Exploring the Effectiveness of DQN, Dueling DDQN and PPO Algorithms on the Lunar Lander

*Note: CS5180 Project Report 2023

Hussain Kanchwala
Mechanical Engineering
Northeastern University
Boston, US
kanchwala.h@northeastern.edu

***Abstract*—Reinforcement Learning (RL) constitutes a subset of machine learning devoted to empowering an agent to effectively navigate an environment characterized by uncertainty, with the overarching goal of maximizing a cumulative, long-term reward. This study implements and scrutinizes three distinct RL techniques, namely DQN, Dueling DDQN and PPO, within the LunarLander-v2 environment provided by OpenAI Gym. The most successful Dueling DDQN model yields an average return of 170.85 for the agent, whereas DQN achieves an average return of 151.55 and PPO accumulates a return of 134.96 for the agent in the original problem.**

***Index Terms*—DQN, Dueling DDQN, PPO**

GitHub:-

I. Introduction

In recent years, reinforcement learning [1] has demonstrated notable success across various applications, notably in the realm of robotic control [2], [3]. Diverse methodologies have been suggested and applied to address these challenges [4], [5]. This study focuses on tackling a prominent robotic control issue, specifically the lunar lander problem, employing distinct reinforcement learning algorithms. Subsequently, we assess the performance of the DQN, Dueling DDQN and PPO algorithms on the lunar lander environment and provide a detailed analysis.

II. Problem Definition

Our objective is to address the lunar lander scenario within the OpenAI Gym toolkit by leveraging reinforcement learning techniques [1]. This environment replicates a scenario in which a lander must safely touch down at a designated location amid low-gravity conditions, featuring a precisely defined physics engine. The primary objective of the task is to guide the agent in accomplishing a soft and fuel-efficient landing on the designated pad. The state space mirrors real-world physics with continuous characteristics, while the action space is discretized.

Identify applicable funding agency here. If none, delete this.

III. Related Work

Previous research has delved into addressing the lunar lander environment employing various methodologies. In [6], modified policy gradient techniques are employed to evolve neural network topologies. [7] adopts a control-model-based approach, focusing on learning optimal control parameters rather than the system's dynamics. Meanwhile, [8] investigates spiking neural networks as a solution for OpenAI virtual environments. Consequently, our objective is to tackle the lunar lander problem using DQN, Dueling DDQN and PPO.

IV. Environment Model

A. Framework

The framework employed to address the lunar lander problem is Gym, a toolkit developed by OpenAI [9] specifically designed for creating and comparing reinforcement learning algorithms. Gym facilitates the creation of environments across a spectrum of learning scenarios, encompassing domains such as Atari games and robotics. In this context, the simulator utilized is named Box2D, and the specific environment is referred to as LunarLander-v2.

B. Observations and State Space

The landers state space comprises 8 variables, delineated as follows:
1. x-coordinate of the lander
2. y-coordinate of the lander
3. v_x, the horizontal velocity
4. v_y, the vertical velocity
5. θ, the orientation in space
6. v_θ, the angular velocity
7. Left leg touching the ground (Boolean)
8. Right leg touching the ground (Boolean)

All coordinate values are referenced relative to the landing pad rather than the lower left corner of the window. The x-coordinate is measured from the line connecting the center of the landing pad to the top of the screen, with positive values on the right side and negative values on the left. The y-coordinate is positive above the landing pad and negative below it.

C. Action Space

There are four discrete actions available: do nothing, fire left orientation engine, fire right orientation engine, and fire main engine. Firing the left and right engines induces a torque on the lander, leading to its rotation and complicating the stabilization process.

D. Reward

Reward for moving from the top of the screen to the landing pad and coming to rest is about 100-140 points. If the lander moves away from the landing pad, it loses reward. If the lander crashes, it receives an additional -100 points. If it comes to rest, it receives an additional +100 points. Each leg with ground contact is +10 points. Firing the main engine is -0.3 points each frame. Firing the side engine is -0.03 points each frame. Solved is 200 points.

With this reward setup, we want the agent to reduce its distance to the landing pad, land on ground without crashing and fire the main and side engines appropriately so to reduce the cost.

