Practical Work in AI Report

Robust Deep Reinforcement Learning against Adversarial Perturbations on State Observations

Chiara Gei
MSc Artificial Intelligence
Johannes Kepler University, Linz, Austria

Abstract

In Deep Reinforcement Learning (DRL), agents rely on state observations to perceive their environment. However, these observations may be distorted by adversarial attacks or simply by measurement uncertainty, errors, and noise. Such deviations from the true states can lead the agent to make sub-optimal decisions. In fact, the agent might believe it is in a certain state different from the actual one and take optimal actions for the observed state. These selected actions would generally be sub-optimal for the actual state. Previous studies have highlighted the susceptibility of DRL algorithms to state adversarial attacks and proposed approaches to deal with this problem and make the decision-making process of the agent more robust. The goal of this report is to highlight the above-mentioned vulnerability of DRL algorithms to adversarial attacks and compare the performance of state-of-the-art solutions addressing this problem when operating within the Lunar Lander environment.

1 Motivation

The motivation behind investigating the vulnerability of reinforcement learning agents under adversarial state perturbations stems from the need for a robust decision-making algorithm, which is capable of operating effectively in a real-world scenario characterized by noise and uncertainties.

Real-world environments are often subject to various sources of noise, disturbances, and uncertainties. Factors such as sensor inaccuracies, environmental variability, and external perturbations can significantly impact the performance of decision-making algorithms, potentially leading to sub-optimal or even dangerous outcomes. Thus, developing reinforcement learning techniques that can effectively navigate and cope robustly with these real-world challenges is essential for enabling autonomous systems to operate safely and efficiently in diverse environments.

Adversarial examples are inputs to machine learning models that have been intentionally perturbed to cause the model to make mistakes. As adversarial attacks are intentional and malicious attacks performed based on some knowledge about the system, they can be considered as a worst case scenario for the perturbations. Making the system robust to adversarial attacks means making it robust to common noise and random disturbances as well.

At this stage, the Lunar Lander environment serves merely as a toy problem for investigating the robustness of reinforcement learning algorithms to adversarial attacks. Understanding how reinforcement learning algorithms perform under such conditions provides valuable insights into their applicability and limitations in other practical settings.

Ultimately, the goal is to develop decision-making algorithms that can transition from simulated environments to real-world settings. By addressing the challenges posed by noise and uncertainties in simulation, we aim to train agents capable of making reliable decisions in real-world environments.

2 Methodology

In principle we want to assess the improvement in term of robustness of two methodologies, over the vanilla version of some DRL algorithms.

The first approach follows the one adopted in [3] and [1]. This approach will be referred to as adversarial training. In this case the model is initially trained the same way as the vanilla algorithm. After an initial phase the agent is presented, at regular intervals, with adversarially attacked observations. The idea behind this approach is that by dealing with perturbed observations at training time, the agent can be better prepared to cope with them at test time.

The second approach is based on the theoretically principled policy regularization approach proposed in in [8]. In this case a regularizer, which can be added as part of the policy optimization objective, is derived from the formalization of the State Adversarial Markov Decision Process (SA-MDP). This approach will be referred to as the robust or SA-MDP version. The derivation of the regularization term requires convex relaxation of neural networks. Convex relaxation of neural networks involves simplifying the non-convex optimization problem of training neural networks by approximating it with a convex optimization problem and it enables the analysis of the outer bounds of a neural network. In this report, convex relaxation is used as a black-box tool (provided by the `auto_LiRPA` library [7].

2.1 Vanilla vs Robust Deep Q-Networks

The vanilla Deep Q-Network (DQN) algorithm builds upon the classic Q-learning framework by leveraging deep neural networks to approximate Q-values [4]. The Q-value, also known as the action-value function, represents the expected cumulative reward an agent can obtain by taking a specific action in a particular state and then following a certain policy thereafter. In other words, it quantifies the goodness of an action taken from a specific state. The DQN algorithm uses an experience replay, where past experiences are stored in a buffer and sampled randomly during training. This approach not only breaks the correlation between consecutive experiences but also enhances training stability. Additionally, DQN employs a target network to compute target Q-values, which are updated less frequently than the main network's parameters, to further stabilize training. Exploration and exploitation are traded-off through an epsilon-greedy strategy.

In [8] the authors propose the following regularizer to make the DQN algorithm robust to perturbations:

$$\mathcal{R}_{DQN}(\theta) := \sum_{s} \max \left\{ \max_{\hat{s} \in B(s)} \max_{a \neq a^*} \{ Q_\theta(\hat{s}, a) - Q_\theta(\hat{s}, a^*(s)) \}, -c \right\} \tag{1}$$

Where:

- $s \in \mathcal{S}$ represent an environment state in the state space $\mathcal{S}$.
- $B(s)$ is a set of possible perturbations around the state s.
- $\hat{s} \in B(s)$ is a perturbed state.
- θ are the parameters of the Q-network used in the DQN algorithm to approximate the Q-value function.
- a represent an action that the agent can take in state s.
- $Q_\theta(\hat{s}, a)$ is the Q-value of taking action a in the perturbed state $\hat{s}$, as estimated by the Q-network parameterized by θ.
- $a^*(s) = \text{argmax}_a Q_\theta(s, a)$ is the optimal action in state s, which is the action that maximizes the Q-value according to the current policy in s.
- $Q_\theta(\hat{s}, a^*(s))$ is the Q-value of taking $a^*(s)$ in the perturbed state $\hat{s}$, as estimated by the Q-network parameterized by θ.
- c is a small positive constant value.

