

# Deep Reinforcement Learning for Motion Simulation

Meet Savla<sup>1</sup> and Dr. Mainak Adhikari<sup>2</sup>

<sup>1</sup>Department of Computer Science, Indian Institute of Information Technology Lucknow

<sup>2</sup>School of Data Science, Indian Institute of Science Education and Research Thiruvananthapuram.

**Abstract**—This paper examines the utilization of Double Deep Q-Learning in fine-tuned control of lunar lander in OpenAI Gym Lunar Lander v2. A version of the Double Deep Q-Network (DDQN) framework built on PyTorch using Multi-Layer Perceptron (MLP) as a Q-network to moderate Overestimation Bias in Single DQN is put forth. Therefore, the present study seeks to improve stability and performance in autonomous lunar lander control systems essential for space exploration.

The recent past has seen some major advancements in autonomous landing systems, but still optimal performance remains challenging. This gap is filled by developing and tuning DDQN models, formulating reward systems and investigating hyperparameters inside PyTorch. Model progress was evaluated based on landing success rate, total score, minimizing landing velocity and touchdown precision.

The proposed DDQN model outperforms other models as demonstrated by results obtained from training with two MLP hidden layers with 256 neurons each thus it can be considered stable and precise for landings. Consequently, after extensive trials, this model scored consistently above 200 in just 1500 episodes which greatly reduced the training time. Conversion from TensorFlow to PyTorch increased training speed by 100 times.

This paper adds to the field of Deep Reinforcement Learning (DRL), showing that it is capable of handling complicated practical cases such as self-piloting space shuttles, hence opening up possibilities for more utilization of DRL in space contexts.

**Keywords:** Double Deep Q-Network (DDQN), PyTorch, Deep Reinforcement Learning, Lunar lander control

## I. INTRODUCTION

To achieve sustainable and efficient exploration of space, autonomous lunar landings are essential. The Moon acts as a critical stepping stone in enabling deep space missions to Mars and beyond. Additionally, it has become essential to develop strong and efficient spacecraft landing algorithms for governmental space agencies (such as NASA and ISRO) as well as private agencies (for example Blue Origin and SpaceX), which continue with their moon exploration efforts. A successful and accurate landing on the moon is necessary for deploying scientific instruments, setting up bases for exploration, as well as ensuring future human missions are viable.

However, the lunar environment challenges the conventional control approaches substantially. Assuring safe touchdowns in varied landscapes such as craters, mountains or planes requires

immediate judgments. Moreover, the thin atmosphere provides minimal aerodynamic drag that necessitates having specific thrust control for the engines to regulate descent velocity at impact point since a softer touch down is required. Moreover, weak lunar gravity makes traditional landing techniques using parachutes impracticable hence calling for propulsive maneuvers into a controlled descent.

Hence, there is a necessity for sophisticated control algorithms that can deal with the lunar environment uncertainties and complexities. In this connection, Deep Reinforcement Learning (DRL) [1] seems to be a good method for addressing the problem. DRL overcomes the drawbacks of conventional approaches that depend on predefined models of the lunar system through trial and error by training an agent to learn optimal policies in simulated environments. This study is to investigate the employment of DRL, in particular using Double Deep Q-Network (DDQN) architecture to achieve accurate and fuel-saving moon landing. The triumph of this approach is significant for forthcoming missions aimed at the moon and it shows the potential of Reinforcement Learning in aerospace applications.

### A. Problem Definition

This research concentrates on the establishment of a powerful form of Deep Reinforcement Learning (DRL) model that can be used for controlling a lunar lander through using OpenAI Gym Lunar Lander v2 environment [2] with Double Deep Q-Network(DDQN). The problem is about enabling an agent to learn the best thrust and orientation maneuvers in low gravity circumstances. To achieve fuel efficient, safe landings, it is important that the model remains stable and converges during training. This research seeks to enhance the dependability of autonomous moon landing systems via state of the art DRL techniques.

### B. Limitations of Traditional Control Methods

Traditional control approaches to lunar landings face many difficulties, which are due to the complexities in the lunar environment:

- **High Dimensionality and Non-Linearities:** Linear systems with known dynamics such as PID controllers are traditional techniques. The problem of landing on the moon is high-dimensional, non-linear and highly uncertain.

- **Inability to Handle Minimal Atmosphere:** The Moon's atmosphere provides extremely low aerodynamic drag. In this case, the conventional approaches cannot provide accurate control of engine thrust which is necessary for a soft landing.
- **Impracticable Specifications:** The optimal control policies that are based on classical control algorithms such as LQR call for a precise mathematical model of the system.
- **Manual Tuning and Computational Cost:** Traditional procedures are so sensitive to parameters that one might have to spend much time manually adjusting them which equates to expensive computation.

These limitations collectively make traditional control approaches insufficient to achieve robust and efficient autonomous lunar landings.

### C. Deep Reinforcement Learning

DRL is a powerful way to train agents to deal with intricate, changing environments without necessarily programming policy controls directly. An agent in a DRL framework observes its environment through a state representation - a compressed data structure, enclosing important information about the environment. Based on this state (action space), the agent then chooses, from among all possible actions, an act. Then, the environment responds to the action chosen by moving to a new state and giving feedback as a reward to the agent. This scalar rewarding signal works as feedback telling how impactful was that particular action in achieving the overall objective. This makes it ideal for problems such as autonomous lunar landing control involving real time decision making and adaption for success where traditional approaches fail due to their inability to cope with complex and dynamic situations.

### D. OpenAI Gym Environment

A very popular benchmark for lunar landing control problem that has been used over time to evaluate and develop RL algorithms is the OpenAI Gym Lunar Lander v2 environment. The simulation in this environment is highly detailed reproducing all complex physical dynamics of a moon landing including lunar gravity ( $g = 1.625 \text{ m/s}^2$ ), controllable lander thrust with specific impulse and detailed interactions with the lunar surface.