V. Algorithms Approaches

A. DQN

The Deep Q-Network (DQN) algorithm is a variant of the Q-learning algorithm that uses a neural network to estimate the Q-value function. It has been successfully applied to various reinforcement learning tasks, including the Lunar Lander environment [8],[9].

This algorithm uses a neural network to approximate the Q-value function. It maintains two networks: a target network and a local network. The target network is used to calculate the target Q-values for the Bellman update, while the local network is used to select actions and estimate the Q-values. The target network is updated periodically with the weights of the local network [10].

It also uses a replay buffer to store past experiences. This buffer stores tuples of states, actions, rewards, next states, and done flags. When updating the Q-values, experiences are sampled randomly from the buffer. This approach helps to break the correlation between consecutive experiences and stabilize the learning process [8].

DQN has several advantages. Firstly, it can handle complex environments with large state and action spaces. Secondly, it can learn directly from raw pixel input, which makes it suitable for tasks like image-based Atari games [9].

Despite its advantages, the DQN algorithm has some limitations. It can be sensitive to hyperparameters and may require a lot of computational resources to train, especially for complex environments. Also, the DQN algorithm can struggle with tasks that require long-term planning or tasks where the optimal policy is non-stationary [9].

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Fig. 1. Pseudo code for DQN

1) Detailed Implementation: The neural network architecture employed in this study comprises three linear layers, each serving a distinct role in the reinforcement learning framework. The initial layer processes the input state and transforms it into a 256-dimensional hidden layer. Subsequently, the second layer accepts the 256-dimensional hidden layer as input and outputs another 256-dimensional hidden layer. The final layer, responsible for generating Q values corresponding to the available actions, takes in the 256-dimensional hidden layer produced by the second layer. It is worth noting that Rectified Linear Unit (ReLU) activation functions are applied to each layer except for the last one.

The optimization process involves the utilization of the mean square error loss function, which quantifies the disparity between the target values and the values generated by the local network. This discrepancy serves as the basis for updating the weights of the local neural network, facilitating a refinement of the local model's performance every 4 iterations. The target neural network is updated with the weights of the local neural network every 10000 iterations.

To facilitate the training process, a batch size of 64 is employed, and a replay buffer of size 200,000 is utilized. This replay buffer is prepopulated with 50,000 samples, providing a diverse set of experiences for the neural network to learn from. The temporal discount factor (gamma) is set to 0.99, reflecting a consideration of future rewards in a dynamic environment.

The exploration-exploitation trade-off is addressed through the implementation of an exponentially decreasing epsilon-greedy strategy. The exploration parameter epsilon starts at 1 and linearly decays to 0.01 over the course of 1,000,000 iterations.The training regimen encompasses 6,000 episodes, during which the neural network refines its understanding of the environment and optimizes its policy.

B. Dueling DDQN

Dueling Double Deep Q Network (DDQN) is a variant of the Double Deep Q Network (DDQN) that aims to improve the learning process in reinforcement learning tasks. It separates the estimator of state values and state-dependent action advantages into two separate streams, which can be beneficial

in certain scenarios where it is unnecessary to know the value of each action at every time step [12].

The key motivation behind using Dueling DDQN is, scenarios where multiple actions in a given state might have similar values. This can lead to unstable updates, causing inflated Q-values. By decoupling state values and action advantages and subtracting the mean advantage from individual action advantages, Dueling DDQN ensures more stable and accurate Q-value estimations. This technique helps prevent overestimation bias, especially in situations where actions are closely ranked in value.

The implementation of Dueling DDQN involves creating two networks, a DQNetwork and a Target Network. The DQNetwork parameters are copied to the target network during the training process. The target network is updated with the DQNetwork every tau step, where tau is a hyper-parameter that we define [11]. The advantage of Dueling DDQN along side the above mentioned is that Dueling DDQN introduces a distinct target network to ameliorate the overestimation bias inherent in Q-value estimation.

This methodological improvement contributes to more reliable and accurate reinforcement learning outcomes, especially in scenarios where the inherent complexities of the environment may lead to inflated Q-value estimates.