This regularizer aims at leaving the top-1 action unchanged under state perturbation. The maximization can be solved using convex relaxation of neural networks.

The vanilla DQN algorithm and the changes introduced by the robust version (in blue) are reported in Algorithm 1.

Algorithm 1 State-Adversarial Deep Q-Learning (SA-DQN). Differences compared to vanilla DQN in blue.

```
1:  Initialize current Q network $Q(s,a)$ with parameters $\theta$
2:  Initialize target Q network $Q'(s,a)$ with parameters $\theta' \leftarrow \theta$
3:  Initialize replay buffer $\mathcal{B}$
4:  for $t = 1$ to $T$ do
5:      With probability $\epsilon_t$ select a random action $a_t$, otherwise select $a_t = \arg\max_a Q_\theta(s_t, a)$
6:      Execute action $a_t$ in environment and observe reward $r_t$ and state $s_{t+1}$
7:      Store transition $\{s_t, a_t, r_t, s_{t+1}\}$ in $\mathcal{B}$
8:      Randomly sample a minibatch of $N$ samples $\{s_i, a_i, r_i, s'_i\}$ from $\mathcal{B}$
9:      For all $s_i$, compute $a_i^* = \arg\max_a Q_\theta(s_i, a)$
10:     Set $y_i = r_i + \gamma \max_{a'} Q'_{\theta'}(s'_i, a')$ for non-terminal $s_i$, and $y_i = r_i$ for terminal $s_i$
11:     Compute TD-loss for each transition: TD-L$(s_i, a_i, s'_i; \theta) =$ MSELoss$(y_i - Q_\theta(s_i, a_i))$
12:     Define $R_{DQN}(\theta) := \sum_i \max\left\{\max_{\hat{s}_i \in B(s_i)} \max_{a \neq a_i^*} \{Q_\theta(\hat{s}_i, a) - Q_\theta(\hat{s}_i, a_i^*)\}, -c\right\}$
13:     Use convex relaxations of neural networks to solve a surrogate loss of $R_{DQN}(\theta)$
14:         For all $s_i$ and all $a \neq a_i^*$, obtain upper bounds on $Q_\theta(s,a) - Q_\theta(s, a_i^*)$:
            $u_{a_i^*,a}(s_i;\theta) = \text{ConvexRelaxUB}(Q_\theta(s,a) - Q_\theta(s,a_i^*), \theta, s \in B(s_i))$
15:         Compute a surrogate loss: $\bar{R}_{DQN}(\theta) = \sum_i \max\left\{\max_{a \neq a_i^*}[u_{a_i^*,a}(s_i)], -c\right\}$
16:     Perform a gradient descent step to minimize $\frac{1}{N}\left[\sum_i \text{TD-L}(s_i, a_i, s'_i; \theta) + \kappa_{DQN} \bar{R}_{DQN}(\theta)\right]$
        with $\kappa_{DQN}$ weighting the regularization term
17:     Update target network every $M$ steps: $\theta' \leftarrow \theta$
18: end for
```

2.2 Vanilla vs Robust Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a policy gradient method. It ensures that the updates to the policy are not too drastic, which helps maintain stable learning. PPO operates by using a surrogate objective function that incorporates a clipping mechanism to restrict how much the policy can change at each update step. This prevents large updates that could destabilize training [6].

In [8] the authors propose the following regularizer to make the PPO algorithm robust to perturbations:

$$\mathcal{R}_{PPO}(\theta) = \frac{1}{2}\sum_s \max_{\hat{s} \in B(s)} (\mu_\theta(\hat{s}) - \mu_\theta(s))^\top \Sigma^{-1} (\mu_\theta(\hat{s}) - \mu_\theta(s)) := \frac{1}{2}\sum_s \max_{\hat{s} \in B(s)} \mathcal{R}_s(\hat{s}, \theta) \quad (2)$$

Where:

- $s \in \mathcal{S}$ represent an environment state in the state space $\mathcal{S}$.
- $B(s)$ is a set of possible perturbations around the state s.
- $\hat{s} \in B(s)$ is a perturbed state.
- θ are the parameters of the neural network.
- $\mu_\theta(\hat{s})$ represents the mean action produced by the policy network parameterized by θ for state $\hat{s}$.
- $\mu_\theta(s)$ represents the mean action produced by the policy network for state s.
- Σ is a diagonal covariance matrix independent of states s and $\hat{s}$, representing the covariance of action distributions.

Minimizing the above-mentioned regularizer involves tackling a min-max objective. Additionally, we have that $\nabla_{\hat{s}} \mathcal{R}(\hat{s}, \theta_\mu)|_{\hat{s}=s} = 0$. Therefore, solving the inner maximization problem to a local maximum using gradient descent is not feasible. One approach the authors propose to use to deal with this problem is convex relaxation of neural networks.

The vanilla PPO algorithm and the changes introduced by the robust version (in blue) are reported in Algorithm 2.