- **State Space:** The lander's current state is represented by an 8-dimensional continuous state space,  $S$ . This extensive state representation enables DRL agents to have a complete understanding of how the lander behaves dynamically.
- **Action Space:** The actions possible in the environment are binary and they can only take values from 0 to 3 as indicated by  $\{0, 1, 2, 3\}$ . Thus, during the descent stages of landing this set of discrete actions will enable DRL agents to steer or control trajectories in midair.
- **Reward Structure:** There is a thin reward structure  $R: S \times A \rightarrow R$  in the environment that encourages behaviors leading to a successful and efficient landing.

- **Episode Termination:** There are two main reasons why episodes end:
  - **Successful Landing:** Occurs if both feet of the lander lie within a defined area on lunar ground.
  - **Crash Landing:** Involves severe contact between the lander and the moon's surface resulting in huge penalties.

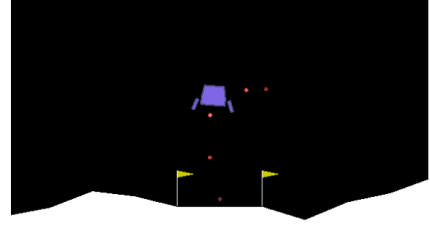


Fig. 1: Lunar Lander Environment. Courtesy: Original

### E. Double Deep Q-Network (DDQN) Solution

An superior approach in Deep Reinforcement Learning (DRL) is the use of Deep Q-Networks (DQNs) that allows for the agents to learn optimal control policies from their interaction with the environment. These neural networks are used by DQNs to approximate state-action value functions referred to as Q-values. The purpose of these Q-values is to help in approximating how much an agent can get in future rewards it accumulates if it acts within a particular state. The DQN progressively learns over several iterations when it associates actions that have higher expected future rewards into its optimal policy.

#### Overestimation Bias in Single DQN:

Single DQNs are a plausible way to learn control policies but over-optimistic approximation can undermine these approaches. This optimistic bias arises when the same neural network is used for action choice (exploitation) and Q-value calculation (target) while training.

#### Double DQN as a Solution to Overestimation Bias:

The Double DQN (DDQN) uses two networks: one for estimation and another for target. In the main network, all possible actions are evaluated in terms of their Q-values of state. On the other hand, during the update of the Q-value, the target network that has its weights synchronized from the main network is used to estimate the value of the chosen action. De-coupling between action selection and target Q-value estimation serves as a mechanism to reduce over-optimism by providing more stable targets for updates. The target network's weights are updated periodically using weights from the main network to ensure it is in line with the current policy learned but not too frequently such that it becomes stale. This enables better learning stability and convergence to optimal control policies.

## II. LITERATURE REVIEW

### A. Traditional Control Approaches

The controlling systems that independent systems rely on include Proportional Integral Derivative (PID) controllers and

Linear Quadratic Regulators (LQRs). For example, a PID controller adjusts its input by observing the state deviations but may face difficulties in dynamic, nonlinear conditions such as those of the lunar surface due to fixed terms. The same case applies to LQR which optimizes control inputs based on cost function at the expense of complex system models which are practically infeasible for moon landing purposes as it may entail changes in the topography and sensor noise. Furthermore, LQRs require extensive manual tunings since they are vastly affected by changes in parameters. Researchers have thus developed adaptive PID and robust LQR controllers that offer improved performance in uncertain environments thus addressing some of these issues.

### *B. Deep Reinforcement Learning in Autonomous Motion and Aerospace*

Deep Reinforcement Learning (DRL) [3] is transforming the aerospace industry's autonomous motion control capabilities by dealing with intricate scenarios and dimensions.

- **Motion Control:** To address dexterity issues when designing robotic hands, Lillicrap et al. (2015) employed DDPG, while Schulman et al. (2015) created walking robots that could negotiate tough terrains using PPO.
- **Autonomous Vehicles:** Levine et al. (2016) for instance, demonstrated the use of DQN on lane following and obstacle avoidance simulated in a Deep Reinforcement Learning illustration of such devices using cameras' images only. As a result, Bansal et al.(2018) improved vehicle dynamics' control efficiency through a combination of Model-Predictive Control and DRL.
- **Aerospace Applications:** In the field of aerospace too, DRL looks promising as applied to UAVs. For example, Zhang et al.(2020), have put up a Deep Reinforcement learned based trajectory planning framework on UAVs for autonomous navigation through obstacles in cluttered spaces with collision avoidance capability; Amodei et al.,(2016), also assessed its potential utility among real-world concepts such as precision flight during aerial refueling missions involving unmanned combat air vehicles or tankers which incorporates precise control and real-time decision-making capabilities required for this task.

### *C. Deep Q-Networks*

Deep Q-Learning (DQN) (Mnih et al., 2015) is a major improvement in reinforcement learning (RL), using deep neural networks for state representation. DQN approximates the Q-value function, estimating future rewards for state-action pairs directly from raw sensory data. It also eliminates manual feature engineering that was a RL bottleneck beforehand. To predict Q-values, DQN trains a deep neural network called the Q-network by using experience replay. Consequently, research into DL has progressed with other approaches like Double DQN (Van Hasselt et al., 2016) which rectifies over-estimation bias and Dueling DQN (Wang et al., 2016) which enhances the value estimation process. Despite its data-hungriness &

exploration-exploitation dilemma, DQN remains one of the key foundational blocks to deep RL as it empowers agents to solve challenging problems right from the raw sensor input.

### *D. Applications Beyond Standard Robotics*

Apart from robotics some of the applications include:

- **Process Control:** Real-time optimization of industrial processes.
- **Traffic Signal Control:** Changing traffic lights dynamically to decrease traffic jams.
- **Power Grid Management:** Equilibrium between supply and demand on power grids Another recent extension is autonomous driving and smart grid management demonstrating their heterogeneousness and efficacy in highly dynamic environments.