Algorithm 2 Operation of the implemented Dueling Double DQN Agent

Initialise agent network with random parameters θ
Initialise target network with random parameters θ'
Initialise memory M with capacity m
Define frequency N for copying weights to target network
for *each episode* **do**
  measure initial state s_0
  while *episode not done* **do**
    choose $a_t = \pi(s_t)$ according to ϵ-greedy policy
    implement a_t and advance until next action needed
    measure s_{t+1}, and calculate r_t
    store transition (s_t, a_t, r_t, s_{t+1}) in M
    calculate TD error $\delta = r + \gamma \max_{a_{t+1}} Q_{\theta'}(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t)$
    calculate priority sampling weight and store in M
    $s \leftarrow s_{t+1}$
  end
  $b \leftarrow$ sample batch of transitions from M according to priority weights
  for *each memory* $m_i = (s_i, a_i, r_i, s_{i+1})$ *in* b **do**
    $\hat{y}_t = r_t + \gamma Q_{\theta'}(s_{t+1}, \text{argmax}_a Q_\theta(s_{t+1}, a))$
  end
  Stochastic Gradient Descent on θ over all $(x_i, y_i) \in b$
  if *number of episode is multiple of* N **then**
    $\theta' \leftarrow \theta$
  end
end

Fig. 2. Pseudo code for Dueling DDQN

1) Detailed Implementation: A 3 linear layer neural network with hidden layers of size 256, with the first layer having the input the size as that of size of state is implemented. The last layer of the neural network is divided into two streams that is it outputs value of the state as well as values of all actions that can be taken in that state. The ReLU activation function is used for the layers except for the last one.

The target value is computed as follows:

$$tv = r_t + \gamma * Q_{\bar{\theta}}(s_{t+1}, argmax_a Q_\theta(s_{t+1}, a)) \quad (1)$$

As in the case of DQN the estimation as well as the action selection was done by the target network but in the case of Dueling DDQN this process is decoupled where the action selection is done by the DQNetowrk and the estimation of value is done by the Target Network. This helps to overcome the overestimation bias.

Parameters such as Batch size, Replay Memory, gamma, Replay memory prepopulation size, Target Network update rate and number of episodes to train the model is maintained the same as in the DQN case. In order to explore the exponentially decaying epsilon from 1 to 0.01 in 1000000 is used.

C. PPO

Proximal Policy Optimization (PPO) is a type of policy optimization method used in reinforcement learning (RL). PPO, introduced by OpenAI, has become popular due to its effectiveness and simplicity. It is an on-policy algorithm, meaning it learns the policy that is used to collect data during training.

PPO attempts to take the best of both worlds from Actor-Critic Policy Optimization (ACPO) and Trust Region Policy Optimization (TRPO). ACPO is simple and has low variance, but sometimes it can lead to poor performance because it takes too large policy updates. On the other hand, TRPO has a mechanism to avoid too large updates, but it is more complicated and computationally expensive. PPO proposes a solution that is as effective as TRPO in terms of sample complexity and performance, but as simple to implement as ACPO.

In essence, PPO tries to keep the new policy close to the old policy. This is achieved by adding a constraint to the policy optimization problem. The objective function that PPO optimizes has an additional term, which penalizes large deviations from the old policy. This ensures that the updates don't stray too far from the current policy, thus making the learning process more stable.

PPO has several advantages. First, it is relatively simple to implement compared to other sophisticated policy optimization methods. Second, it is sample efficient which means it requires fewer interactions with the environment to learn a good policy. This is particularly beneficial in real-world scenarios where collecting data is costly or time-consuming. Third, PPO is very stable and robust. It works well across a wide range of tasks without much hyperparameter tuning.

However, PPO also has some limitations. While it is more stable than some policy optimization methods, it can still suffer from instability and divergence in some cases, especially in environments with high-dimensional state spaces. Furthermore, PPO is an on-policy method, which means it

can't reuse old data, making it less sample-efficient than off-policy methods. Lastly, tuning the hyper-parameters for the constraint in the objective function can sometimes be tricky and task-dependent.

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Fig. 3. Pseudo code for PPO

1) Detailed Implementation: The actor and critic is a 3 layered Neural Network with activation function ReLU and hidden layer of size 256. The first layer takes in input the state and the last layer of actor help providing the probabilities of actions at a given state. The output of the critic network is the value of the state.

The networks are trained after every 300 steps for 10 times by forming 3 batches of size 100 each. Each and every time the 3 batches are randomly generated and once the weights of actor and critic are updated the memory from which the batches were formed is cleared. A policy clip of 0.2 is used which is a very important hyperparameter that can be manipulated and the learning is very dependent on this.

