Solving Problems Using Reinforcement Learning

Saul Vassallo
saul.vassallo.22@um.edu.mt
University of Malta

Abstract

Reinforcement Learning (RL) has proven to be a powerful approach for solving sequential decision-making problems. In this study, we apply RL techniques to control problems using the OpenAI Gym LunarLander environment, addressing both discrete and continuous action spaces. The first experiment implements a Deep Q-Network (DQN) to solve the discrete action LunarLander-v3 task. We compare a standard DQN with an improved variant, a Noisy DQN. Additionally, we analyze the impact of state noise levels (HIGH - 0.01, MEDIUM - 0.05, LOW - 0.1) on model robustness, assessing how performance degrades with increasing uncertainty.

In the second experiment, we address LunarLanderContinuous-v3, a continuous control task, using the Deep Deterministic Policy Gradient (DDPG) algorithm. Hyperparameter tuning is performed to optimize policy learning, and the role of exploration noise is analyzed across different levels. Performance is evaluated using reward trends over training episodes, comparing convergence rates and stability between different models and noise conditions.

Keywords: Reinforcement Learning (RL), Machine Learning (ML), Deep Q-Network (DQN), Deep Deterministic Policy Gradient (DDPG)

1 Introduction

Reinforcement Learning (RL) is a subfield of machine learning where an agent learns an optimal policy by interacting with an environment to maximize cumulative rewards [1, 2]. Unlike supervised learning, which requires labeled data, RL relies on trial-and-error learning, where agents explore actions, receive feedback in the form of rewards, and refine their policy over time. The RL framework is formally defined as a *Markov Decision Process (MDP)*, consisting of:

- A set of states S
- A set of actions A
- A transition function $P(s'|s, a)$ defining the probability of moving to state s' after taking action a in state s
- A reward function $R(s, a)$ specifying the reward for taking action a in state s
- A discount factor γ that determines the weight of future rewards

RL is widely used in domains such as robotics, finance, and autonomous systems, where learning optimal strategies through interaction is essential [3].

1.1 Comparison with Other Machine Learning Approaches

Reinforcement Learning differs fundamentally from traditional machine learning paradigms:

1. **Supervised Learning vs. RL**: Supervised learning relies on labeled datasets, whereas RL learns through interaction and delayed rewards. This makes RL particularly suitable for *sequential decision-making* problems where optimal solutions depend on long-term outcomes [3].
2. **Unsupervised Learning vs. RL**: Unsupervised learning focuses on pattern discovery (e.g., clustering), whereas RL optimizes a policy to maximize rewards. RL involves active decision-making rather than passive pattern recognition.
3. **Exploration-Exploitation Trade-off**: RL agents must balance *exploration* (trying new actions to gain more knowledge) and *exploitation* (using current knowledge to maximize rewards). This trade-off does not exist in standard supervised learning [3].

1.2 Categories of RL Methods

RL algorithms are broadly classified into:

- **Value-Based Methods**: These methods estimate the expected cumulative reward for each state-action pair using a *Q-function*. Example: *Deep Q-Networks (DQN)*, which extend Q-learning with deep learning for high-dimensional state spaces [4–6].
- **Policy-Based Methods**: Instead of learning a value function, these methods directly optimize the policy to select actions. Example: *Policy Gradient Methods (e.g., REINFORCE)* [9, 10].
- **Actor-Critic Methods**: A hybrid approach that combines both value-based and policy-based learning. Example: *Advantage Actor-Critic (A2C) and Proximal Policy Optimization (PPO)* [11].

Each approach has trade-offs: Value-based methods are more sample-efficient but struggle with continuous action spaces, while policy-based methods work well in high-dimensional spaces but have high variance. Actor-Critic methods aim to balance these strengths by leveraging value estimates for policy improvement.

1.3 Paper Contribution

This paper focuses on applying RL techniques to control problems in the OpenAI Gym LunarLander environment:

- **Experiment 1**: Solve LunarLander-v3 using DQN and an improved variant. Analyze the impact of noise on performance.
- Experiment 2: Solve LunarLanderContinuous-v3 using DDPG. Study the role of exploration noise in continuous control.

Through these experiments, we evaluate the robustness of RL models under different noise conditions and discuss the implications for real-world control applications.

2 Background

This section provides a comprehensive overview of fundamental reinforcement learning (RL) concepts, focusing on value-based and policy-based methodologies. Additionally, we introduce Deep Q-Networks (DQN), its Noisy variant, and the Deep Deterministic Policy Gradient (DDPG) algorithm, all of which are central to our experimental framework.

2.1 Value-Based Methods

Value-based reinforcement learning methods utilize a value function to evaluate and select optimal actions within an environment. Two principal functions govern these approaches: the state-value function, denoted as $V(s)$, which represents the expected cumulative reward when starting from state s and following a policy π, and the action-value function, $Q(s, a)$, which estimates the expected cumulative reward for executing action a in state s and subsequently adhering to policy π.