### *E. Deep Reinforcement Learning and DQN Limitations*

Deep Q-Networks (DQNs) are a combination of neural networks and Q-learning that can be used by agents in dealing with huge state spaces. However, DQNs have to deal with the overestimation bias which is due to the use of one network for both action selection and value evaluation thereby leading to suboptimal control. Another problem arises when trying to balance exploitation and exploration during training thus resulting in bad strategies. The second challenge DQNs face is continuous control tasks that have infinite action spaces. Although techniques such as discretizing actions or using Deep Deterministic Policy Gradients [4] (DDPG) may help, they make things more complicated. Innovations like Rainbow DQN which incorporate several improvements including prioritized experience replay and distributional Q-learning yield better performance and improved stability in comparison to other methods.

### *F. Addressing DQN Limitations*

To enhance DQN performance, a variety of approaches are suggested by researchers.

- **Overestimation Bias:** Methods such as Double DQN and Dueling DQN split value estimation, thereby stabilizing the learning process.
- **Exploration-Exploitation Trade-off:** Some of the strategies that could be used include epsilon-greedy exploration with decay and prioritized experience replay to balance between exploitation and exploration.
- **Continuous Control Challenges:** These can be managed through DDPG and actor-critic among other ways of improving exploration and control. Over time Twin Delayed DDPG (TD3) and Trust Region Policy Optimization (TRPO) have also been developed to enhance continuous control ability.

### *G. Current Research Gaps and Future Directions*

Several challenges continue to face DRL when applied for the Lunar Lander problem. These demands would often bottleneck due to high computational requirements, posing a

significant demand for resources at times, as well as time dedication. Furthermore, there are complexities involved in transferring trained models from simulation environments into real world situations due to unmodeled dynamics and sensor noise. One important research question is how to guarantee robustness and adaptability of DRL algorithms across diverse landing scenarios without having to retrain extensively.

### III. RESEARCH METHODS

This section will give a comprehensive description of the technical details that have been used for research methodology to create and improve the Double Deep Q-Network (DDQN) model for the Lunar Lander problem. The approach includes migrating from TensorFlow Keras into PyTorch, implementing different models for comparison and evaluating them afterwards.

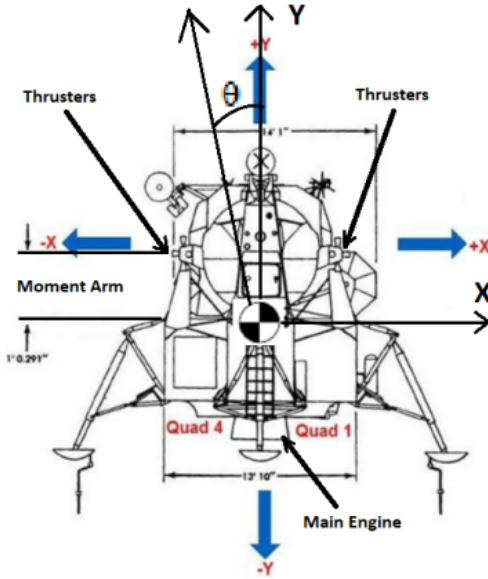


Fig. 2: Lunar Lander Diagram. Courtesy: NASA, Karl Dodenhof

#### A. Framework

To solve the lunar lander problem, Gym is used. This is a toolkit created by OpenAI for making and comparing reinforcement learning algorithms. It can be used to simulate different problems, from Atari games to robotics. The simulator in this case is named Box2D and the environment is called LunarLander-v2.

##### 1) State Space:

$$state \rightarrow \left\{ \begin{array}{l} \text{The lander's } x \text{ coordinate} \\ \text{The lander's } y \text{ coordinate} \\ \text{The horizontal velocity } v_x \\ \text{The vertical velocity } v_y \\ \text{The angular orientation } \theta \\ \text{The angular velocity } v_\theta \\ \text{Left leg ground contact (Boolean)} \\ \text{Right leg ground contact (Boolean)} \end{array} \right\}$$

##### 2) Action Space:

Action	Description
0	Take no action
1	Left orientation engine fire
2	Main engine fire
3	Right orientation engine fire

TABLE I: Action Menu

##### 3) Reward Structure:

$$Reward(state) = hasLanded(state) - 100 * (d_t - d_{t-1}) - 100 * (v_t - v_{t-1}) - 100 * (\omega_t - \omega_{t-1})$$

This reward function favors soft, controlled touchdowns within the landing zone. This is done using:

- **Proximity Bonus:** It rewards the agent for being near to the landing pad when it lands.
- **Velocity:** It imposes a penalty on fast touchdown speed hence discouraging crashing landings.
- **Stability Encouragement:** Weakening angular velocity prevents the tilting or spinning of the agent after landing which can unbalance it.
- **Docking Penalty:** The reward model applies negative reinforcement in case the agent bounces back or takes off again after touching down to ensure a safe mission completion by preventing such behavior.

In this way, this combination of rewards guides an agent to choose actions leading into a safe and controlled landing.

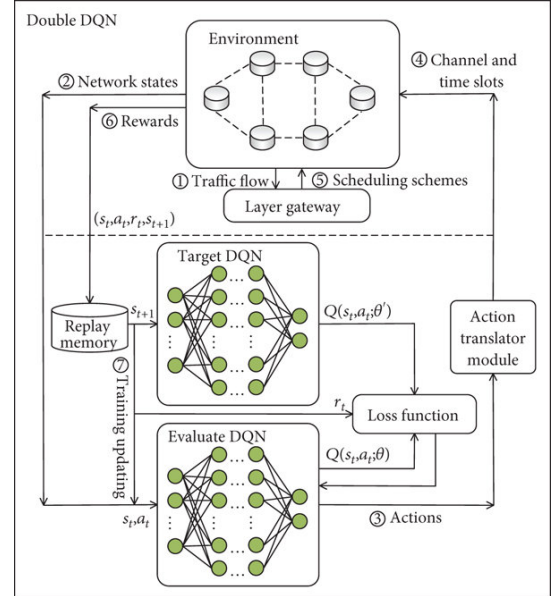


Fig. 3: An Illustration of DDQN Architecture. Courtesy: Research Gate

#### B. Double Deep Q-Network

Deep Reinforcement Learning (DRL) has a modern type of neural network architecture called the Double Deep Q-Network (DDQN) that attempts to mitigate the over-estimation bias present in traditional Deep Q-networks. This kind of bias occurs when one makes use of a single neural net for both selecting actions and evaluating Q-values.