The number of epoches for which the model is trained is 6000 and the γ value used 0.99 with a learning rate of 0.0001.

VI. RESULTS

The discounted sum of rewards over the episode length is used to calculate the returns at the end of each episode.

The average returns collected at the end of last 1000 episodes for the case of DQN, Dueling DDQN and PPO are as follows 151.55, 170.85 and 134.97. The returns don't just let us know whether the agent is learning or not but it also helps us learn very crucial points about the policy being learned by each of the algorithms which I will be discussing further.

A. DQN Results

In figure (4), the case of DQN it is observed that the returns are highly fluctuating in last 1000 episodes, this says that a sub-optimal policy is being learned by the algorithm. Though it is providing a high return of 152 a stable policy isn't learn which produces reliable results thereby giving the user confidence that the agent will make good decisions over time.

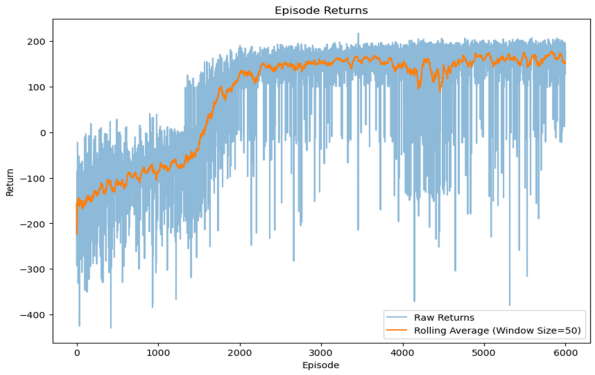

Fig. 4. Returns by DQN

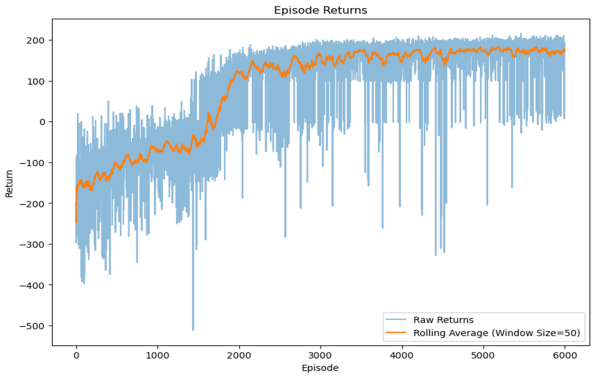

Fig. 5. Returns by Dueling DDQN

B. Dueling DDQN Results

Figure (5), the case of Dueling DDQN it is seen that this fluctuation has considerably decreased compared to the DQN case there by showing the effectiveness of decoupling the action selection and estimation along with incorporating the change in structure of calculating the Q value for the LunarLander case. This has resulted in a much more stable policy being learned as well as provided an increased returns when compared to the DQN case.

C. PPO Results

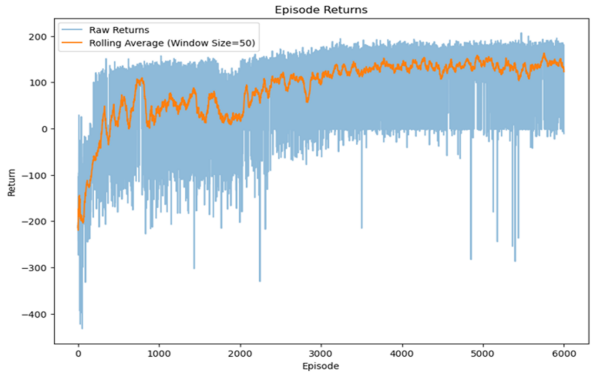

Fig. 6. Returns by PPO

Figure (6) we can observe the returns of PPO algorithm. Keeping the focus on the fluctuating returns PPO results in a much stable policy being learned than DQN and Dueling DDQN algorithms. But the average returns over the last 1000 episodes is comparatively less when compared to DQN and Dueling DDQN case, this highlights some of the major drawbacks of PPO that arise mainly due to non resuability of data that is being generated and also the effect of not tuning the hyper parameters correctly.

