CS7642-RL Project 2 Report - Double Q-Learning to solve Lunar Lander problem

Agnibho Chattarji
Github Commit Hash - 644a332214208960da1c32fdabc0cb7cb1d91aae
Email: achattarji3@gatech.edu

***Abstract*—This study addresses the ”Lunar Lander” challenge via the OpenAI Gym’s LunarLander-v2 environment, leveraging a Double/Deep Q-Network (DQN) to achieve and maintain an average score of at least 200 points over 100 consecutive attempts. Our findings highlight the crucial role of randomness and the epsilon decay parameter in navigating the trade-off between exploration and exploitation to guide the learning model towards successful outcomes. The investigation reveals that maintaining an adaptable epsilon decay strategy in response to performance gains, and introducing randomness when needed, could significantly impact the model’s probability of convergence. Notably, consistent performance emerged as a more reliable success indicator than peak average scores. We also investigate building a model by optimizing three chosen hyperparamaters.**

I. Deep Q-Learning to solve Lunar Lander problem

A. Lunar Lander

In the ”Lunar Lander” challenge, presented through the OpenAI Gym’s [1] LunarLander-v2 [2] environment, players are tasked with piloting a spaceship for a safe lunar landing. The game intricately simulates the landing process, incorporating eight specific details including the spaceship’s position, velocity, and orientation, to define its state. Players can manipulate the spacecraft using four actions: doing nothing, turning left, descending straight down, or turning right, all aimed at achieving a precise landing on a designated area known as the landing pad.

The scoring system in LunarLander-v2 is designed to reward precision and judicious management of resources. Points are awarded for guiding the spaceship towards the landing pad, with a successful touchdown yielding a significant bonus of +100 points. Additionally, making contact with the ground using the spaceship’s legs nets +10 points per leg. However, the game imposes penalties to discourage recklessness: crashing the spaceship results in a hefty deduction of -100 points, and fuel consumption also impacts the score negatively, with the use of the main engine deducting 0.3 points per use and the side engines 0.03 points, thus promoting efficient fuel use.

The overarching goal in this OpenAI Gym environment is to master the delicate balance of maneuvering, landing and resource management, aiming to achieve an average score of at least 200 points across 100 consecutive attempts.

B. Why Double Q-Learning?

Deep Q-Learning (DQN) stands out as a particularly effective strategy for tackling the Lunar Lander game, a complex challenge that demands navigation through a high-dimensional state space. This approach’s potential was first highlighted in the pioneering work by Mnih et al., ”Playing Atari with Deep Reinforcement Learning”[3] in 2013. Through this study, DQN was shown to adeptly manage tasks requiring the interpretation of complex and continuous input spaces—much like the environment presented in Lunar Lander. The core appeal of DQN lies in its utilization of deep neural networks to estimate the Q-value function, thereby learning optimal policies directly from high-dimensional sensory inputs in an end-to-end manner.

However, traditional DQN approaches can sometimes overestimate Q-values due to their singular network structure, potentially leading to instability in training and less-than-ideal policy outcomes. This challenge is addressed by Double Q-Learning, which was specifically adapted for deep learning contexts by van Hasselt et al. in ”Deep Reinforcement Learning with Double Q-learning”[4] in 2015. Double Q-Learning minimizes the overestimation bias of Q-learning by separating the processes of action selection and target Q-value computation. It achieves this through the deployment of two distinct networks: one for selecting actions (policy network) and another for evaluating the Q-value of subsequent states (target network). This bifurcation ensures more accurate and stable Q-value predictions, promoting a more consistent advancement toward optimal decision-making policies.

In the context of the Lunar Lander challenge, the precision and reliability offered by Double Q-Learning are particularly valuable. The environment’s requirements for careful control and decision-making under uncertainty align well with the method’s focus on exact Q-value estimation. These attributes are crucial for mastering the nuanced strategies necessary for successful and efficient landings. Thus, Double Q-Learning is well-suited to meet the specific demands of the Lunar Lander problem, making it an excellent choice for this application.

