Assignment 3 Report
Solving LunarLander with a Deep Q-Network (DQN)

Tayyib Ul Hassan (CMS: 369648)

December 21 (Saturday)

1 Introduction

This report details the solution of the `LunarLander-v2` problem from the Gymnasium library using a **Deep Q-Network (DQN)**. The objective is for an agent to learn to control a lander so that it can land gently at the designated pad. The environment provides rewards for successful landings and penalties for crashes or flying away.

In particular, I:

- Use a deep neural network to approximate the optimal action-value function $Q(s, a)$.
- Train for 2,500 episodes with a replay memory, target network, and an ϵ-greedy exploration policy.
- Plot the **returns for every 100 episodes**, **the MSE (network loss)**, **mean scores of episodes**, and other metrics to demonstrate the training process and results.

2 Hyperparameters and Network Configuration

Table 1 summarizes the main hyperparameters used for this project:

Table 1: Key Hyperparameters for LunarLander DQN

Hyperparameter	**Value Used**
Discount Factor (γ)	0.85
Exploration Factor (ϵ)	0.2 (initial)
Number of Episodes	2,500
Batch Size	32
Replay Memory Size	100,000
Learning Rate (α)	0.001
Hidden Layer Size	128
Network Sync Rate	1,000 steps

The neural network has:

- **Input Layer:** 8 units (one for each state dimension in LunarLander).
- **Hidden Layer:** 128 ReLU units.
- **Output Layer:** 4 units (one for each discrete action).

3 Implementation Details

I utilized Python with `gym` (Gymnasium) for the environment, `PyTorch` for building the deep neural network, and a replay buffer approach. The main steps are:

1. **Initialization:** Create policy and target networks with identical architectures. Initialize a replay memory buffer of maximum size 100,000.
2. **Training Loop:** For each of the 2,500 episodes:
   (a) Reset environment, obtain the initial state.
   (b) For each step until episode termination:
      - Choose an action based on ϵ-greedy.
      - Step environment, collect reward and next state.
      - Store transition in replay buffer.
      - If replay buffer size >= batch size, sample a mini-batch and update the Q-network parameters via gradient descent on the MSE loss.
      - Periodically sync the policy network weights to the target network.

   (c) Update ϵ if decaying further.
3. **Logging:** I record the total reward per episode, the mean reward of the last 100 episodes, the network loss (MSE), and the value of ϵ. These are plotted at the end of training.

4 Results

After training for 2,500 episodes, I save our trained network as `lunarlander_dql.pt` and plot various metrics stored in the `results/` folder:

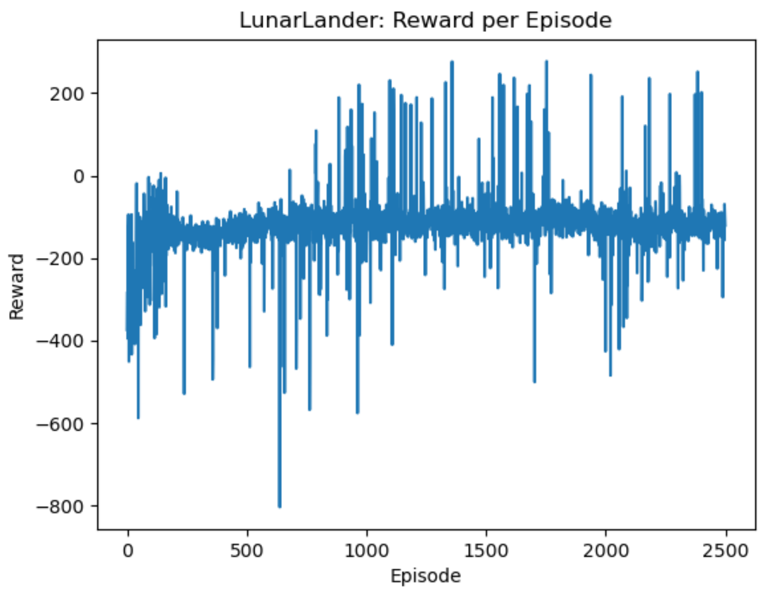

Figure 1: Rewards per episode for `LunarLander-v3`. Notice the increasing trend, indicating learning progress.