Algorithm 2 State-Adversarial Proximal Policy Optimization (SA-PPO). Differences compared to vanilla PPO in blue.

```
Input: Number of iterations $T$, a $\epsilon$ schedule $\epsilon_t$
1:  Initialize actor network $\pi(a|s)$ and critic network $V(s)$ with parameters $\theta$ and $\theta_V$
2:  for $t = 1$ to $T$ do
3:      Run $\pi_\theta$ to collect a set of trajectories $\mathcal{D} = \{\tau_k\}$ containing $|\mathcal{D}|$ episodes, each $\tau_k$ is a trajectory
        containing $|\tau_k|$ samples, $\tau_k := \{(s_{k,i}, a_{k,i}, r_{k,i}, s_{k,i+1})\}, i \in [|\tau_k|]$
4:      Compute cumulative reward $\hat{R}_{k,i}$ for each step $i$ in every episode $k$ using the trajectories and
        discount factor $\gamma$
5:      Update value function by minimizing the mean-square error:
            $$\theta_V \leftarrow \arg\min_{\theta_V} \frac{1}{\sum_k |\tau_k|} \sum_{\tau_k \in D} \sum_{i=0}^{|\tau_k|-1} (V(s_{k,i};\theta_V) - \hat{R}_{k,i})^2$$
6:      Estimate advantage $\hat{A}_{k,i}$ for each step $i$ in every episode $k$ using generalized advantage
        estimation (GAE) and value function $V_{\theta_V}(s)$
7:      Define the state-adversarial policy regularizer:
            $$R_{PPO}(\theta) := \sum_{\tau_k \in D} \frac{1}{|\tau_k|} \sum_{i=0}^{|\tau_k|-1} \max_{\bar{s}_{k,i} \in B(s_{k,i},\epsilon_t)} \mathrm{D}_{KL}(\pi(a|s_{k,i}) \| \pi(a|\bar{s}_{k,i}))$$
8:      Solve $R_{PPO}(\theta)$ using convex relaxations:
9:          $\bar{R}_{PPO}(\theta) := \text{ConvexRelaxUB}(R_{PPO}, \theta, \bar{s}_{k,i} \in B(s_{k,i}, \epsilon_t))$
10:     Update the policy by minimizing the SA-PPO objective (minimization solved using ADAM):
            $$\theta \leftarrow \arg\min_{\theta'} \frac{1}{\sum_k |\tau_k|} \left[ \sum_{\tau_k \in D} \sum_{i=0}^{|\tau_k|-1} \min\left( r_{\theta'}(a_{k,i}|s_{k,i})\hat{A}_{k,i}, C(r_{\theta'}(a_{k,i}|s_{k,i}))\hat{A}_{k,i} \right) + \kappa_{PPO}\, \bar{R}_{PPO}(\theta') \right]$$
        where $r_{\theta'}(a|s) := \frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)}$, $C(r) := \text{clip}(r_{\theta'}(a|s), 1-\epsilon_{\text{clip}}, 1+\epsilon_{\text{clip}})$ and with $\kappa_{PPO}$ weighting
        the regularization term
11: end for
```

3 Adversarial Attacks

To evaluate the performance and robustness of the reinforcement learning agents different types of attacks can be applied. In principle, the stronger the attack the better it can highlight the robustness properties of the agent.

3.1 Critic attack

A common approach taken in previous works [5] is based on the use of the gradient of the action value function $Q_\theta(s,a)$. The gradient in this case is used to determine the direction to take, in order to update the states adversarially in K steps. At each step the state is updated as follows:

$$s_{k+1} = s_k - \eta \cdot \text{proj}\left[\nabla_{s_k} Q(s_0, \pi(s_k))\right], \quad k = 0, \dots, K-1 \tag{3}$$

where proj$[\cdot]$ is a projection to $B(s)$, η is the learning rate, s_0 is the initial state under attack and π is the policy. This approch aims at finding a perturbed state $\hat{s} := s_K$ which leads to the selection of an action $\pi(\hat{s})$ that minimizes the action-value $Q(s_0, \pi(\hat{s}))$ at the original state s_0.

This approach however has some drawbacks: it relies on the action value function Q available during the training process, the attack strength depends on the critic quality, and it cannot be directly applied to actor-critic methods like PPO where the value function $V(s)$ instead of the action value function $Q(s,a)$ is used.

To cope with this limitations, the authors of [8] propose two critic-independent attacks, which can be used also in actor-critic methods like PPO: Robust Sarsa and Maximal Action Difference attacks.

The following subsections provide more details on these two attacks as well as on the the projected gradient descent attack, commonly used in DQN.

Note that in the evaluation section the critic attack is still applied in the PPO case. Since there is no $Q(s, a)$ available during training, the authors of [8] rely for this on the state value function $V(s)$ instead and find a state $\hat{s}$ that minimizes $V(\hat{s})$.

3.2 Robust Sarsa (RS) attack

This approach is based on the consideration that at evaluation time the policy π is already learned and fixed. Therefore, the corresponding action value function $Q^{\pi}(s, a)$ can be derived via on-policy temporal-difference (TD) algorithms similar to Sarsa. This approach does not need to have knowledge about the critic network during training. The action value function is learned as a neural network parameterized in θ with the objective of minimizing a loss function which includes a TD loss term as in Sarsa and an additional term to promote the robustness of Q with respect to the action (to provide a good direction for attacks):

$$L_{RS}(\theta) = \sum_{i=1}^{N} \left[r_i + \gamma Q_{\theta}^{\pi}(s_i', a_i') - Q_{\theta}^{\pi}(s_i, a_i)\right]^2 + \lambda_{RS} \sum_{i=1}^{N} \max_{\hat{a} \in B(a_i)} \left(Q_{\theta}^{\pi}(s_i, \hat{a}) - Q_{\theta}^{\pi}(s_i, a_i)\right)^2 \quad (4)$$

where N is the batch size. The batches contain N tuples of transitions (s, a, r, s', a') sampled from the agent interactions with the environemnt. The term λ_{RS} weights the robustness term with respect to the TD loss. $B(a_i)$ is a small set near action a_i (e.g., a l_{∞} ball of norm 0.01 when action is normalized between 0 to 1). Also in this case the inner maximization can be solved using convex relaxation. Once the action value function Q_{θ}^{π} is learned, it can be used to perform critic-based attacks as discussed above.