II. Implementation of the Deep Q-Network (DQN) Agent

Our DQN Agent is designed to learn optimal policies in a reinforcement learning setting by utilizing a neural network to approximate Q-values. The core components of our implementation include the Replay Buffer, the neural network architecture, and the learning mechanism which incorporates Double Q-Learning.

A. Replay Buffer

The Replay Buffer plays a critical role in our DQN Agent's learning process. It stores a finite history of experiences, each described by the state, action taken, reward received, the subsequent state, and a terminal flag indicating the end of an episode. We have set the memory size of the Replay Buffer to 128,000. This large buffer size allows the agent to learn from a wide variety of experiences, reducing the correlation between consecutive learning samples and ensuring a diverse range of situations from which to learn.

B. Epsilon Greedy Strategy

The agent utilizes an epsilon-greedy strategy for action selection, balancing exploration of the environment with exploitation of known information. Initially, epsilon is set to 1 to encourage significant exploration. Over time, epsilon decays according to a factor of 0.9999, gradually shifting the strategy towards exploitation of the learned Q-values. The decay continues until epsilon reaches a minimum threshold of 0.03, ensuring that the agent maintains a degree of exploration even as it becomes more confident in its action-value estimates. This approach allows the agent to explore efficiently early on while benefiting from its accumulated knowledge as training progresses. The decay value chosen is large (0.9999), since learning occurs at every step I wanted to train the initial model on enough episodes where significant random exploration occurred.

C. Neural Network Architecture

Our agent's decision-making capabilities are powered by a neural network with three fully connected layers. The network architecture is as follows:

- An input layer that takes 8 input states.
- Two hidden layers, each with 128 neurons and ReLU activation functions, provide the network with the capability to learn non-linear relationships.
- An output layer that predicts the Q-values for each possible 4 actions in the state, facilitating the selection of the action with the highest expected reward.

This setup uses a Mean Squared Error (MSE) loss function to train the network, optimizing the difference between predicted Q-values and the target Q-values derived from the Bellman equation during the learning process.

D. Optimization with the Adam Optimizer

For the training of our neural network, we employ the Adam optimizer, a choice motivated by its robust performance across a wide range of deep learning tasks.

1) Learning Rate Configuration: The learning rate is a critical hyperparameter that controls the step size at each iteration while moving toward a minimum of a loss function. In our implementation, the learning rate of the Adam optimizer is set as a configurable parameter, denoted by 'lr', allowing for flexibility in adjusting the speed of convergence during the training process.

E. Learning Process and Double Q-Learning

The agent's learning process is central to updating the policy network based on interactions with the environment, sampled from the Replay Buffer. Our implementation adopts Double Q-Learning to address the overestimation bias observed in standard Q-Learning. By utilizing two separate networks—the policy network for action selection and the target network for value estimation—Double Q-Learning decouples the selection of actions from their evaluation, leading to more accurate Q-value estimates. The algorithms for action selection and learning are outlined below in Algorithm 1, 2.

Algorithm 1 Epsilon Greedy Action Selection ('get_action')

```
Require: $\epsilon$, $n\_actions$, $current\_state$, $Q$ network
1: Generate a random number $r$ between 0 and 1
2: if $r < \epsilon$ then
3:     $action \leftarrow$ random integer from $0$ to $n\_actions - 1$
4: else
5:     Convert $current\_state$ to tensor and forward through $Q$ network
6:     $action \leftarrow \arg\max_a Q(\text{state}, a)$
7: end if
8: return $action$
```