These functions satisfy the Bellman equation, a recursive relation that defines the value of a state in terms of expected future rewards:

$$Q(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q(s', a')|s, a], \tag{1}$$

where r represents the immediate reward, s' is the next state, and γ is the discount factor [3]. This equation serves as the foundation for algorithms such as Q-learning, which iteratively refines the Q-value estimates to achieve an optimal policy.

Value-based methods have proven highly effective in discrete-action environments, but they exhibit limitations when extended to continuous spaces. The curse of dimensionality further impacts their ability to scale to complex problems, making function approximation techniques such as deep neural networks essential in modern reinforcement learning applications.

2.2 Deep Q-Networks (DQN)

Deep Q-Networks (DQN) extend the traditional Q-learning algorithm by incorporating deep neural networks to approximate the Q-function, enabling RL agents to handle high-dimensional state spaces. The stability of DQN training is enhanced through two key mechanisms: experience replay, which stores past experiences in a buffer to break correlations between consecutive samples, and target networks, which maintain a separate Q-network for computing target values, thereby reducing training instability.

The update rule for DQN follows:

$$\theta \leftarrow \theta + \alpha\Big(y - Q(s, a; \theta)\Big)\nabla_\theta Q(s, a; \theta), \tag{2}$$

where the target value y is computed as:

$$y = r + \gamma \max_{a'} Q(s', a'). \tag{3}$$

In this context, θ represents the parameters of the deep neural network used to approximate the Q-function. These parameters define the weights and biases of the network, which are optimized during training. The gradient term $\nabla_\theta Q(s, a; \theta)$ denotes the partial derivative of the Q-value function with respect to the network parameters, ensuring that updates are directed toward minimizing the temporal-difference (TD) error. The learning rate α controls the magnitude of these updates, balancing between convergence speed and stability.

The update rule was implemented in python as follows:

```
1 optimizer.zero_grad()
2 loss = loss_fn(qvalues, expected_qvalues)
3 loss.backward()
4 optimizer.step()
```

Here, line 1 calculates the temporal difference error, corresponding to:

$$y - Q(s, a; \theta) \tag{4}$$

Where y is the target Q-value computed using the Bellman equation. Line 2 computes the gradient of the loss function with respect to the network parameters:

$$\nabla_\theta Q(s, a; \theta) \tag{5}$$

ensuring that updates are directed towards minimizing the loss and finally line 3 implements the gradient update step using the computed gradients:

$$\theta \leftarrow \theta + \alpha\nabla_\theta Q(s, a; \theta) \tag{6}$$

where α is the learning rate.

By leveraging deep learning techniques, DQN has demonstrated success in complex control tasks, particularly in environments where state representations are intricate and high-dimensional [5, 6].

While DQN provides significant advancements, it suffers from instability and inefficiency in continuous or highly stochastic environments. Several improvements, such as Noisy DQN and prioritized experience replay, have been introduced to mitigate these issues.

2.3 Noisy DQN: Experiment 1 Enhancement

Standard DQN employs ϵ-greedy exploration, which may lead to suboptimal exploration policies due to its reliance on a decaying random action probability. Noisy DQN addresses

this limitation by integrating learnable noise into the network weights, fostering state-dependent exploration. The network parameters are updated as:

$$W = W_\mu + W_\sigma \cdot \epsilon, \tag{7}$$

where W_μ and W_σ are learnable parameters, and ϵ is sampled from a noise distribution [6].

This was implemented in code as follows:

```
class NoisyLinear(nn.Module):
    def __init__(self, in_size, out_size):
        super(NoisyLinear, self).__init__()

        self.w_mu = nn.Parameter(torch.empty
            ((out_size, in_size)))
        self.w_sigma = nn.Parameter(torch.
            empty((out_size, in_size)))
        self.b_mu = nn.Parameter(torch.empty
            ((out_size)))
        self.b_sigma = nn.Parameter(torch.
            empty((out_size)))

        # Initialize parameters
        uniform_(self.w_mu, -math.sqrt(3 /
            in_size), math.sqrt(3 / in_size)
            )
        uniform_(self.b_mu, -math.sqrt(3 /
            in_size), math.sqrt(3 / in_size)
            )
        nn.init.constant_(self.w_sigma,
            0.017)  # Fixed initialization
            for stability
        nn.init.constant_(self.b_sigma,
            0.017)
```

The corresponding *forward* function implements the noise-injected transformation:

```
def forward(self, x, sigma=1):
    if self.training:
        w_noise = torch.normal(0, sigma,
            size=self.w_mu.size()).to(x.
            device)
        b_noise = torch.normal(0, sigma,
            size=self.b_mu.size()).to(x.
            device)
        return F.linear(
            x,
            self.w_mu + self.w_sigma *
                w_noise,
            self.b_mu + self.b_sigma *
                b_noise,
        )
    else:
        return F.linear(x, self.w_mu, self.
            b_mu)
```

This implementation defines the Noisy Linear Layer, where the weights are parameterized by W_μ and W_σ. The trainable noise parameters replace traditional fixed-weight matrices, allowing for dynamic exploration strategies based on the training progress. The `forward` function modifies the standard linear transformation by injecting noise into both weights and biases, ensuring adaptive exploration by adjusting the randomness in action selection during training. Specifically:

- **Line 7** implements $W = W_\mu + W_\sigma \cdot \epsilon$ by adding scaled noise to the learned parameters.
- **Line 8** applies noise to biases, following the same principle.
- **if self.training** Ensures that noise is only applied during training, while evaluation uses deterministic parameters.