The DDQN uses two separate neural networks, each with an equally configured Multi-Layer Perceptron (MLP), which is comprised of several fully connected hidden layers that have a non-linear activation function like Rectified Linear Unit (ReLU).

- Actor Network ( $\theta$ ): Calculates the q-values (action-values) for each available action 'a' in a given state 's' that represent the expected future reward for that action. For a given state s, the output of the actor network is

$$Q(s, a; \theta)$$

- Target Network ( $\tau$ ): It evaluates the Q-values estimated by  $\theta$  and thus decouples them from DQNs overestimation bias. The target network parameters are periodically synchronized with the actor network parameters at a slower rate using this update rule:  $\tau = \rho * \theta + (1 - \rho) * \tau$  (where  $0 < \rho \leq 1$ ). The update rule for Q-Learning is as follows:  $Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
- Action Selection ( $\epsilon$  - greedy Policy [5]): The actor network ( $\theta$ ) takes the state s as input and outputs Q-values for all possible actions ( $a \in A$ ). The agent employs an  $\epsilon$ -greedy policy for action selection. With probability  $1 - \epsilon$  (exploitation), the agent selects the action with the highest estimated Q-value:  $a^* = \operatorname{argmax}_a Q(s, a; \theta)$ . With probability ( $\epsilon$ ) (exploration), random action is chosen to encourage exploration of the environment.
- Polyak Update Rate ( $\rho$ ): This value ranges between 0 and 1 and it allows us to balance between stability (high  $\rho$ ) and learning speed (low  $\rho$ ). Usually found somewhere in between 0.95 and 0.99 standard values.
- Action Execution and Reward: The chosen action ( $a^*$ ) is applied to the lunar lander. After that, there would be a new state ( $s'$ ) and reward ( $r$ ) which comes from the result of undertaken action like; a positive reward for a successful landing and a negative reward for crashing.
- Experience Replay: The experience tuple ( $s, a^*, r, s'$ ) is stored in a replay buffer. This helps in improving sample efficiency and reducing correlations during training.
- Target Network Update (Periodic): Periodically updating the target network parameters ( $\tau$ ) with actor network parameters ( $\theta$ ) by using the polyak update rule mentioned above would therefore be considered.
- Loss Calculation and Backpropagation: Mini-batches of experience tuples are sampled from the replay buffer for loss calculation and backpropagation. Use Q-value estimated by target network for next state  $s'$  with selected action  $a^*$  as target value y. The loss function (mean squared error) is calculated between y(target value) and Q-value estimated by the actor network for all possible actions(a) on the current state(s).  $\text{Loss} = E[(y - Q(s, a; \theta))^2]$ . The loss (expected value of the square error between target and predicted values) is then backpropagated through the actor network ( $\theta$ ) to update its weights and improve its Q-value estimation accuracy.

The DDQN agent learns an optimal control policy for the lunar

lander by iteratively performing these steps, which allows it to make successful landings in a simulated environment without using much fuel.

### C. Multi-Layer Perceptron Network Structure

The mathematical expression of the MLP architecture [6] is as follows:  $Q(s, a) = W_o * \sigma(W_h * \sigma(W_i * s + b_i)) + b_o$  where:

- $Q(s, a)$ : Prediction of the Q-value for state s and action a.
- $W_i$ : Matrix of weight for the input layer.
- $W_h$ : Matrix of weight for the hidden layer(s).
- $W_o$ : Matrix of weights for the output layer.
- $b_i$ : Bias vector for the input layer.
- $b_o$ : Bias vector for the output layer.
- $\sigma$ : Activation function (ReLU).
- s: State vector which represents lunar lander environment, for example, position, velocity, angles.
- a: Action to be taken by the agent (e.g., thrust left, thrust right, turn on/off main engine).

This network consists of the following layers:

- Input Layer:
  - Size: d (depending on the state dimensionality of the lunar lander environment.  $d = 8$  for position, velocity, angles).
  - Function: Receives the state vector 's' as input.
- Hidden Layers (2):
  - Number of Neurons: h (pre-defined.  $h = 64, 128, 256$  or 512 neurons per layer).
  - Activation Function: ReLU (applied element-wise)
  - Function: Introduce non-linearity to capture complex relationships between states and actions.
- Output Layer:
  - Size: k (number of available actions for the lunar lander.  $k = 4$  for thrust left/right and turn main engine on/off).
  - Function: Outputs the predicted Q-value  $Q(s, a)$  for each action a in the state s.