3.3 Maximal Action Difference (MAD) attack

MAD is another simple critic independent attack. Here the goal is to find an adversarial state $\hat{s}$ by maximising the Kullback Leibler divergence $D_{KL}(\pi(\cdot|s) \| \pi(\cdot|\hat{s}))$. In the case of actions parameterized by Gaussian mean $\pi_{\theta}(s)$ and covariance matrix Σ (independent of s), this can be achieved by minimizing $L_{MAD}(\hat{s}) := -D_{KL}(\pi(\cdot|s) \| \pi(\cdot|\hat{s}))$:

$$\text{argmin}_{\hat{s} \in B(s)} L_{MAD}(\hat{s}) = \text{argmax}_{\hat{s} \in B(s)} \left(\pi_{\theta}(s) - \pi_{\theta}(\hat{s})\right)^T \Sigma^{-1} \left(\pi_{\theta}(s) - \pi_{\theta}(\hat{s})\right)$$

3.4 Projected Gradient Descent (PGD) Attack for DQN

In PGD, the attack algorithm iteratively perturbs an input example in the direction of increasing the model's loss, and consequentely the likelihood of misclassification, while ensuring that the perturbed example remains within a specified constraint set. The starting point for this process is an uncorrupted input example that the model correctly classifies. The gradient of the model's loss function with respect to the input example is computed. This gradient indicates the direction in which the input needs to be modified to increase the loss. The original input is then modified by adding a small perturbation in the direction of the gradient, scaled by a small step size (learning rate) to control the magnitude of the perturbation. As the resulting modified input may not satisfy certain constraints, such as remaining within a bounded region of the input space or preserving certain properties of the original input, the perturbed input is projected back onto the constraint set to ensure it satisfies the constraints. This process is repeated for a certain amount of iterations. This attack is commonly used for DQNs [5]. The PGD attack perturbs the state so that the Q-network outputs an action other than the optimal action a^* for the original state. To do so the initial state is modified by taking K steps as follows:

$$s_{k+1} = s_k + \eta \, \text{proj}[\nabla_{s_k} \mathcal{H}(Q_{\theta}(s_k, \cdot), a^*)], \quad s_0 = s, \quad k = 0, ..., K-1 \quad (5)$$

where proj$[\cdot]$ is the projection operator depending on $B(s)$, η is the learning rate, and $\mathcal{H}(Q_\theta(s_k, \cdot), a^*)$ is the cross-entropy loss between the output logits of $Q_\theta(s_k, \cdot)$ and the one-hot-encoded distribution of $a^* := \arg\max_a Q_\theta(s, a)$. Therefore, the steps are taken in the direction which maximises this cross-entropy. To guarantee that the final state obtained by the attack is within an ℓ_∞ ball around s ($B(s) = \{\hat{s} : s^- \leq \hat{s} \leq s^+\}$), the projection proj$[\cdot]$ is a sign operator, and η is typically set to $\eta = \frac{\epsilon}{K}$.

4 Lunar Lander Environment

The selected environment is the Gym Lunar Lander simulation environment commonly used in reinforcement learning research and experimentation. It is part of the OpenAI Gym toolkit, which provides a collection of environments for developing and testing reinforcement learning algorithms [2].

This environment was selected as it can be approached both as a discrete and continuous action space problem, accommodating different reinforcement learning algorithms and methodologies. Additionally, the choice was influenced by the constraints posed by the available computational resources.

The state is a vector of 8 elements, which includes: the coordinates of the lander in two dimensions, its linear velocities in two dimensions, its angle, its angular velocity, and two boolean parameters to determine whether each leg is in contact with the ground. The vector with minimum state values is [-1.5, -1.5, -5., -5., -3.14, -5., -0., -0.], the vector with maximum state values is [1.5, 1.5, 5., 5., 3.14, 5., 1., 1.]. This information is important to better understand how the maximum perturbations are defined.

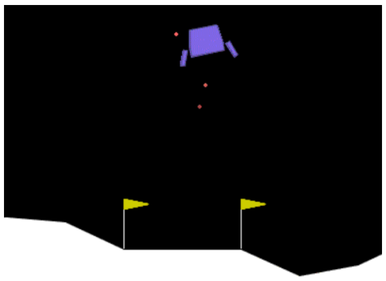
Figure 1: Lunar Lander environment.

There are two environment versions: discrete or continuous. In the discrete version the four discrete actions are: do nothing, fire left engine, fire main engine, fire right engine. In the continuous version the actions represent the throttle of the engines. The action in this case is a vector with two elements containing throttle of the main engine and of the lateral boosters.

In this environment, the goal is to control a lunar lander to safely land it on the surface of the moon. The landing location is always at coordinates (0,0). Otherwise, the episode terminates if the lander crashes, if it gets outside of the viewing area, or if it is not awake.

The environment provides feedback to the agent based on its actions and the resulting state of the lander. The agent receives rewards or penalties depending on how well it performs the task. Moving from the starting point at the top of the screen to the landing pad on the bottom and coming to rest provides about 100-140 points. Moving away from the landing pad leads to negative reward. Crashing leads to -100 points. Coming to rest guarantees +100 points. In case each leg is in contact with the ground +10 points are given. Firing the main engine is -0.3 points each frame. Firing the side engine is -0.03 points each frame. At 200 points the environment is considered to be solved.