This method dynamically adjusts exploration throughout training, improving stability and overall learning efficiency.

Recent studies have demonstrated that Noisy DQN significantly enhances exploration efficiency compared to traditional exploration strategies, particularly in environments with sparse rewards. By allowing the model to adapt exploration rates dynamically, Noisy DQN avoids premature convergence to suboptimal policies and fosters more robust learning trajectories [7, 8].

2.4 Policy-Based Methods

Policy-based methods directly optimize the policy π without relying on value function estimation. These methods are underpinned by the Policy Gradient Theorem:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a) \right], \tag{8}$$

where $J(\theta)$ represents the objective function, and π_θ is the parameterized policy [10]. Policy-based approaches, such as REINFORCE and Trust Region Policy Optimization (TRPO), are particularly effective in continuous action spaces where value function discretization may be infeasible.

Despite their advantages, policy gradient methods suffer from high variance, requiring advanced techniques such as baseline functions and entropy regularization to improve convergence rates and learning stability.

2.5 Actor-Critic Methods

Actor-Critic methods combine value-based and policy-based paradigms, leveraging a dual-network architecture. The *actor* updates policy parameters based on the policy gradient, while the *critic* evaluates state-action pairs using a value function. By reducing variance in policy updates, Actor-Critic methods enhance stability and convergence speed in RL applications [11].

This framework has led to the development of more sophisticated algorithms such as Advantage Actor-Critic (A2C)

and Proximal Policy Optimization (PPO), which have demonstrated superior performance across a wide range of continuous and discrete RL tasks.

2.6 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) extends DQN to continuous action spaces through an Actor-Critic framework. The actor network selects actions, while the critic network evaluates them using a learned Q-function. DDPG incorporates:

- **Target Networks**, which mitigate instability by maintaining separate networks for action selection and evaluation.
- **Ornstein-Uhlenbeck Noise**, which introduces temporally correlated noise for exploration in continuous domains.

The policy update follows:

$$\nabla_{\theta^\mu} J = \mathbb{E}\left[\nabla_a Q(s, a|\theta^Q) \nabla_{\theta^\mu} \mu(s|\theta^\mu)\right], \tag{9}$$

where θ^μ represents the actor network parameters [12].

The corresponding equation was implemented in code as follows:

```
actor_loss = -critic(states, actor(states)).
    mean()
actor_optimizer.zero_grad()
actor_loss.backward()
actor_optimizer.step()
```

This code performs the *actor network update*, where:

$$\mu(s|\theta^\mu) = \text{actor(states)} \tag{10}$$

$$Q(s, a|\theta^Q) = \text{critic(states, actor(states))} \tag{11}$$

$$\nabla_{\theta^\mu} J = \text{actor\_loss.backward()} \tag{12}$$

$$\theta^\mu \leftarrow \theta^\mu + \alpha \nabla_{\theta^\mu} J = \text{actor\_optimizer.step()} \tag{13}$$

DDPG has demonstrated superior performance in robotic control tasks and high-dimensional continuous environments by enabling fine-grained action control.

3 Methodology

This section describes the experimental setup, implementation details, and evaluation methodology adopted in this study.

3.1 Libraries Used

Several external libraries were utilized in this study to facilitate reinforcement learning experiments and model training. The `Gymnasium` library provides a standardized and reliable interface for reinforcement learning environments. `NumPy` was employed for efficient numerical computations, particularly for handling arrays and mathematical operations. The deep learning framework `PyTorch` was used to define neural network architectures, implement training loops, and perform tensor-based computations. The `Matplotlib` library was used for visualizing training progress and model performance. Additionally, `Google Colab's Drive API` was integrated to store and retrieve experimental data, ensuring reproducibility. The `json` library was utilized to serialize and save experimental results, while `warnings` was employed to suppress unnecessary output during training.

3.2 Evaluation Framework

For each configuration (hyperparameter tuning, DQN vs Noisy DQN comparison, and effect of noisy states on DQN, Noisy DQN and DDPG), the model was trained, and the reward trajectory and convergence episode were recorded. To ensure reproducibility, training results were stored in a structured format. If results were previously saved, they were loaded from a JSON file, avoiding redundant computations. The training was stopped at 1000 episodes for experiment 1 and 750 episodes for experiment 2, or once the average reward over the last 50 episodes was greater than or equal to 195. An episode limit was used due to computation limits on google colab.

Models were then evaluated by examining both the convergence episode (if converged) and the reward trajectory over episodes. It is important to note that during these experiments each individual test/run was only executed once. A recommended improvement would be to repeat the training process numerous times and evaluate the average convergence episode over these runs.