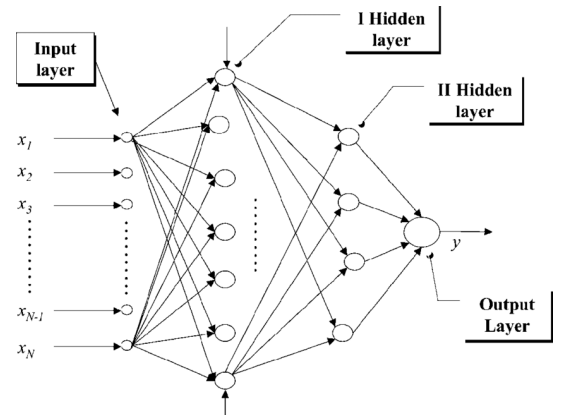


Fig. 4: Multi-Layer Perceptron Network Structure. Courtesy: Research Gate

#### D. Hyperparameter Tuning

- Learning Rate (LR):
  - Definition: Learning rate refers to the value that decides how much the weight ought to be updated based on the difference between predicted and actual Q values.
  - Parameter value: To start with DoubleQAgent, lr is initialized as 0.001.
- Discount Factor (gamma,  $\gamma$ ):
  - Definition: The Discount factor plays an important role in how much future rewards are weighted when calculating an expected return (Q-value) for an action.
  - Parameter Value: For DoubleQAgent initialization, this was set to gamma = 0.99.
  - Impact: A high discount factor (nearer to 1) emphasizes that it values more its future rewards than what has been earned in the past. Hence, this is why it makes a lot of decisions aimed at long-run profit maximization. Conversely, a low discount factor lays much emphasis on the immediate rewards hence making an agent concentrate so much on short-run benefits. The selection of gamma relies upon the problem domain and reward horizon.
- Exploration Rate (epsilon,  $\epsilon$ ):
  - Definition: The exploration rate is a hyperparameter that controls the agent's balance between exploration (trying new actions) and exploitation (utilizing learned knowledge). With a higher exploration rate, the agent is more likely to choose random actions, encouraging it to discover new and potentially better strategies. As the agent learns, the exploration rate is gradually reduced, favoring the exploitation of the knowledge gained through experience.
  - Parameter Value: Set to epsilon = 1.0 (initial value) and epsilon\_end = 0.01 (minimum value) during DoubleQAgent initialization. The decay rate is set to epsilon\_dec = 0.995.
- Batch Size:
  - Definition: In each training update, the number of transitions (state, action, reward, next state, termination) drawn from the agent's replay buffer is called batch size.
  - Parameter Value: During DoubleQAgent initialization set batch\_size = 128.
  - Impact: A bigger batch size can result in training with more stable gradient estimates but it costs a lot of computational resources. Conversely, a smaller batch size may be less computationally expensive but it can create noisier gradients and take a long time to converge.
- Memory Size (mem\_size):
  - Definition: The maximum number of transitions that can be stored in its replay buffer is determined by the memory size. This buffer contains past situations that help the agent learn a variety of situations during

training.

- Parameter Value: Set to mem\_size = 200000 during DoubleQAgent initialization.

- Double DQN Specific Parameters:

- Target Network (q\_func\_target): This generates a separate neural network known as q func target which is introduced alongside the main Q-network(q func) in DDQN. This distinction plays an important role in mitigating maximization bias. Although during training, action selection is based on predictions from the main Q-network (online network), when calculating the learning target (target Q-value) for a given state action pair, the value from the target network is used.
- Replace Target Parameter (replace\_q\_target): replace\_q\_target parameter to 100 must be set during DoubleQAgent initialization. That is, after every so many steps of training new weights are set on target network using those from the main network.

---

#### 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  $\epsilon$  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. 5: Deep Q-Learning Algorithm. Courtesy: Louis Wang

#### E. Implementation

This section will talk about the implementation details and training of Double DQN (DDQN). The problems encountered and their solutions are highlighted.

1) *Initial Implementation Challenges with TensorFlow Keras (tfkeras)*: The first version of the Double DQN agent used TensorFlow Keras (tfkeras) for constructing neural networks. This approach, however, proved non-performing as expected.

- Slow Training: Due to slow convergence during training, the learning rate was slow.
- Memory Exhaustion: On each training episode's completion, memory usage kept rising until it threw out-of-memory exceptions that necessitated saving NN weights and replay buffer state on disk and reloading them after crashes for resumption from the last checkpoint. Functionally it works but it is far too sluggish and error prone.
- Limited Scalability: As a result of memory constraints in the tfkeras based implementation, agents have been limited to simple environments or have shorter durations of training.



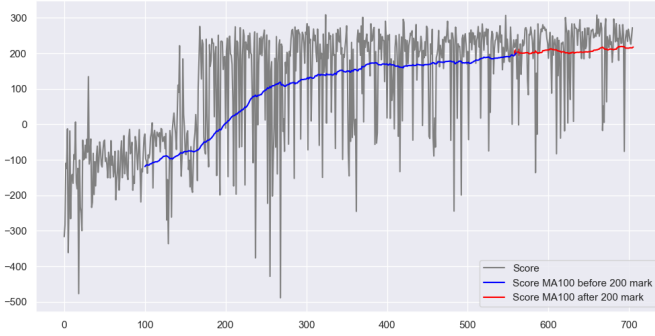


Fig. 6: Tensorflow-Keras Performance. Courtesy: Original

## 2) Transition to PyTorch and Performance Gains:

- The implementation was moved to a deep-learning framework called Pytorch to solve the performance problems experienced with tfkeras which is not efficient enough and may not be advantageous on GPU hardware. This transition was able to make a significant difference.
- Performance Increase: Pytorch demonstrated a great 100 times improvement than tfkeras. This helped reduce the time of training to further facilitate the development and testing of models quickly.
- Improved Scalability: The memory efficiency of PyTorch in addition to being GPU-friendly, allowed the agent to train for longer without being constrained by memory resources, which made it possible to handle more complex environments.
- Streamlined Training: Pytorch now requires neither manual checkpointing nor error recovery (saving/reloading weights and buffer state) partly because of its strong memory handling.

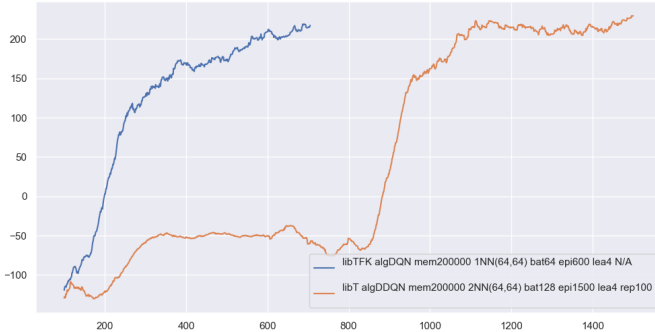


Fig. 7: Tensorflow-Keras vs PyTorch Performance. Courtesy: Original

3) *DDQN Architecture in PyTorch:* The  $q$  function of the target network and that of the main Q-network are implemented using a multi-layer perceptron (MLP) architecture whose final version is shown below.