Algorithm 2 Learning Process with Double Q-Learning ('learn')

```
Require: Replay Buffer $memory$, batch size $batch\_size$, discount factor $\gamma$, target network $Q\_target$, policy network $Q$
1: if the number of experiences in $memory < batch\_size$ then
2:     return without learning
3: end if
4: Sample a minibatch of experiences $(state, action, reward, next\_state, terminal)$ from $memory$
5: Convert $state$, $action$, $reward$, $next\_state$, and $terminal$ to tensors
6: Forward pass $state$ through $Q$ to get current Q-values
7: Select actions for $next\_state$ using $Q$, not $Q\_target$
8: Evaluate these actions with $Q\_target$ to get next Q-value estimates
9: Set next Q-values to 0 for terminal states
10: Calculate target Q-values as $reward + \gamma \times$ next Q-value estimates
11: Compute loss between current Q-values and target Q-values
12: Backpropagate the loss, update $Q$ network weights
13: Update $\epsilon$ with decay
14: Soft update $Q\_target$ weights with a mix of $Q$ weights
```

F. Soft Update Function

The soft update function is a crucial component in the stability of our DQN Agent's learning process. This function methodically updates the weights of the target network (Q_target) by blending them with the weights from the primary network (Q) according to a specified proportion, controlled by the parameter τ.

1) Soft Update Mechanism: The soft update mechanism is governed by the following equation:

$$Q\_target = \tau \times Q + (1 - \tau) \times Q\_target \tag{1}$$

where $0 < \tau \leq 1$ is a hyperparameter dictating the update rate. A smaller value of τ ensures that the target network's weights change more gradually, providing a more stable learning signal to the primary network. This gradual update helps to maintain the consistency of the target Q-values over multiple learning iterations, reducing the variance in the policy and value estimates.

2) Frequency of Soft Updates: Implementing the soft update on every run, as in our agent's learning process, is a deliberate strategy to maintain a consistent yet flexible adaptation of the target network. This approach ensures that the target values evolve smoothly in tandem with the primary network's learning progress. The choice of τ is pivotal in balancing between the stability and adaptiveness of the learning process. A very small τ can slow the learning, as the target network lags significantly behind the primary network. Conversely, a larger τ risks instability due to rapid shifts in the learning targets.

III. Training the Model

To address the Lunar Lander challenge via Deep Q-Learning, our focus centers on optimizing three pivotal hyperparameters: the update rate τ, the learning rate(lr) of the Adam optimizer, and the discount factor γ. These parameters are instrumental in dictating the learning dynamics and eventual success of the model.

The training regimen spans 1,000 episodes, maintaining a batch size of 256. This constant batch size ensures a balance between computational tractability and the diversity of experiences for each training iteration.

Given the intricacies of determining optimal hyperparameter ranges for the Lunar Lander's complex environment, we decided to start with a wide range of values. The idea is to gridmap multiple models and find the first successful model that achieves a moving average score above 200 on the previous 100 episodes during the training process. We can use these hyperparameters as a new base to reduce the range of our gridmap search which might help with computation time. With this in mind we want to gradually decrease τ and lr, and gradually increase γ. The goal being to find the first successful model with larger learning rates and smaller discount factors. The reason being these models due to higher learning rates might be quicker to converge and train.

- **Update Rate** (τ): Values for the soft update mechanism of the target network, with exploration values set within $\{0.01, 0.005, 0.001, 0.0005, 0.0001\}$. Initiating with higher values facilitates an exploration of how frequently the target network should adjust to the learned experiences.
- **Learning Rate of the Adam Optimizer**: Determines the magnitude of weight updates during optimization, chosen from $\{0.01, 0.005, 0.001, 0.0005, 0.0001\}$. Starting with larger values aids in comprehensively navigating the parameter space for efficient learning.
- **Discount Factor** (γ): Influences the valuation of future rewards, selected from $\{0.9, 0.95, 0.99, 0.999\}$. A greater γ emphasizes the importance of future rewards, which is crucial for the long-term strategic planning required in Lunar Lander.

IV. Initial Results

The hyperparameter values for the first successful model were $\tau = 0.01$, $\alpha = 0.001$, and $\gamma = 0.99$.

The results, as illustrated in Figure 1, show the model's score and moving average score over 1000 episodes. Notably, the model achieved a commendable average score of 202 over 100 episodes, which is considered successful for the Lunar Lander challenge. The successful score was achieved on episode 706 of training.