3.3 Noise Levels

The noise levels for the 2 experiments were set as follows:

- **LOW** = 0.01
- **MEDIUM** = 0.05
- **HIGH** = 0.1

3.4 Experiment 1: LunarLander-v3

3.4.1 Experimental Design. The experiment utilizes the LunarLander-v3 environment from OpenAI Gymnasium, which serves as a standard benchmark for reinforcement learning algorithms. The objective is to develop an agent capable of controlling a spacecraft to achieve a stable landing on a designated landing pad while minimizing fuel consumption and avoiding crashes. The environment comprises an eight-dimensional state space, encapsulating the position, velocity, angle, angular velocity of the spacecraft, and contact indicators for both landing legs. The agent operates within a discrete action space consisting of four possible actions: no thrust, firing the left engine, firing the right engine, or engaging the main engine. The reward function is structured to incentivize smooth landings while penalizing actions that lead to instability or excessive fuel expenditure [3, 13].

3.4.2 Algorithm Implementation. In this experiment both a simple DQN and a noisy DQN were implemented. The architecture of the DQN used in this experiment consists

of an input layer, two hidden layers with 128 neurons each, and an output layer corresponding to the number of possible actions. The selection of this architecture is justified based on efficiency, expressiveness, and stability.

The two hidden layers allow the network to capture hierarchical feature representations, a necessity for environments where raw state inputs require abstraction. A single hidden layer may not be sufficient to approximate complex Q-value functions accurately, while deeper networks often introduce instability in training due to vanishing gradients and unstable updates [14]. Empirical studies have demonstrated that two hidden layers strike a balance between expressiveness and computational feasibility, making them a common choice in reinforcement learning applications [4, 15].

Each hidden layer consists of 128 neurons, a choice driven by computational efficiency and the capacity required for function approximation. Networks with fewer neurons may struggle to generalize across diverse states, leading to suboptimal policies, whereas excessively large networks increase the risk of overfitting and unnecessary computational complexity [16].

ReLU (Rectified Linear Unit) activation is applied to both hidden layers:

$$\text{ReLU}(x) = \max(0, x) \tag{14}$$

ReLU is chosen due to its computational efficiency and ability to mitigate the vanishing gradient problem, ensuring better gradient flow and faster convergence [17]. Unlike sigmoid and tanh activations, which saturate for large input values, ReLU maintains activations without unnecessary bounds, making it highly effective for deep reinforcement learning architectures [18].

Overall, the chosen architecture optimally balances depth, neuron count, and activation functions to enhance stability and efficiency in Q-value estimation. This design has been validated in benchmark reinforcement learning environments, including Atari and continuous control tasks [4, 15].

The Noisy DQN algorithm was implemented as discussed in the background section maintaining the same architecture as the standard DQN. This was done so as to ensure that the DQN comparisons carried out later on were as fair as possible.

3.4.3 Hyperparameter Selection and Justification.

The experiment used the following key hyperparameters:

- **Discount Factor** ($\gamma = 0.99$): Ensures that long-term rewards are appropriately considered while preventing excessive reliance on immediate rewards [4].
- **Batch Size** ($B = 64$): Balances computational efficiency and training stability, as smaller batch sizes increase variance, while larger ones slow convergence [15].
- **Replay Buffer Size** (100000): Allows the agent to store past experiences for decorrelated training updates, enhancing sample efficiency.
- **Minimum Replay Size** (10000): Ensures sufficient initial experience before training begins to stabilize early learning dynamics.
- **Exploration Parameters**: *Epsilon Decay Strategy* ($\epsilon_{start} = 1.0$, $\epsilon_{end} = 0.01$, $\epsilon_{decay} = 0.995$) promotes gradual transition from exploration to exploitation, reducing randomness as learning progresses [7].
- **Learning Rate** ($\alpha = 0.0001$): A low learning rate ensures stable training and avoids large oscillations in Q-value updates [16].
- **Optimizer Choice**: *Adam optimizer* is used for standard DQN due to its adaptive learning rate properties, while *AdamW optimizer* is used for Noisy DQN, which provides better weight decay regularization and improves robustness in noisy training settings.
- **Target Update Frequency**: This was tuned in the next section.
- **Loss Functiont**: This was tuned in the next section.

These hyperparameter choices align with best practices in deep reinforcement learning and are validated through empirical studies demonstrating improved learning stability and efficiency [4, 15].

3.4.4 Hyperparameter Tuning.

Hyperparameter tuning is an essential aspect of reinforcement learning model optimization, as different configurations can significantly impact performance. In this study, we conducted a systematic exploration of key hyperparameters, focusing on the loss function and target network update frequency.

The loss function is a fundamental component in training reinforcement learning models, as it determines how the model learns from errors in Q-value estimation. Two loss functions were evaluated:

- **Mean Squared Error (MSE)**: A traditional loss function used in deep Q-learning, which penalizes large deviations between predicted and target Q-values.
- **Huber Loss (Smooth L1 Loss)**: A robust alternative that reduces sensitivity to outliers and mitigates instability in Q-value updates.

Additionally, we examined the effect of varying the target network update frequency, selecting values of {1, 10, 50, 100}. Frequent target updates (e.g., every step) can cause high variance in training, whereas infrequent updates may slow convergence.

The training process involved iterating through different combinations of loss functions and target update frequencies. The impact of these hyperparameters on model performance was analyzed following prior research recommendations [19].