MLP Configuration:

- Input Layer: Set the size of the input layer to correspond to the number of features in the Lunar Lander state vector, which is 8 dimensions.
- Hidden Layers: There are two hidden layers with a variable number of neurons per layer such as 64, 128 or 256. Used ReLU Activation function.
- Output Layer: This output layer has a size equal to the total number of actions that can be taken within an

environment (4 for Lunar Lander). On this layer each neuron represents a Q-value for one action in each state.

4) *Optimizer: Adam for Efficient Weight Updates:* Adam for Efficient Weight Updates: During training, Adam optimizer was used which is a Stochastic Gradient Descent(SGD) variant that adapts learning rates in real time. FGSD has some shortcomings compared to Adam optimization:

- Efficiency: Its learning rate is flexible enough because it adapts it separately for each parameter thus allowing faster convergence than typical SGD.
- Reduced Noise: It includes adaptive learning rates and momentum which result into less noisy gradients hence better convergence stability.

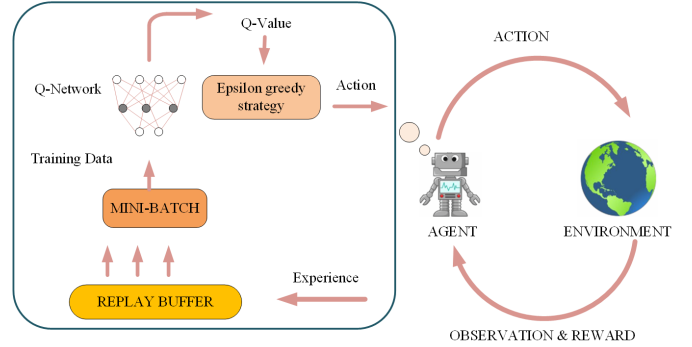


Fig. 8: Deep Q-Learning Process Flow. Courtesy: sokipriala jonah

5) *Training Process with Experience Replay:* The agent harnesses experience replay by saving prior experiences (transitions) in a replay buffer (MemoryBuffer class) that has the size `mem_size` determined during agent initialization. An assortment of experiences can be learned by the agent from this buffer over training. The experience replay approach used here is rudimentary. Prioritized experience replay (PER) could be considered for further improvement.

- Loss Function: In the course of learning, training employs the mean squared error (MSE) loss function which is computed within the learn method of DoubleQAgent class. It measures how far apart the predicted Q-value by `qfunc` for a given state-action pair and target Q-value obtained from `q_func_target` network differ in terms of squares. The latter difference acts as an error signal for the optimization of weights by the Adam optimizer in the main network update stage. Loss is calculated using this equation:

$$L_i(\theta_i) = E_{(s,a) \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$$

where

$$y_i = E_{s' \sim \epsilon} [r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})]$$

- Training Loop: The `train_agent` function implements the training loop, which goes through these steps:
  - State Collection: The environment state is gotten via `env.reset()` or taken from the previous step.
  - Action Selection (epsilon-greedy): The agent chooses an action from the current state according to a predefined greedy policy with some randomization.
  - Reward Evaluation and Transition Storage: Performing the chosen action in the environment

using `env.step()` gives back a reward, next state and termination information. This transition (current state, action, reward, next state, termination) is remembered by the agent.`save(state, action, reward, new_state, terminated)` and stored in the replay buffer.

- Experience Replay and Training Update: Define `LEARN EVERY` value as how it learns for all values. For every multiple of `LEARN EVERY` on steps that have passed since the last update mini-batch of transitions from the replay buffer is sampled. The learn method considers MSE loss and then propagates it through the main network. To minimize MSE loss between its target Q-values predicted by itself and target Q-values from target networks Adam optimizer modifies main network weights.

**Double DQN Specifics: Target Network Updates:** The `DoubleQAgent` class incorporates the core aspects of Double DQN:

- Target Network: Another network called ‘`q_func_target`’ is maintained by this algorithm. Weights of this network are synchronized with those of the main network periodically to avoid maximization bias.
- Target Network Update Frequency: The frequency of updating the target network with the main network’s weights is defined by the ‘`replace_q_target`’ parameter during agent initialization.

#### IV. THE RESULTS AND DISCUSSION

The initial development of the Double DQN agent employed TensorFlow Keras for the construction of a neural network. Nevertheless, this approach had significant performance bottlenecks. Therefore, strategic migration to PyTorch was done which reduced training time considerably. Over a wide range of hyperparameter configurations, this PyTorch model tends to exhibit consistently faster convergence. The speedup in the training process underscores the effectiveness of PyTorch, making it possible to quickly iterate on new models and perhaps tackle more complex learning tasks at an earlier stage.

##### A. Performance Evaluation: Deep Q-Learning Agents in Lunar Lander

This section evaluates how Deep Q-Learning (DQN) and Double Deep Q-Learning (DDQN) agents performed on the lunar lander Deep Reinforcement Learning task. Two important Key Performance Indicators (KPIs) are considered:

- Episodic Reward Convergence: This is calculated as the episode where the mean episodic reward over a fixed number of 100 episodes becomes greater than 200 points. It quantifies how good or bad the agent’s ability to learn within a reasonable period by successfully landing on the moon’s surface.
- Terminal State Performance: Assessed via video demonstrations of the terminal state performance in

which a trained agent is compared to an untrained one. This qualitative review provides a holistic insight into the control policy of the agent and its navigational effectiveness in the changing lunar lander setting.