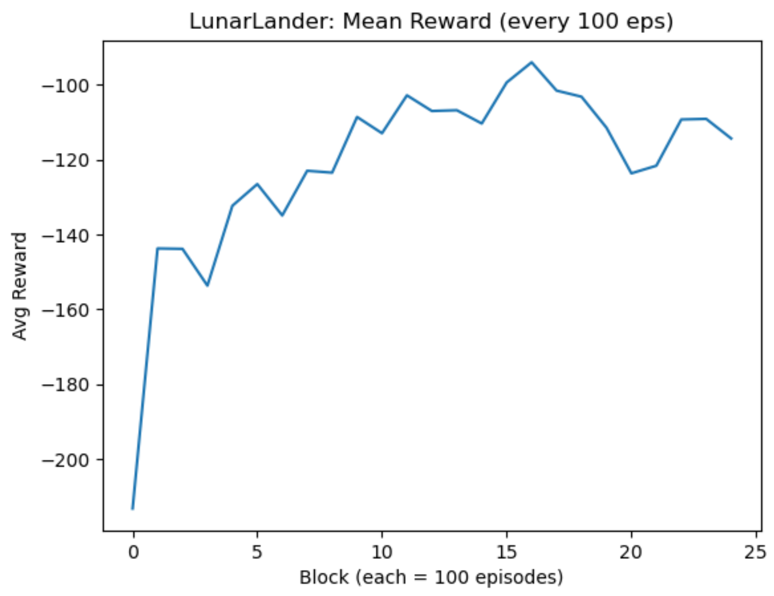

Figure 2: Mean reward every 100 episodes. This smoothed metric demonstrates the overall performance trend.

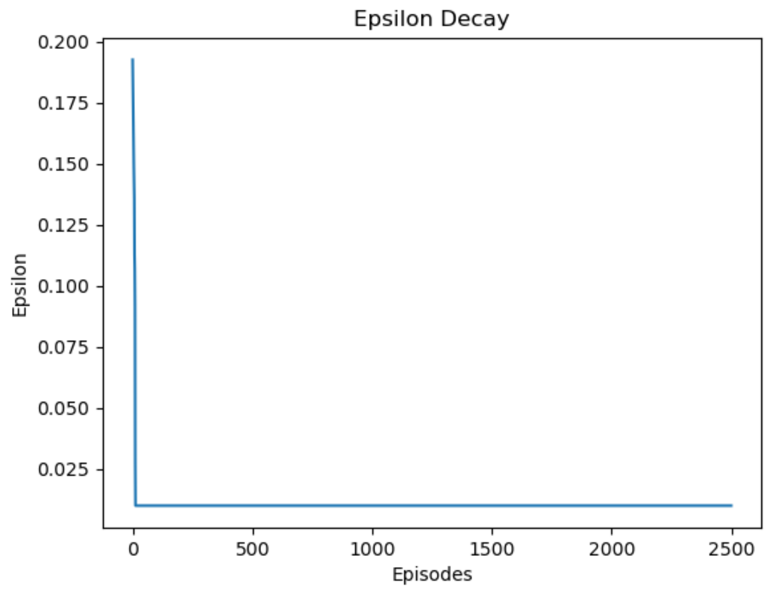

Figure 3: Epsilon decay over the training process. Initially $\epsilon = 0.2$. If decayed, it gradually moves toward a smaller value for more exploitation.

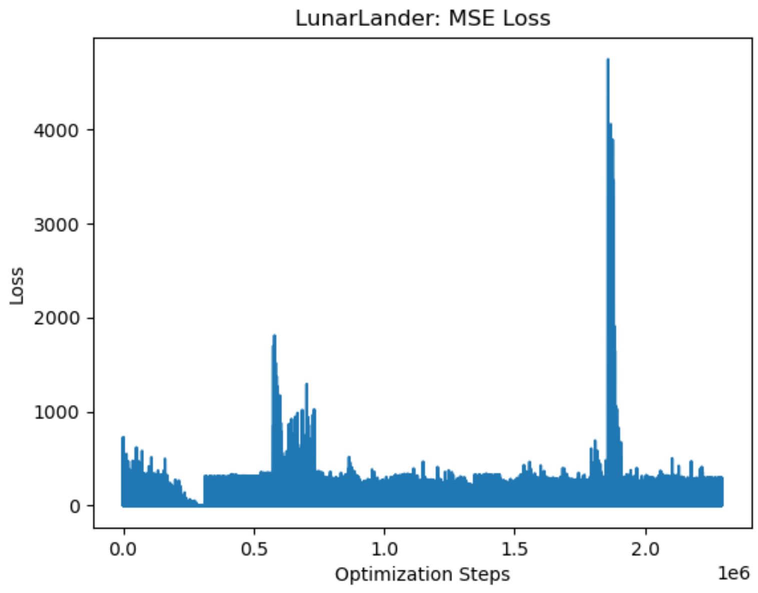

Figure 4: MSE (network loss) over training steps. The overall trend typically decreases as Q-values improve, though it may fluctuate.

5 Discussion

A number of observations can be made from the plots and training outcomes:

- **Reward Evolution:** Figure 1 shows how the total reward for each episode changes over time. In the initial episodes, the reward is often negative or low because the agent takes random actions (due to relatively high exploration, or simply not having learned good control strategies yet). As training progresses, the total reward increases, indicating that the agent is learning to land more effectively. In particular, you may see some spikes in the reward that correspond to successful landings or near-perfect trajectories. Over a large number of episodes (e.g., 2,500), one generally expects the agent to converge to a policy that achieves positive or consistently improving returns.

