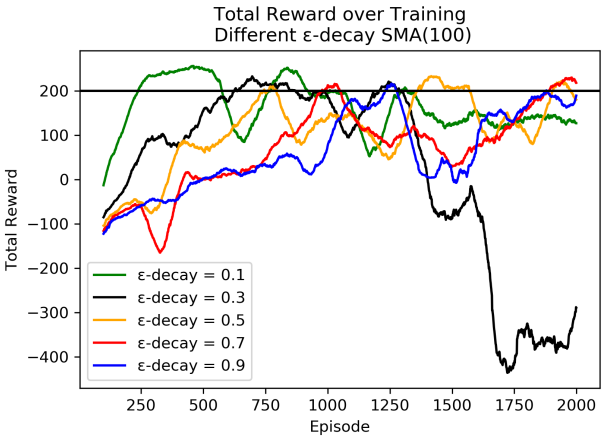

Fig. 6. Total reward received while training for different values of -decay.

Thus, although the agent demonstrated good performance in this particular case, it is quite possible that the lack of exploration may have detrimental effects in other runs. In fact, in Figure 6, the performance of the agent with ϵ = 0.3 is observed to deteriorate significantly after the 1,250th episode.

This decline could be attributed to the agent converging to a suboptimal policy due to limited exploration. The initial success in attaining high rewards may have led the agent to believe that its current policy was optimal. As the agent persists in exploiting its learned knowledge without adequate exploration, it may fail to identify superior strategies. Consequently, the performance deteriorates beyond the 1,250th episode, as the agent becomes trapped in a local optimum and is unable to adapt to novel challenges or more effective approaches.

C. Batch Size

In DQL, the batch size refers to the number of experiences or transitions sampled from the replay buffer to train the neural network during each update step. The batch consists of a set of state-action-reward-next state tuples, which are used to compute the loss function and update the network's weights.