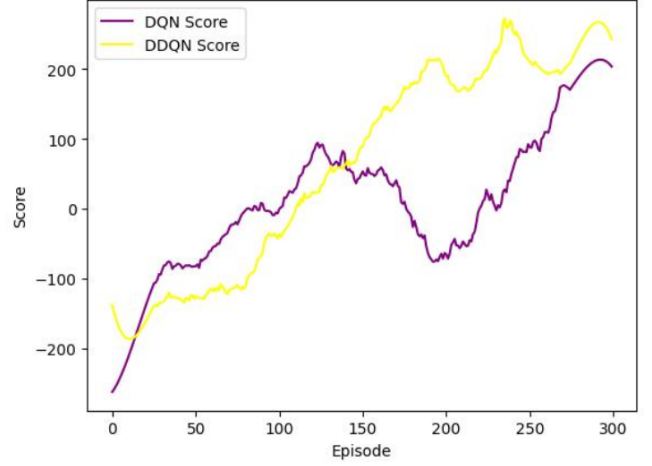


Fig. 9: DQN vs DDQN Performance. Courtesy: Original

##### B. DQN vs. DDQN: A Comparative Analysis

Both DQN and DDQN agents were implemented using the PyTorch deep learning framework. The results strongly support the DDQN approach as it has made it clear that it has a greater advantage over DQN. Compared to DQN, the DDQN agent achieved faster convergence to the target reward threshold concerning time. As shown in Figure 9, the best performing variant of DDQN ( $256 \times 256$  neurons) converged in episode 369, while that of DQN ( $64 \times 64$  neurons) converged in episode 461. This represents a savings of approximately 20% in training time for DDQNs before they can be said to reach stable performance. Such rapid convergence indicates that DDQN architecture effectively mitigates overestimation bias exhibited by DQNs thereby promoting better learning and policy improvement.

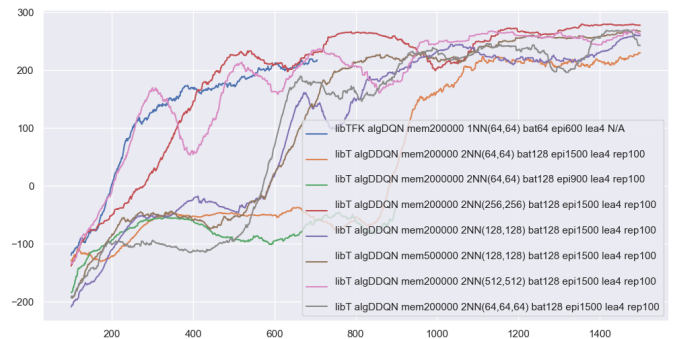


Fig. 10: Learning curves analysis. Courtesy: Original

##### C. Impact of Network Architecture on DDQN Performance

Some experiments were done to investigate how network architecture influenced the performance of DDQNs mainly focusing on the number of neurons in hidden layers as seen in Figure 10.



- **Impact of Hidden Layer Breadth:** The speed at which convergence happens is highly determined by the width of hidden layers (number of neurons). Networks with a wider hidden layer (256 neurons) converged faster than other configurations having narrower ones (64 neurons). This implies that a wider network has a greater capacity for learning intricate relationships between states and actions in the lunar lander environment, which enables the agent to develop a more nuanced understanding of control dynamics and hence formulate effective landing strategies.
- **Depth vs. Breadth:** However, increasing the depth (number of hidden layers) also makes a difference in addition to increasing the breadth of the hidden layer that impacts on convergence speed most. Networks having a single hidden layer (even with 256 neurons) had slower convergence than those having two hidden ones (both 64 and 256 neurons). It indicates that some level of depth is beneficial for learning complex control policies. Nevertheless, the thickness has less impact than width showing that broader architecture allows more efficient feature extraction and representation learning.
- **Computational Complexity vs. Performance Trade-off:** It is important to understand that there is a tradeoff between network complexity and performance. Networks with an excessive number of neurons, 512 neurons per layer, can converge slightly faster than the best setup with 256 neurons per layer. However, they also have higher training computational costs. As such, these results suggest that the optimal trade-off between convergence speed and computational efficiency is reached when the network is configured with 256 neurons per layer to facilitate efficient learning without posing a significant strain on training resources.

Index	Learning rate	Discount factor	Average reward	Completion time(s)
1	0.0005	0.99	234.88	450
2	0.001	0.99	243.36	410
3	0.0005	0.975	204.08	510
4	0.001	0.975	112.42	498
5	0.0005	0.95	-46.69	715
6	0.001	0.95	-63.35	727

TABLE II: The result of DDQN

#### D. Optimal Hyperparameter Configuration

It was found from these results that, in terms of convergence speed and suitability for efficient training, the DDQN agent with the following settings showed the best performance:

- **Network Architecture:** Two hidden layers each having 256 neurons using ReLU activation. This permits a balance between model capacity and computational efficiency for enabling agents to learn complicated control policies without requiring too many training resources.
- **Experience Replay Memory Size:** 200000 transitions. Since most historical experiences can be stored within this memory size, it allows an agent to generalize

on various state-action sequences hence fostering exploration.

- **Batch Size:** 128. Updating models fast but still maintaining some randomness during learning requires such a batch size which will cause sample variance but can be efficient during training.
- **Training Episodes:** 1500 episodes (converging at episode 369). Training time is sufficient enough for the agent to reach good control policies within reasonable time limits.

By doing so, this configuration effectively balances learning capacity and computational demands thereby leading to successful training of the DDQN agent.