3.5 Experiment 2: LunarLanderContinuous-v3

3.5.1 Experimental Design. The second experiment employs the `LunarLanderContinuous-v3` environment from OpenAI Gymnasium, which is a continuous control benchmark for reinforcement learning algorithms. The objective is to train an agent capable of autonomously landing a spacecraft on a designated pad while minimizing fuel consumption and avoiding crashes. The environment provides an eight-dimensional state space representing the lander's position, velocity, angle, angular velocity, and landing leg contact indicators. Unlike the discrete action space of `LunarLander-v3`, this environment features a continuous action space where the agent controls the main and lateral thrusters with continuous values between predefined limits.

3.5.2 Algorithm Implementation. The experiment utilizes the Deep Deterministic Policy Gradient (DDPG) algorithm, a model-free reinforcement learning method designed for continuous action spaces. The DDPG implementation in this experiment follows an architecture with two hidden layers of 256 neurons, using `ReLU` activations for non-linearity. The critic network processes both the state and action as input, concatenating them before passing through its fully connected layers. The selected architecture balances computational efficiency and the ability to approximate complex continuous functions.

3.5.3 Hyperparameter Selection and Justification. The experiment used the following key hyperparameters:

- **Discount Factor** ($\gamma = 0.99$): Encourages long-term reward optimization while preventing excessive discounting of future rewards.
- **Batch Size** ($B = 128$): A moderate batch size that provides stable training updates while maintaining computational efficiency.
- **Replay Buffer Size** (100000): Stores past transitions to enable decorrelated training samples, improving sample efficiency.
- **Minimum Replay Size** (10000): Ensures that the buffer has accumulated enough experience before updates begin, stabilizing early training dynamics.
- **Optimizer**: The `AdamW` optimizer was used for both actor and critic networks to leverage weight decay regularization, improving training robustness.

3.5.4 Hyperparameter Tuning. The experiment investigated the effect of two key hyperparameters:

- **Target Update Rate** (τ): Governs the soft update of the target networks. The tested values were {0.001, 0.005, 0.01} to evaluate the trade-off between stability and adaptability.
- **Noise Standard Deviation** (σ): Determines the scale of exploration noise. The tested values were {0.1, 0.2, 0.3}, balancing between excessive randomness and premature exploitation.

Each combination of τ and σ was evaluated based on training convergence and stability. The results were stored in a structured format, ensuring that previously completed experiments were not repeated. The combinations were evaluated as discussed in section 3.2

4 Results

4.1 Experiment 1

4.1.1 Hyperparameter Tuning. The hyperparameter tuning experiment involved testing different loss functions (Huber and MSE) and varying update frequencies (1, 10, 50, 100). The results are summarized in Table 1 and Figures 1 and 2.

Combination	Convergence Episode
HUBER_freq1	389
MSE_freq1	478
HUBER_freq10	492
MSE_freq10	686
HUBER_freq50	No Convergence
HUBER_freq100	No Convergence
MSE_freq50	No Convergence
MSE_freq100	No Convergence

Table 1. Hyperparameter tuning results sorted by convergence speed.

Figures 1 and 2 illustrate the moving average rewards for different hyperparameter combinations.

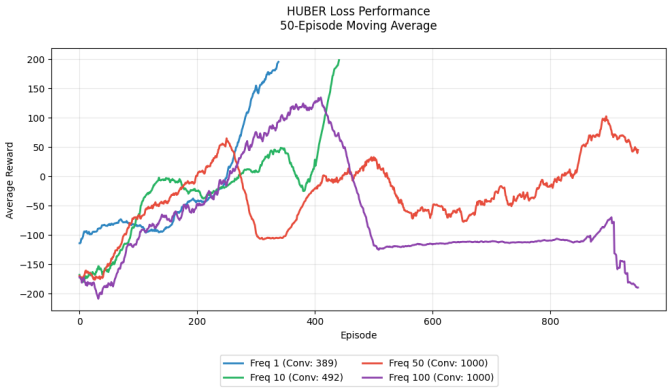

Figure 1. Performance of different Huber loss configurations.

4.1.2 DQN vs Noisy DQN. This experiment compared standard DQN and Noisy DQN under different noise levels. The results are summarized in Table 2 and illustrated in Figures 3.

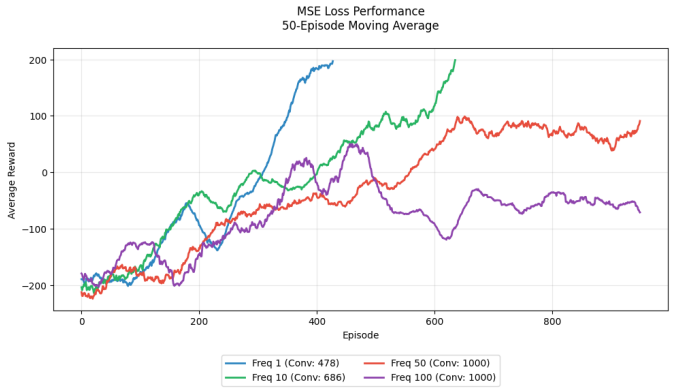

Figure 2. Performance of different MSE loss configurations.