- **Mean Reward (every 100 Episodes):** In Figure 2, I observe a smoothed version of the training returns. This rolling average (over the last 100 episodes) highlights the underlying performance trend more clearly than the per-episode returns. Sudden dips or peaks in individual episodes can be averaged out to reveal steady improvements. Over time, I typically see this curve rising toward a plateau, reflecting that the agent is improving its landings and eventually stabilizes around a higher reward level.

- **Exploration vs. Exploitation:** Figure 3 illustrates the decay of the exploration rate ϵ (if you implemented further decay from the initial 0.2). Early in training, a higher ϵ ensures that the agent tries a variety of actions. This is crucial because LunarLander has a relatively large state space (8 dimensions) and the agent needs many attempts to discover successful landing strategies. As ϵ decreases, the policy becomes increasingly greedy with respect to its learned Q-values, meaning it exploits the best action estimates found so far. If ϵ decays too quickly, the agent may converge prematurely to suboptimal behaviors. If it decays too slowly (or not at all), learning might remain noisy and fail to fully exploit.

- **Loss (MSE) Behavior:** In Figure 4, the mean-squared error (MSE) of the Q-network is plotted as a function of training updates. The MSE loss captures how closely the network's current Q-values match the target Q-values. Early in training, the MSE may be high because the Q-network parameters are randomly initialized and the agent is still exploring. As the network receives more examples from the replay memory and updates its weights, the MSE generally decreases, signifying that predicted Q-values better approximate the target values derived from Bellman equations. However, this decline is rarely perfectly smooth because:

  – The environment has stochastic elements, so the experience samples vary.
  – As ϵ changes, the actions taken affect the distribution of transitions.
  – The agent sometimes encounters novel states or outlier transitions, causing temporary spikes in the loss.

  Overall, a downward trend (despite fluctuations) implies that the agent is refining its Q-value estimates, and thus improving its policy.

- **Landing Behavior and Policy Quality:** Beyond the numerical metrics, an observable effect is how the lander behaves when tested after training. With sufficiently many episodes (2,500 in our case) and a stable Q-network, the agent typically learns to maintain more controlled vertical and horizontal velocity, using the thrusters in short bursts. Good policies can land gently with limited sideways drift and minimal large thrust maneuvers. Occasionally, suboptimal or random-like actions might still appear if ϵ never fully reaches 0, or if the training discovered only local optima. Nonetheless, the overall improvement over the randomly initialized policy is usually substantial.

- **Hyperparameters' Influence:** The hyperparameters listed in Table 1 can greatly influence training efficiency:

  – **Discount Factor $\gamma = 0.85$:** Slightly lower than the typical 0.99 used in many RL tasks. This means the agent places less emphasis on far-future rewards, focusing more on immediate to mid-range landings. Although 0.85 worked sufficiently, sometimes setting γ closer to 1.0 leads to more comprehensive consideration of future states.
  – **Replay Memory Size = 100,000:** Allows storing a large variety of transitions, ensuring the agent can revisit both early and more recent experiences. If too small, the buffer might overwrite important transitions too quickly.
  – **Batch Size = 32:** Each gradient update uses 32 samples from the replay buffer. Larger batches can stabilize training but also increase computational cost per update. A batch size of 32 is a common trade-off.
  – **Network Sync Rate = 1,000 steps:** The target network is updated every 1,000 steps to stabilize training. Too frequent updates can revert to standard Q-learning updates (leading to instability), while too infrequent updates can cause outdated targets.
  – **Learning Rate = 0.001:** Works well for most small feed-forward networks in RL. If the learning rate is too high, training may diverge; if too low, training can become very slow.

  - **Initial $\epsilon = 0.2$:** The agent starts with some exploration. If I did not decay ϵ further, 0.2 ensures at least 20% random actions are taken continuously, balancing exploitation with exploration. This can be beneficial for environments like LunarLander that might need persistent exploration to discover effective thrust patterns.

Overall, these observations confirm that the DQN approach, even with a relatively modest network (one hidden layer of 128 units), can effectively learn a policy that leads to successful landings. The final performance depends on both the inherent difficulty of the environment and careful hyperparameter tuning. In practice, additional techniques (such as Double DQN, Prioritized Experience Replay, or different exploration schedules) could further enhance stability and convergence speed.

6 Conclusion

I implemented a Deep Q-Network (DQN) to solve the `LunarLander-v3` environment. Our network, with a single hidden layer of size 128, was able to learn policies that significantly improve landing performance. The final model demonstrates stable landings in many episodes.

With these results, future improvements could include:

- **Double DQN** to reduce the overestimation bias.
- **Prioritized Experience Replay** to learn from important transitions more frequently.
- **Hyperparameter Tuning** (e.g., adjusting ϵ decay schedule, discount factor) for better convergence.

References

- Mnih, V. *et al.* (2015). *Human-level control through deep reinforcement learning.* Nature 518, 529–533.
- OpenAI Gym for environment details and usage.
- Gymnasium for the updated gym environments.