5 Experimental Setup

5.1 Maximum allowed perturbation

In the experiments in [8] the authors consider a perturbation set $B(s)$ which is defined as an ℓ_∞ norm ball around the states s with a radius ϵ: $B(s) = \{\hat{s} : \|s - \hat{s}\|_\infty \leq \epsilon\}$. In their study, they deal with state vectors where all entries fall within the same value range, such as in images where all pixels take values between 0 and 255. In the case of Lunar Lander, however, the entries of the state vector vary in their respective ranges. For consistency and clarity in defining perturbations, it was chosen to express them relative to the range of each entry in the state vector. For example, let's consider the first element of the state vector in Lunar Lander. It ranges from -1.5 to 1.5, giving it a range of 3 units. Therefore, with a perturbation rate of $\epsilon = 0.01$, the perturbation magnitude is calculated

Table 1: DQN Hyperparameters Table

Hyperparameter	**Description**	**Value**
gamma (γ)	Discount factor	0.99
lr	Learning rate	5e-4
batch_size	Size of the batches	128
act_epsilon_start	Initial epsilon for ϵ-greedy action selection	1
act_epsilon_final	Final epsilon for ϵ-greedy action selection	0.05
act_epsilon_duration	Iterations to get from act_epsilon_start to act_epsilon_final	50000
replay_initial	Minimum size of the replay buffer to start learning	100000
buffer_capacity	Capacity of the replay buffer	100
attack_epsilon	Magnitude of the attack	0.01
niters	Number of PGD iterations	10
kappa (κ)	Term to trade-off the regularization term and the main loss	0.025

as $0.01 \times 3 = 0.03$. The perturbation added to the first element of the state vector ranges actually between -0.015 and 0.015. This transformation remains consistent across all methodologies, during both training and testing phases.

5.2 DQN Model and Hyperparameters

For the sake of reproducibility, the general hyperparameters used at training time for DQN are reported in Table 1. Most of the hyper-parameters are common between the vanilla, the adversarial and robust versions. Adversarial training additionally needs the magnitude of the attack applied at training time. The robust version requires an additional regularization parameter κ which determines the weight given to the regularizers in relation to the rest of the loss function.

The model consists of a linear layer with 8 input features (the number of states) and 128 output features, followed by a ReLU activation, another linear layer with 128 input and output features, another ReLU activation, and a final linear layer with 128 input features and 4 output features (the number of actions in the discrete case).

5.3 PPO Model and Hyperparameters

The hyperparameters used in PPO are reported in Table 2. Also in this case most of the hyper-parameters are common between the vanilla, the adversarial and robust versions. The adversarial training is presented with perturbed states during training, whose magnitude is given by the attack_epsilon value. As for DQN, κ tunes the weight given to the regularizers in relation to the rest of the loss function in the robust version.

The policy network consists of a linear layer with 8 input features (the number of states) and 64 output features, followed by a Tanh activation, another linear layer with 64 input and output features, another Tanh activation, and a final linear layer with 64 input features and 2 output features (the number of actions in the continuous case).

The value network consists of a linear layer with 8 input features (the number of states) and 64 output features, followed by a Tanh activation, another linear layer with 64 input and output features, another Tanh activation, and a final linear layer with 64 input features and 1 output feature.

6 Experimental Results

The three versions of the DQN and PPO algorithms show different behaviours already during training. Once trained they can be tested in unperturbed conditions as well as under attack. The following subsections discuss the behaviour of the algorithms in the training phase as well as the results of the testing phase.

Table 2: PPO Hyperparameters Table

Hyperparameter	Description	Value
gamma (γ)	Discount factor	0.99
ppo_lr	Learning rate for the policy network	0.0001
val_lr	Learning rate for the value network	0.00025
lambda	Factor used in Generalized Advantage Estimation (GAE)	0.95
clip_eps	Controls the range of permissible policy updates	0.2
entropy_coeff	Entropy coefficient	0.0
num_minibatches	Number of minibatches	32
num_steps	Number of interactions with the environment	3000
attack_epsilon	Magnitude of the attack	0.01
kappa (κ)	Term to trade-off the regularization term and the main loss	0.025

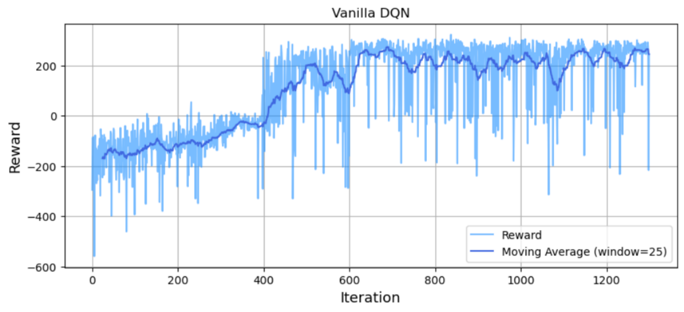

Figure 2: Reward at training time of the **vanilla DQN** algorithm.

6.1 DQN: Training Phase

In all three variants the model is trained for 1300 episodes. The evolution of the reward over the episodes is reported in Figure 2 for the vanilla algorithm, in Figure 3 for the adversarially attacked DQN algorithm, and in Figure 4 for the robust DQN algorithm.

By looking at these rewards evolutions we can already notice some important differences. The vanilla DQN version, after about 700 episodes reaches on average rewards close to 200. However, we notice that there is a very high variability with peaks that still reach negative rewards in this final phase of the training. In the case of the adversarially attacked DQN algorithm we can see that the attacks actually hinder the learning of an optimal policy. In fact, in this case the rewards do not settle to a value around 200, but below 100. Finally, in the robust DQN plot we can notice that not only the reward settles to values above 200, but it also shows a stabler trend.