DQN Variant	Convergence Episode
Standard DQN	511
Noisy DQN	583

Table 2. Comparison of standard DQN and Noisy DQN convergence.

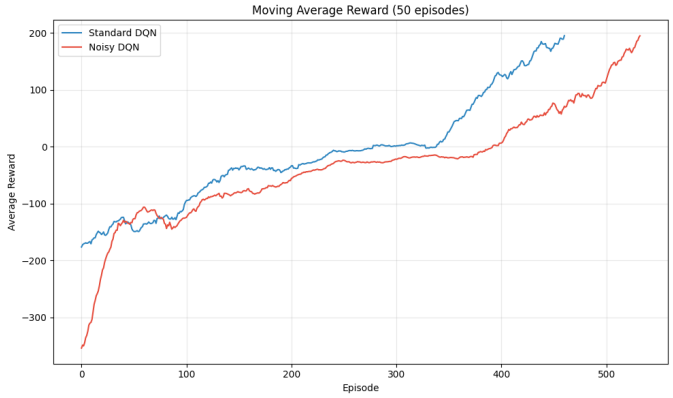

Figure 3. Comparison of standard DQN and Noisy DQN performance.

4.1.3 Noisy State in DQN and Noisy DQN.

The introduction of state noise had a significant impact, as seen in Table 3 and figures 4 5.

4.1.4 Discussion and Conclusions.

The experiments conducted provide insight into the impact of hyperparameter tuning and noise on the performance of DQN.

In the first experiment, Huber loss with an update frequency of 1 resulted in the fastest convergence (episode 389), demonstrating its effectiveness in stabilizing training. MSE loss generally resulted in slower convergence, while higher update frequencies (50 and 100) led to instability and failure to converge.

The second experiment highlighted the effects of noise on DQN performance. Standard DQN converged faster than

Noise Level	Convergence Episode
Standard_LOW	430
Standard_baseline	511
Noisy_baseline	583
Noisy_LOW	No Convergence
Noisy_MEDIUM	No Convergence
Noisy_HIGH	No Convergence
Standard_MEDIUM	No Convergence
Standard_HIGH	No Convergence

Table 3. Effect of noisy states on DQN and Noisy DQN performance.

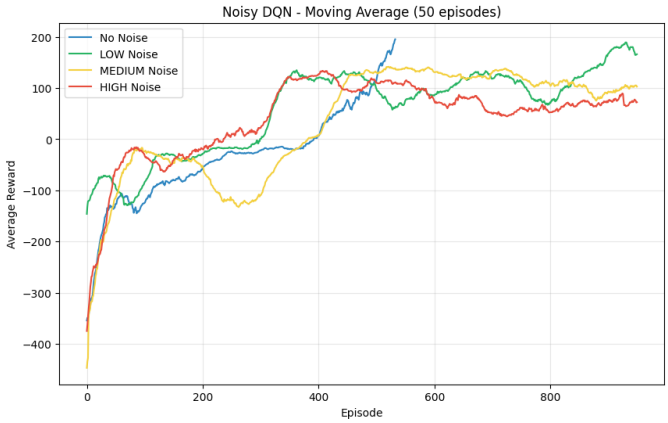

Figure 4. Impact of noisy states on Noisy DQN performance.

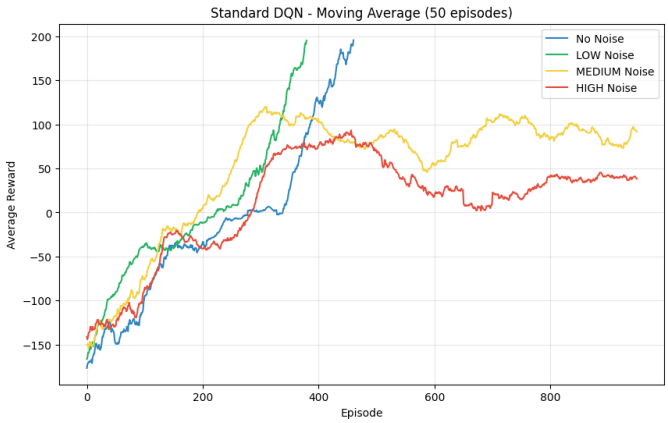

Figure 5. Impact of noisy states on DQN performance.

Noisy DQN, likely due to the additional stochasticity introduced by Noisy DQN's exploration strategy. However, Noisy DQN was still able to achieve optimal performance, albeit at a slower rate (episode 583 vs. 511). The presence of state noise significantly impacted training, with only the Standard_LOW noise condition achieving convergence (episode

430), whereas medium and high noise levels led to failure in training for both DQN variants.

Although none of the noisy variants converged, the average rewards of the noisy variants are always higher than that of the standard variants when experiencing the same noise.

Overall, hyperparameter tuning plays a crucial role in reinforcement learning stability and efficiency. Noisy DQN provides an alternative exploration strategy but requires more episodes to converge. State noise, especially at high levels, severely affects the learning process, indicating that robust state representations are necessary for effective policy learning. Noisy DQN deals with state noise much better than a standard DQN.