D. Comparision

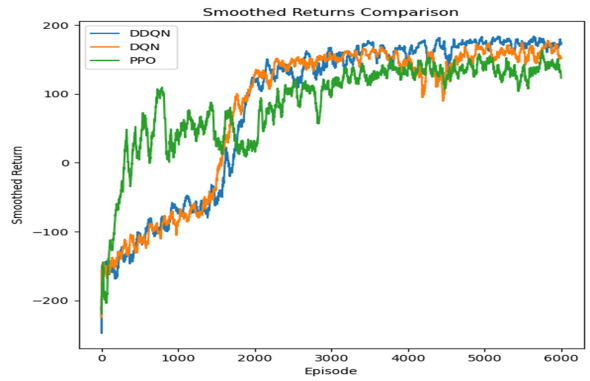

Fig. 7. Returns Comparison

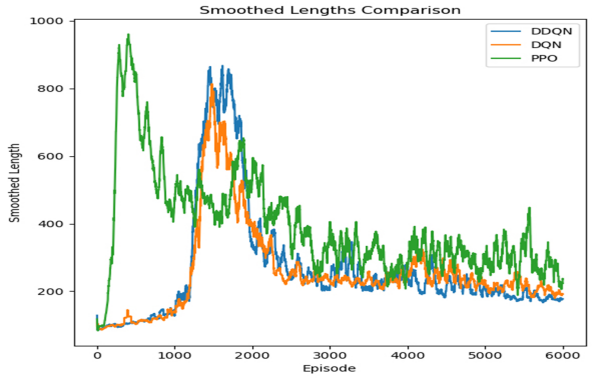

Fig. 8. Lengths per Episode Comparison

Looking at the figure (7) and figure (8) it is evident that the PPO learns policy much faster than the DQN and Dueling DDQN case which highlights the sample efficiency of PPO. Focusing on figure(8) it is observed that the PPO agent learns to avoid crashing and remain in air for the maximum episode time step which is intermediate point of the learning curve seen much earlier in PPO within 1000 episodes than the other two algorithms that learn this policy at 1500 to 2500 episodes. A comparative showcase of the stable policy learned by PPO was also done in the above subsections. Now for the performance aspect of PPO to provide higher returns comes down to tuning the hyper parameters which is a negative point of this algorithm when compared to DQN and Dueling DDQN where the number of hyper parameters are less when compared to the PPO case.

VII. CONCLUSION

While DQN and Dueling DDQN can be used to solve the LunarLander-v2 problem, PPO might be a better choice due to its stability, efficiency, and performance provided that the hyper parameters are tuned properly.

REFERENCES

[1] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction. MIT press, 2018.

[2] J. Kober, J. A. Bagnell, and J. Peters, “Reinforcement learning in robotics: A survey,” The International Journal of Robotics Research, vol. 32, no. 11, pp. 1238–1274, 2013.

[3] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, “Reinforcement learning for robot soccer,” Autonomous Robots, vol. 27, no. 1, pp. 55–73, 2009.

[4] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” 1996.

[5] C. Szepesv´ari, “Algorithms for reinforcement learning,” Synthesis lectures on artificial intelligence and machine learning, vol. 4, no. 1, pp. 1–103, 2010.

[6] A. Banerjee, D. Ghosh, and S. Das, “Evolving network topology in policy gradient reinforcement learning algorithms,” in 2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP), Feb 2019, pp. 1–5.

[7] Y. Lu, M. S. Squillante, and C. W. Wu, “A control-model-based approach for reinforcement learning,” 2019.

[8] K. Jackson, ”Deep Q-Network (DQN) with Lunar Lander,” Kyle Jackson, 2021. [Online]. Available: https://kylejackson.org/dqn/rl/2021/12/29/lunar_lander.html.

[9] R. B. Diddigi, G. Nangue Tasse, Y. Vidal, S. Krishnagopal, S. Rajaee, ”Deep Q-Learning on Lunar Lander Game,” Neuromatch Academy: Deep Learning, 2021.

[10] A. M. Al-Afifi, ”Deep Q-Learning on Lunar Lander Game,” ResearchGate, 2020.

[11] ”Deep Reinforcement Learning,” IEEE Xplore, 2023.

[12] M. Sewak, ”Deep Q Network (DQN), Double DQN, and Dueling DQN,” Deep Reinforcement Learning, Springer, Singapore, 2019.