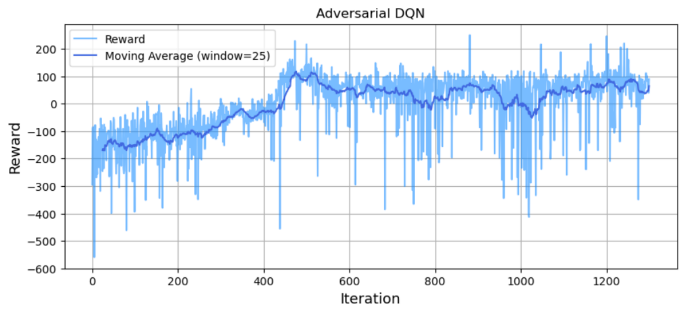

Figure 3: Reward at training time of the **adversarially trained DQN** algorithm.

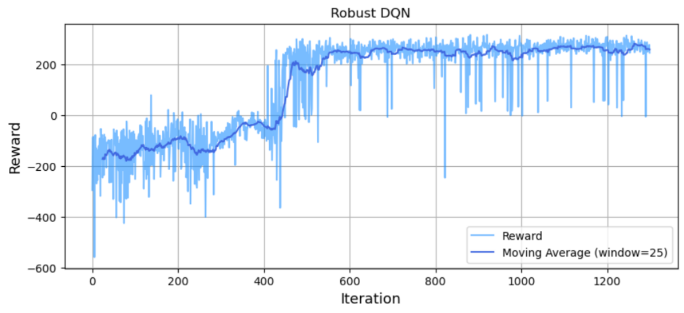

Figure 4: Reward at training time of the **robust DQN** algorithm.

Table 3: Experimental Results DQN. Average episode rewards ± std and median over 500 episodes

Attack at test time	Metric	Vanilla DQN	Adv. Training DQN	Robust SA-MDP DQN
No attack	(mean ± std)	245.61 ± 79.76	150.61 ± 153.69	252.40 ± 63.40
	(median)	267.84	224.28	268.90
10 steps PGD	(mean ± std)	80.39 ± 90.29	36.14 ± 87.98	152.86 ± 72.08
	(median)	96.85	34.81	152.93

6.2 DQN: Evaluation of Vanilla, Adversarial and Robust Versions

In the case of DQN the attack applied is a PGD attack as described in 3.4. More specifically, as reported in the hyperparameters table, the algorithms are evaluated under 10-step PGD attacks. The results are evaluated over 500 test episodes. In Table 3 we report the results of this analysis. We can see that if we test the environment without state observation perturbations the vanilla DQN and robust DQN show comparable performance, with the robust version achieving slightly higher average rewards. The DQN algorithm trained with adversarial attacks instead shows a significant degradation in terms of reward, even in unperturbed conditions. Therefore, attacking the environment during training not only does not improve robustness under attack but degrades performance also in normal unperturbed conditions. Under attack robust DQN performs better than the vanilla DQN algorithm and the adversarially trained DQN. A more complete overview of the distribution of the rewards over the different runs is provided in Figure 5.

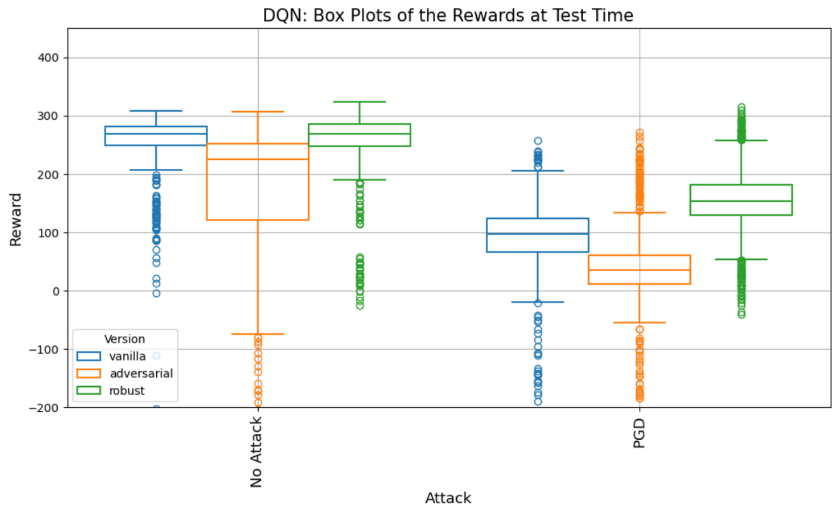

Figure 5: Experimental Results DQN. Box plot of the rewards at test time.

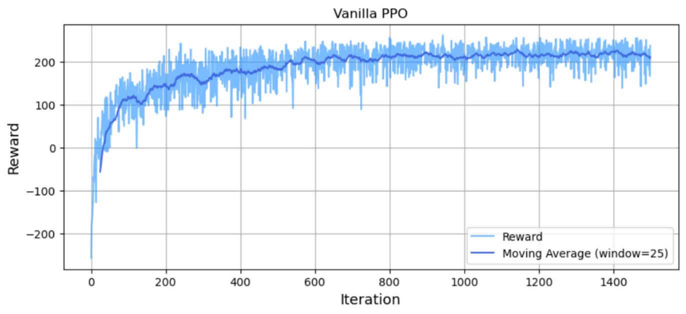

Figure 6: Reward at training time of the **vanilla PPO** algorithm.