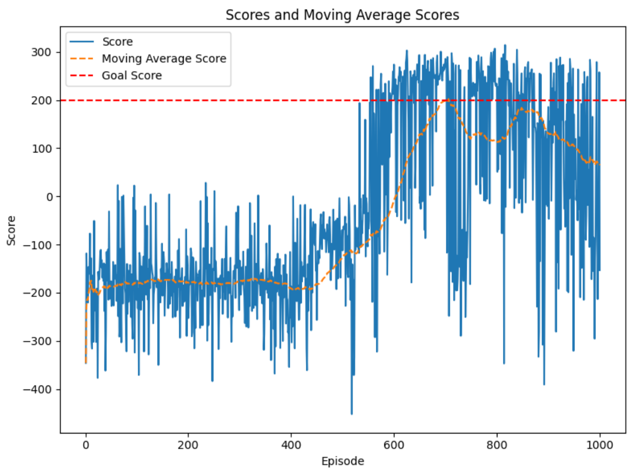

Fig. 1: Score and 100-episode moving average score of the first successful model with $\tau = 0.01$, $\alpha = 0.001$, and $\gamma = 0.99$.

A. Analysis of Initial Results

The analysis of the graph from our initial successful model reveals a rapid policy adaptation to the environment, as indicated by swift changes in the scores over episodes. This observation suggests that the learning process is sensitive to the hyperparameter settings, particularly the update rate (τ) which governs how quickly the agent incorporates new information into the target network.

Given these dynamics, we are narrowing the range of τ for subsequent models to the interval $[0.005, 0.001, 0.0005]$, anticipating a more stable learning progression. The learning rate (lr) is also adjusted to a set of lower values $[0.001, 0.0005, 0.0003, 0.0001]$, with the aim of fine-tuning the agent's ability to converge to an optimal policy without abrupt variations in performance.

Concurrently, we observed that models with γ set at 0.9 underperformed significantly. With this in mind we adjust γ values to the interval $[0.95, 0.99]$.

The updated hyperparameter ranges aim to harness the insights gained from the initial results, steering the search towards a domain where the model not only achieves the target performance but also demonstrates improved stability and consistency in learning.

V. Results and Observations

Figure 2, 3 and 4 showcase the moving average scores for models trained with a constant γ value of 0.99 and varying τ and learning rate (lr) parameters. Each graph represents a set of models with a single τ and γ value and varying lr values .

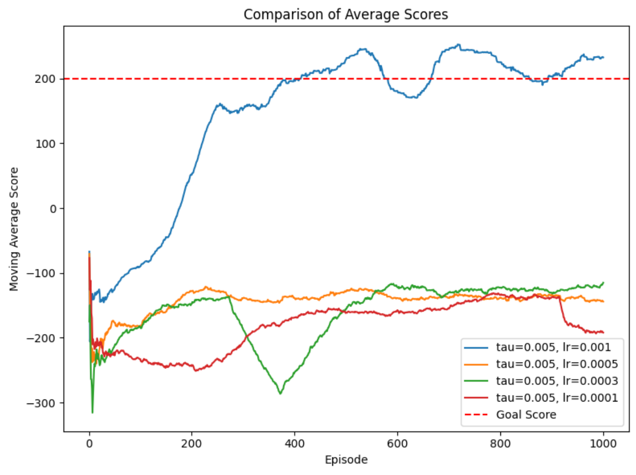

Fig. 2: Comparison of 100-episode moving average scores with $\tau = 0.005$, $\gamma = 0.99$ and different lr values