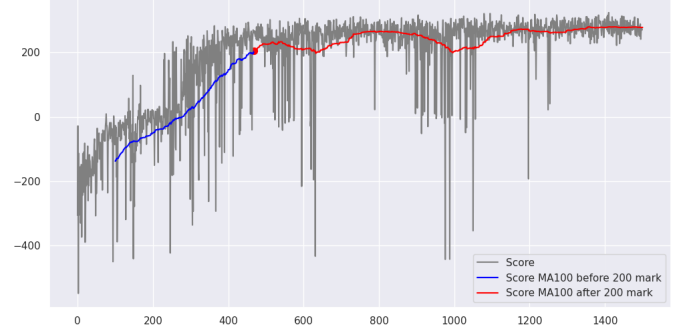


Fig. 11: Best model performance. Courtesy: Original

#### E. Evaluation of Trained Agent

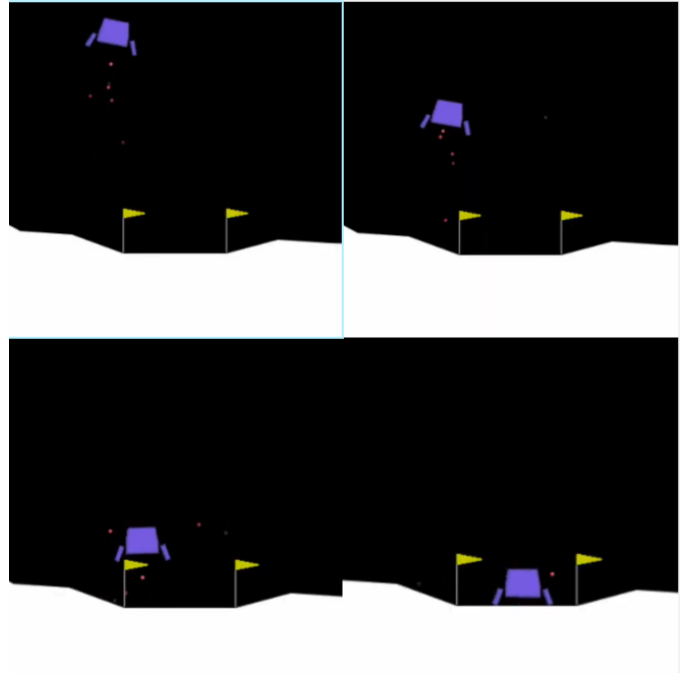


Fig. 12: Flight simulation stages. Courtesy: Original

Videos show that the trained agent is better at landing a rocket on the moon than an untrained one.

- Video of the best-performing agent landing the rocket on the moon (experiment M5)
- Video of an untrained agent landing the rocket on the moon

Through watching these videos, it can be confirmed that a stable and controlled landing has been achieved by this learned

agent.

The findings support DDQN's suitability as an approach to lunar lander problem. The DDQN was also shown to converge much faster than DQN, pointing out its benefit toward bias in estimation. Additionally, experiments demonstrated how convergence speed depends on network architecture, with two hidden layers along with 256 neurons configuration achieving better performance and computational efficiency tradeoffs.

## V. CONCLUSION

This research has explored the Double Deep Q-Learning (DDQN) for accurate control of the lunar lander in the OpenAI Gym Lunar Lander v2 environment. DDQN models are implemented using TensorFlow Keras (tfkeras) and PyTorch frameworks and their performances were compared.

### A. Technical Details

DDQN Architecture: The best performing model employed two hidden layers, each of which had 256 neurons with a ReLU activation function.

Optimal Hyper-parameter Configuration:

- $\gamma = 0.99$ ,
- $\epsilon = 1.0$ ,
- $\epsilon_{dec} = 0.995$ ,
- learning rate (lr) = 0.001,
- mem\_size = 200000,
- batch\_size = 128,
- $\epsilon_{end} = 0.01$

### B. Key Findings and Future Work

The superiority of PyTorch: While the tfkeras model converged in 461 episodes, the Pytorch's model took approximately 369 episodes to converge. This suggests that the PyTorch model is converging faster than its counterpart which achieves convergence in about 369 episodes as opposed to around 461, thereby implying that tfkeras is slower than pytorch. These findings indicate that PyTorch could be useful for developing effective DDQN agents with optimal hyperparameters (256 neurons per layer, 200,000 memory size, 128 batch size). Further research can be done on this topic building on the success of a simple implementation such as Pytorch-based DDQN. Some examples of simpler TD methods (like SARSA and Dyna-Q) might provide hints as to whether deep learning has a better compromise over simpler algorithms for lunar landing. Furthermore, better settings may be found through a more exhaustive search of hyperparameters and trying out some network architectures like CNNs or dueling networks. The last is the combination of uncertainties [7] (exploration-exploitation, model, sensor noise).

## ACKNOWLEDGMENT

It is with a lot of gratitude that I express my appreciation for those whose priceless support made this B.Tech research internship a success. At the outset, I wish to extend my sincere thankfulness to my esteemed supervisor, Dr. Mainak Adhikari for their unparalleled guidance, support and motivation during

the whole period of the internship. His ideas and knowledge played an important role in shaping the course taken by the study. Thanks to the Indian Institute of Information Technology Lucknow (IIIT Lucknow) which has enabled me to get knowledge and skills that will be useful in my future career.

Furthermore, I would like to acknowledge the open-source community as well as the powerful frameworks that played vital roles in this project. PyTorch, TensorFlow and OpenAI Gym were heavily used in developing and putting into place the most important functionalities of the research.

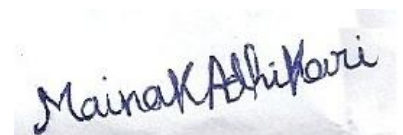
## REFERENCES

- [1] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction. MIT press, 2018.
- [2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba, "Openai gym," 2016.
- [3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through Deep Reinforcement Learning," Nature, vol. 518, no. 7540, p. 529, 2015.
- [4] Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. Advances in neural information processing systems, 12, 1057-103, 1999.
- [5] E. Rodrigues Gomes and R. Kowalczyk, "Dynamic analysis of multiagent q-learning with  $\epsilon$ -greedy exploration," in Proceedings of the 26th Annual International Conference on Machine Learning, ser. ICML '09. New York, NY, USA: ACM, 2009, pp. 369–376. [Online]. Available: <http://doi.acm.org/10.1145/1553374.1553422>.
- [6] Marius-Constantin Popescu, Valentina Emilia Balas, Aurel Vlaicu, Liliana Perescu-Popescu, Nikos E Mastorakis, Multilayer perceptron and neural networks, 2009.
- [7] Gadgil, S., Xin, Y., & Xu, C. Solving the lunar lander problem under uncertainty using reinforcement learning. In 2020 SoutheastCon, 2, 1-8 2020.

Sign here



(Meet Savla)



(Dr. Mainak Adhikari)