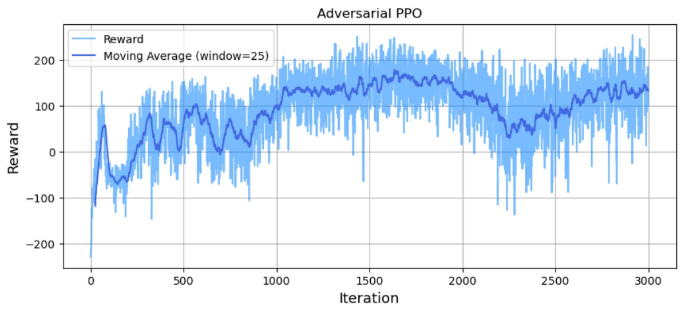

Figure 7: Reward at training time of the **adversarially trained PPO** algorithm.

6.3 PPO: Training Phase

In all three variants the model is trained for 1500 episodes. The evolution of the reward over the episodes is reported in Figure 6 for the vanilla algorithm, in Figure 7 for the adversarially attacked PPO algorithm, and in Figure 8 for the robust PPO algorithm.

The behaviour of the three versions is similar to the one observed for the DQN algorithm. The vanilla PPO version reaches rewards above 200. The rewards of the adversarial PPO version strongly oscillate and remain mostly below 200. Robust PPO reaches rewards between 200 and 300. The stability of the rewards over the iterations for the vanilla and robust PPO are in this case comparable.

6.4 PPO: Evaluation of Vanilla, Adversarial and Robust Versions

In the case of PPO the following attacks are applied: uniformly distributed random perturbation, critic attack (as defined in 3.1), MAD attack (as described in 3.2), RS attack (as described in 3.3) and a combination of RS and MAD. Considering the increased amount of attacks, in this case the results are evaluated over 100 test episodes. In Table 4 we report the results of this analysis in terms of mean,

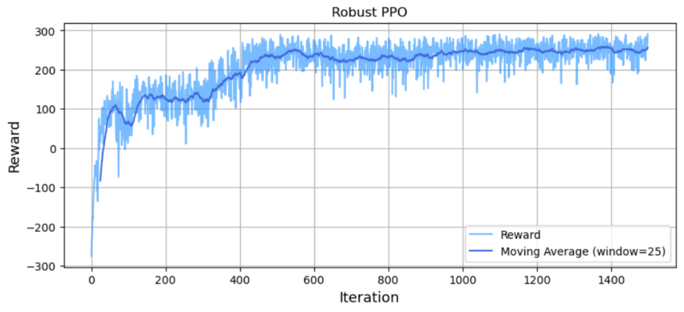

Figure 8: Reward at training time of the **robust PPO** algorithm.

Table 4: Experimental Results PPO. Average episode rewards ± std and median over 500 episodes

Attack at test time	Metric	Vanilla PPO	Adv. Training PPO	Robust PPO
No attack	(mean ± std)	222.06 ± 63.77	230.78 ± 68.58	241.18 ± 80.09
	(median)	239.39	247.46	268.26
Random (0.01)	(mean ± std)	207.86 ± 60.04	207.67 ± 89.76	250.60 ± 72.33
	(median)	226.61	244.09	267.89
Critic (0.01)	(mean ± std)	198.65 ± 63.06	156.79 ± 106.25	267.49 ± 58.42
	(median)	222.12	223.24	277.88
RS (0.01)	(mean ± std)	164.63 ± 87.57	128.10 ± 102.31	156.93 ± 121.22
	(median)	208.61	59.62	203.10
MAD (0.01)	(mean ± std)	162.07 ± 58.73	178.37 ± 101.16	258.27 ± 70.64
	(median)	144.97	233.94	276.21
RS+MAD (0.01)	(mean ± std)	152.20 ± 66.75	149.51 ± 99.92	204.44 ± 106.48
	(median)	137.93	157.62	249.31

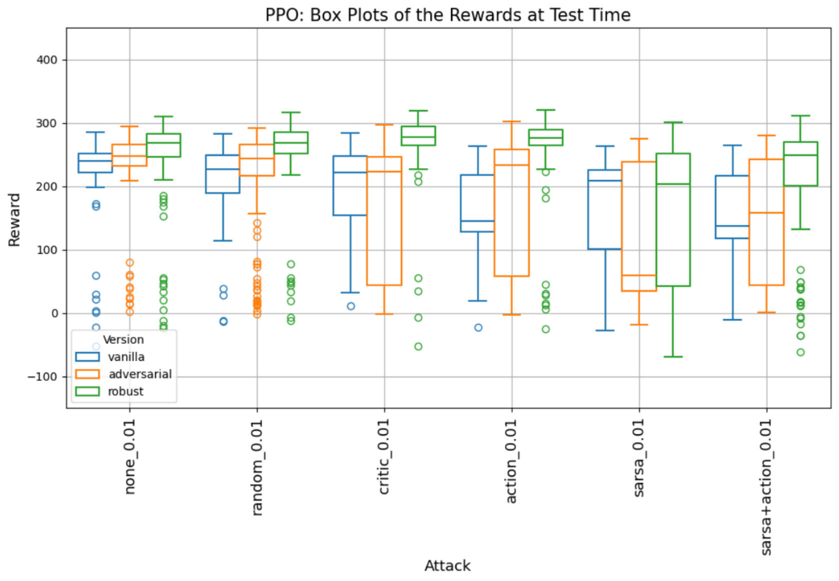

Figure 9: Experimental Results PPO. Box Plot of the Rewards at Test Time.

median and standard deviation of the rewards over the test episodes. The overall distribution of the rewards over the different runs is provided in Figure 9. In general the same considerations made for DQN apply also to the PPO case. Furthermore, we can see that the RS and MAD attacks are the ones with the strongest impact on the robust PPO algorithm.