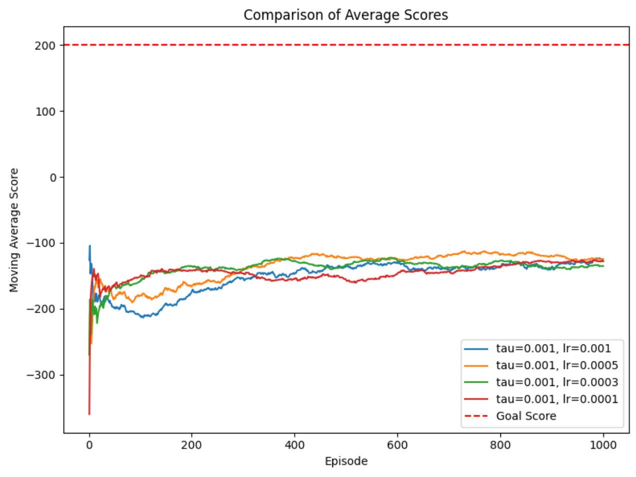

Fig. 3: Comparison of 100-episode moving average scores with $\tau = 0.001$, $\gamma = 0.99$ and different lr values

Out of the two dozen models trained, only two managed to surpass the benchmark with a moving average score above

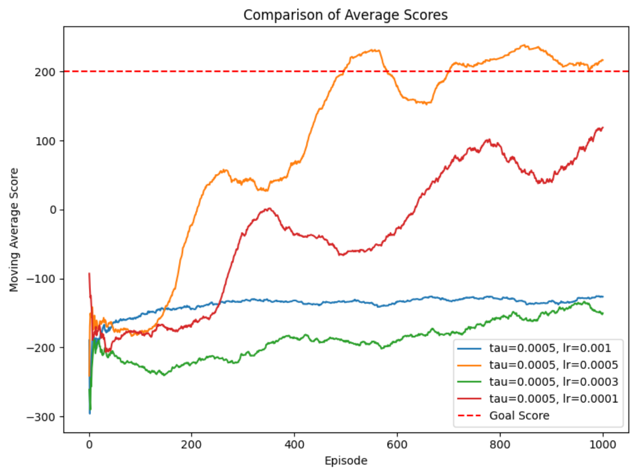

Fig. 4: Comparison of 100-episode moving average scores with $\tau = 0.0005$, $\gamma = 0.99$ and different lr values

200. None of the models with $\gamma = 0.95$ were close to being successful(Hence data not shown). These observations suggest the value of 0.95 is too low for reward propagation. The most successful model, with $\tau = 0.005$ and lr = 0.001, achieved a peak average score of 252 (Figure 2). Another model, with $\tau = 0.0005$ and lr = 0.0005, reached a peak average score of 238 (Figure 4). The states of these two models were saved at their 100-episode moving average score peaks for future analysis.

A. *Dependency on Favorable Early Learning Outcomes*

The training process for our models has revealed a dependency on favorable outcomes during the early learning stages, where the epsilon parameter is set high to promote exploration. Our experiments indicate that when the model benefits from advantageous experiences early on, there is an increased likelihood of training a winning model. The models don't consistently converge to winning models. However, we are able to build two potentially winning models. For future consideration, I would like to look at how ϵ and ϵ-decay might impact our success rate of building a winning model.

B. *Performance Peaks and Subsequent Drop-offs*

A consistent pattern observed in our winning models is their tendency to reach a peak in moving average scores, followed by a notable decline before regaining high performance levels. This phenomenon raises intriguing questions about the underlying factors influencing these fluctuations. It would be interesting to see if changing ϵ and ϵ-decay has an impact on this.

VI. Model Evaluation

Following the identification of two promising models from the training phase, we proceeded to evaluate their performances in a test phase consisting of 1000 episodes each, with no further learning. The first model, which had reached a peak average score of 252 at episode 721, and the second model, with a peak average score of 238 at episode 848, were loaded from their saved states.