4.2 Experiment 2: DDPG

4.2.1 Hyperparameter Tuning. The hyperparameter tuning for the Deep Deterministic Policy Gradient (DDPG) algorithm involved testing different learning rates (Tau values) and noise levels. The results are summarized in Table 4 and Figures 6, 7 and 8.

Combination	Convergence Episode
Tau0.01_Noise0.2	100
Tau0.01_Noise0.3	153
Tau0.005_Noise0.1	186
Tau0.001_Noise0.3	205
Tau0.001_Noise0.2	214
Tau0.01_Noise0.1	232
Tau0.005_Noise0.3	288
Tau0.001_Noise0.1	337
Tau0.005_Noise0.2	403

Table 4. DDPG hyperparameter tuning results sorted by convergence speed.

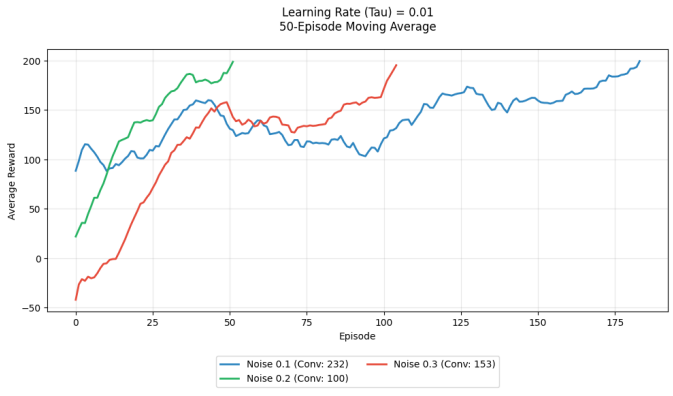

Figure 6. Performance of DDPG with Tau=0.01.

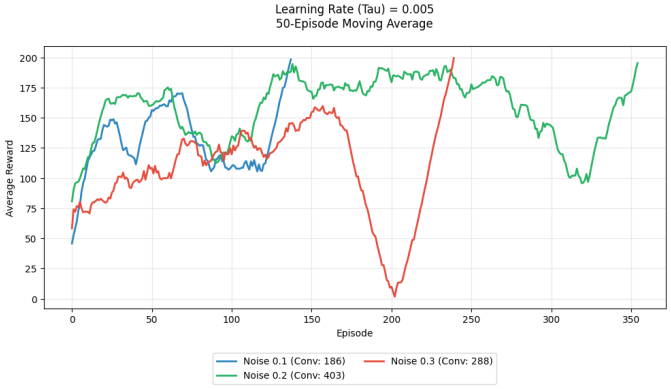

Figure 7. Performance of DDPG with Tau=0.005.

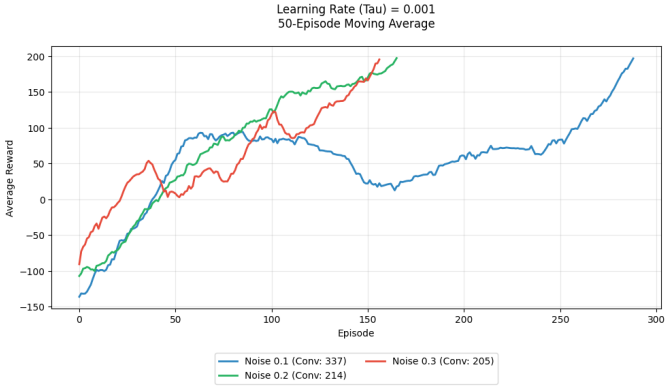

Figure 8. Performance of DDPG with Tau=0.001.

4.2.2 Noisy State in DDPG. The impact of state noise was analyzed in DDPG, and the results are shown in Table 5 and Figure 9.

Noise Level	Convergence Episode
LOW	226
MEDIUM	No Convergence
HIGH	No Convergence

Table 5. Effect of noisy states on DDPG performance.

4.2.3 Discussion and Conclusions. The hyperparameter tuning results for DDPG highlight the sensitivity of the algorithm to the choice of learning rate (Tau) and noise levels. Tau=0.01 with Noise=0.2 led to the fastest convergence at episode 100, reinforcing the importance of balancing exploration and exploitation in continuous action spaces. Higher noise levels (e.g., 0.3) tended to slow down convergence, suggesting that excessive noise hinders learning stability.

The impact of state noise was also pronounced in DDPG. While low noise levels (converging at episode 226) had a tolerable effect, medium and high noise levels resulted in

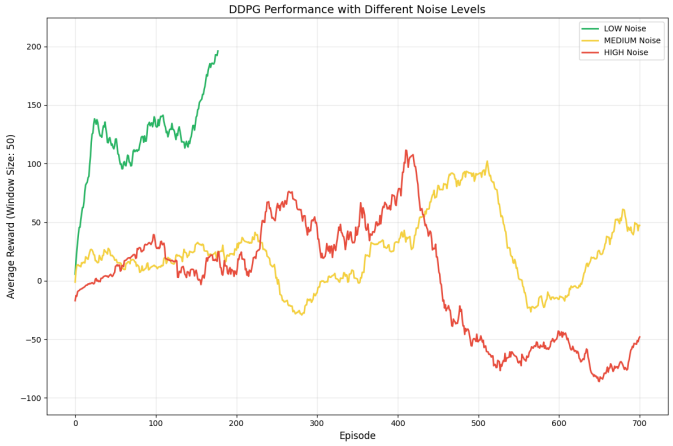

Figure 9. Impact of noisy states on DDPG performance.

failure to converge, underscoring DDPG's vulnerability to environmental instability. This suggests that robust state estimation and noise reduction techniques may be necessary for reliable DDPG deployment in real-world scenarios.