As random noise is a very common perturbation in many real applications, a dedicated analysis was conducted considering increasing levels of random noise. More specifically, the noise considered was uniformly distributed within an interval defined by the perturbation rate ϵ in the way described in 5.1. The evaluation was run for ϵ equal to: 0.01, 0.025, 0.05, 0.075, 0.1. Where the ϵ considered during training for the adversarial and robust PPO version was $\epsilon = 0.01$. The results obtained are reported in Table 5 and in Figure 10 and show that the robust PPO version is not much impacted by the noise even for noise levels higher than the maximum perturbation seen at training time. Even with $\epsilon = 0.05$ robust PPO performs as well as it would perform in unperturbed conditions. For higher values of ϵ the performance in terms of rewards decreases but still outperforms the vanilla and adversarial versions.

7 Conclusions

DRL algorithms are vulnerable to adversarial attacks on state observations. These attacks can distort the agent's perception of its environment, leading to sub-optimal decision-making. Adversarial training tries to prepare the agent for perturbed observations by exposing it to adversarial attacks

Table 5: Experimental Results PPO under Random Noise. Average episode rewards ± std and median over 500 episodes

Attack at test time	Metric	Vanilla PPO	Adv. Training PPO	Robust PPO
Random (0.01)	(mean ± std)	207.86 ± 60.04	207.67 ± 89.76	250.60 ± 72.33
	(median)	226.61	244.09	267.89
Random (0.025)	(mean ± std)	167.35 ± 63.82	184.33 ± 92.54	258.24 ± 66.88
	(median)	161.75	232.04	276.71
Random (0.05)	(mean ± std)	132.51 ± 66.62	113.42 ± 70.37	253.20 ± 70.15
	(median)	133.33	127.99	272.29
Random (0.075)	(mean ± std)	59.46 ± 53.55	100.77 ± 58.55	184.12 ± 114.19
	(median)	74.57	116.37	262.96
Random (0.1)	(mean ± std)	28.53 ± 48.79	72.22 ± 49.36	99.58 ± 122.23
	(median)	33.15	76.85	44.37

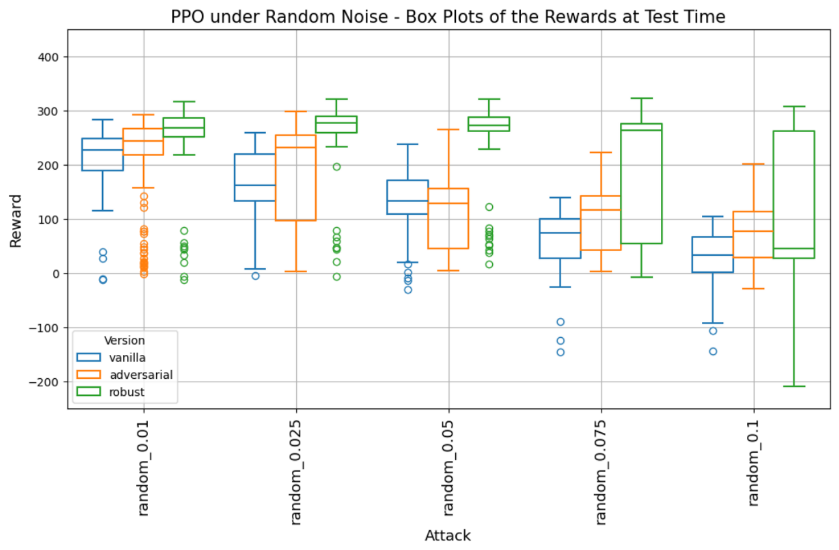

Figure 10: Experimental Results PPO under Random Noise. Box Plot of the Rewards at Test Time.

during training. The robust methodology proposed in [8] based on a policy regularization approach aims to provide a more principled way to enhance the robustness of decision-making processes.

In this report both methodologies were applied to the Lunar Lander Environment both in the discrete and continuous settings, using DQN and PPO algorithms. The results are in line with the ones presented in [8] with the robust version of the algorithms achieving higher average rewards under all conditions.

The insights gained from testing in the Lunar Lander environment are a first step for understanding the applicability and limitations of these algorithms in real-world settings. By addressing the challenges posed by perturbations in simulation, the ultimate goal is to develop DRL agents capable of making reliable decisions in diverse, real-world environments. As adversarial attacks represent a worst-case scenario for perturbations, making DRL agents robust against such attacks would also enhance their resilience to common noise and random disturbances.

References

[1] Vahid Behzadan and Arslan Munir. Whatever does not kill deep reinforcement learning, makes it stronger, 2017.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] Jernej Kos and Dawn Song. Delving into adversarial attacks on deep policies, 2017.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[5] Anay Pattanaik, Zhenyi Tang, Shuijing Liu, Gautham Bommannan, and Girish Chowdhary. Robust deep reinforcement learning with adversarial attacks, 2017.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[7] Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond, 2020.

[8] Huan Zhang, Hongge Chen, Chaowei Xiao, Bo Li, Mingyan Liu, Duane Boning, and Cho-Jui Hsieh. Robust deep reinforcement learning against adversarial perturbations on state observations, 2021.