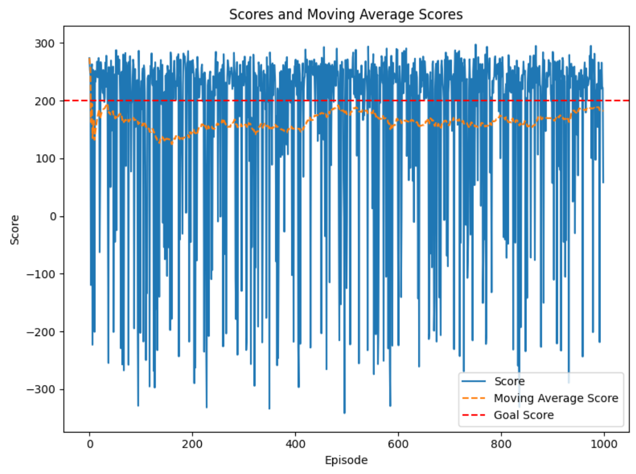

Fig. 5: Performance of the first model over 1000 episodes.

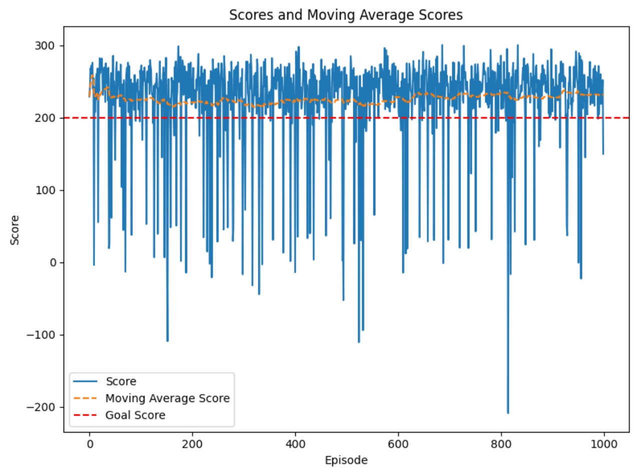

Fig. 6: Performance of the second model over 1000 episodes.

The test results presented an intriguing contrast between the two models. As illustrated in Figure 5, the first model exhibited significant volatility in its scores, failing to consistently replicate the high average score achieved during training. This erratic behavior, characterized by many large negative scores, suggests that the model may have been overly tuned to specific scenarios encountered during training, leading to poor generalization and a failure to maintain the benchmark average score of 200.

Conversely, as shown in Figure 6, the second model demonstrated remarkable consistency and robustness, maintaining an average score above 200 throughout the entire 1000 episodes, and rarely producing a score below zero.

These observations highlight the intricate nature of reinforcement learning models, where a higher training score does not always translate to better general performance. It also underscores the importance of thorough testing across a broad range of episodes to accurately assess a model's capability.

These results prompt a reflection on my decision of saving models based solely on their highest achieved moving average scores. As evidenced by the evaluation phase, a model's peak performance during training is not always indicative of its future consistency. This suggests that other metrics, perhaps the standard deviation of the moving average, might provide a more comprehensive view of a model's reliability.

The first model, which had a peak average score of 252, exhibited a high standard deviation of scores around 157.23, reflecting significant volatility and inconsistent performance. In contrast, the second model achieved a lower peak average score of 238 but with a substantially smaller standard deviation of approximately 65.89, indicating a more stable and consistent behavior.

The lower standard deviation of the second model suggests that its performance is more reliable over time, not just at the moment of peak average scoring.

VII. Conclusion

The exploration into the Lunar Lander environment using Deep Q-Learning has yielded insights into the role of randomness in the success of reinforcement learning models. One of the key takeaways from our experiments is the importance of randomness in guiding models towards successful outcomes. This observation suggests that fine-tuning the ϵ-decay parameter could be pivotal. A more dynamic approach, where ϵ decays only in response to improvements in the model's average score, may enhance the likelihood of building a winning model.

Retraining models with identical hyperparameters did not always result in a winning state within 1000 episodes, highlighting the inherent variability and stochastic nature of the learning process.

Factors such as the standard deviation and the consistency of scores should be considered when determining the ideal checkpoints for model preservation.

For future experiments the effects of ϵ-decay strategies can be investigated.

References

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.
[2] "Lunarlander," https://gym.openai.com/envs/LunarLander-v2/.
[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.
[4] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.