Another key observation was the significantly longer training time per episode for DDPG compared to DQN. This is expected given that DDPG operates in a continuous action space, requiring more complex policy updates and gradient computations.

In conclusion, while DDPG can effectively learn optimal policies, careful hyperparameter tuning is crucial. Additionally, mitigating the impact of state noise is essential for stable training, as higher noise levels drastically reduce the likelihood of convergence.

5 Conclusion

This study examined the application of Reinforcement Learning (RL) methodologies to control problems within the OpenAI Gym *LunarLander* environment. By conducting two distinct experiments, the effectiveness and robustness of Deep Q-Networks (DQN) and Deep Deterministic Policy Gradient (DDPG) algorithms were evaluated under varying conditions, including differing levels of state noise.

In the first experiment, a standard DQN model was implemented and compared against a Noisy DQN variant to address the discrete-action *LunarLander-v3* task. The results indicate that while Noisy DQN enhances exploration through adaptive strategies, it requires a greater number of training episodes to achieve convergence compared to the standard DQN model. Furthermore, an analysis of state noise revealed a substantial degradation in model performance as noise levels increased, leading to non-convergence in most scenarios. However, the Noisy DQN demonstrated superior resilience to noise compared to the standard DQN, suggesting its potential for enhanced robustness in uncertain environments.

In the second experiment, the DDPG algorithm was applied to solve the continuous-action *LunarLanderContinuous-v3* problem. A systematic hyperparameter tuning process indicated that a higher target update rate ($\tau = 0.01$) coupled with moderate exploration noise ($\sigma = 0.2$) yielded the fastest convergence. Nevertheless, the introduction of state noise proved to be a significant challenge for DDPG, with only the lowest noise level resulting in successful convergence. This finding underscores the algorithm's sensitivity to environmental uncertainty and suggests the necessity of additional robustness mechanisms for real-world applications.

Overall, the findings of this study highlight the critical importance of hyperparameter optimization and noise mitigation strategies in reinforcement learning. While Noisy DQN facilitates improved exploration, its advantages come at the expense of slower convergence. Similarly, DDPG demonstrates efficacy in continuous control tasks but remains highly susceptible to noise perturbations. Future research could explore the integration of state-estimation techniques and meta-learning strategies to enhance the stability and generalization capabilities of RL models in dynamic and noisy environments.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA: MIT Press, 2018.
[2] D. Silver, "Introduction to Reinforcement Learning," *DeepMind RL Course*, 2021. [Online]. Available: https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver
[3] V. Vella, *Basic RL Methods*, University of Malta, Lecture Notes, 2024.
[4] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
[5] V. Vella, *DQN Part 1*, University of Malta, Lecture Notes, 2024.
[6] V. Vella, *DQN Part 2*, University of Malta, Lecture Notes, 2024.
[7] M. Fortunato, M. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, *Noisy networks for exploration*, in *Proc. Int. Conf. Learning Representations (ICLR)*, 2018. [Online]. Available: https://arxiv.org/abs/1706.10295
[8] H. Karino, M. Imai, and T. Yamaguchi, *Directional noise for adaptive exploration in deep reinforcement learning*, in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 2018. [Online]. Available: https://arxiv.org/abs/1809.06570
[9] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3, pp. 229-256, 1992.
[10] V. Vella, *Policy Gradient Methods*, University of Malta, Lecture Notes, 2024.
[11] V. Vella, *Actor-Critic Methods*, University of Malta, Lecture Notes, 2024.
[12] V. Vella, *More State-of-the-Art RL Methods*, University of Malta, Lecture Notes, 2024.
[13] OpenAI, "Gymnasium: A Toolkit for Developing and Comparing Reinforcement Learning Algorithms," 2024. [Online]. Available: https://gymnasium.openai.com/
[14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.
[15] M. Hessel et al., "Rainbow: Combining Improvements in Deep Reinforcement Learning," *Proc. AAAI Conf. Artificial Intelligence*, 2018.
[16] P. Henderson et al., "Deep reinforcement learning that matters," *Proc. AAAI Conf. Artificial Intelligence*, 2018.

[17] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks," *Proc. Int. Conf. Artificial Intelligence and Statistics (AISTATS)*, 2011.

[18] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[19] S. Nair, "DQN and Hyperparameters: Studying the Effects of the Various Hyperparameters," 2023. [Online]. Available: https://saashanair.com/blog/blog-posts/dqn-and-hyperparameters-studying-the-effects-of-the-various-hyperparameters