From REINFORCE to DORAEMON: Enhancing Policy Robustness in MuJoCo Hopper

Calabrese Lorenzo (s345902), Calfapietro Flavia (s345862),
Melchionda Lorenzo (s339805), Spedicato Giuseppe (337116).
Politecnico di Torino, Turin (Italy)
https://github.com/beppSpedicato/ReinforcementLearning

Abstract

This project investigates the application of Reinforcement Learning algorithms to train a simulated agent for the jumping task in the Hopper environment from MuJoCo. We began by implementing the REINFORCE algorithm, both with and without a baseline, to analyze how variance reduction impacts learning. We then adopted the Actor-Critic approach, which combines an Actor network that selects actions and a Critic network that evaluates them. These components are updated through gradient descent, with their respective losses weighted to balance their influence. To further enhance performance, we implemented Proximal Policy Optimization (PPO), a policy gradient method that stabilizes training via a clipped objective function. We optimized hyperparameters, such as clip range, learning rate, and discount factor, through grid search and averaged results over three random seeds to ensure robustness. PPO showed limited generalization to environments with altered dynamics, such as changes in torso mass. To mitigate this sim-to-real gap, we employed Domain Randomization techniques, specifically Uniform Domain Randomization (UDR) and DORAEMON, which inject variability into training by randomizing physical parameters. These strategies significantly improved the agent's ability to deal with unseen scenarios.

1. INTRODUCTION

Reinforcement Learning (RL) is an advanced branch of machine learning inspired by the way humans learn through trial and error. In this paradigm, an agent learns to interact with a continuously evolving environment, progressively improving its decisions to achieve specific goals.

Mathematically, the problem is modeled using a Markov Decision Process (MDP), which describes a system where decisions are made sequentially under uncertainty [1]. An MDP is defined by:

- A set of states $\mathcal{S}$, representing all possible situations the agent can be in;
- A set of actions $\mathcal{A}$ available to the agent;
- A transition function $P(s'|s,a)$, which specifies the probability of moving from state s to state s' after taking action a;
- A reward function $R(s,a)$, giving the immediate reward received after performing action a in state s;
- A discount factor γ, which balances the importance of future rewards relative to immediate ones [2].

The agent acts autonomously without direct supervision, guided by a strategy known as the *policy*, denoted $\pi(a|s)$. This policy determines the action to take given the current state and can be deterministic or stochastic. In complex environments, policies are often represented by deep learning models such as neural networks [3].

The main goal of RL is to discover a policy that maximizes the expected cumulative reward, even when the environment dynamics and reward functions are unknown in advance. To achieve this, the agent explores and interacts with the environment in episodes, which end when certain conditions are met, such as reaching a terminal state or a time limit.

During each episode, the agent observes the current state, selects an action, receives feedback in the form of a reward, and updates its policy to improve future decisions, aiming to maximize the cumulative reward over time [4].

2. THE GYM HOPPER ENVIROMENT

The **Hopper** environment, included in the OpenAI Gym suite and built on the MuJoCo physics engine, simulates a planar, one-legged robotic system whose task is to learn to perform dynamic forward hopping. The environment models physical dynamics realistically, requiring the agent to

coordinate movement in a way that is both efficient and stable under gravity and inertia.

The robot consists of four rigid segments—torso, thigh, leg, and foot—connected by three actuated hinge joints. Each joint is controlled by applying a torque, forming a continuous **action space** of dimension three. Actions are represented as NumPy arrays with values in the range $[-1, 1]$, where each component controls the torque applied to the hip, knee, and ankle joints, respectively.

The **state space** is also continuous and 11-dimensional, combining positional and velocity information. It includes the torso's vertical position, the angles of all three joints, and the associated linear and angular velocities. This detailed representation allows the agent to perceive the full physical state of the hopper at every timestep.

To evaluate generalization capabilities, two variants of the Hopper environment are often used: **Source** and **Target**. In the Source environment, the mass values for each link are approximately: torso – 2.53 kg, thigh – 3.93 kg, leg – 2.71 kg, and foot – 5.09 kg. In the Target version, the torso is made heavier (3.53 kg), while all other masses remain unchanged. This slight but significant alteration is designed to assess the robustness of policies transferred across nearly identical dynamics, highlighting the challenges of sim-to-real or domain transfer in RL.

Training takes place on flat terrain. The agent is rewarded for forward motion and maintaining balance and episodes terminate either when the robot becomes unstable or after a fixed number of steps (typically 1000).

Crucially, the environment must be reset after each episode. Continuing without a reset leads to invalid episodes where observations and rewards no longer reflect meaningful dynamics, corrupting the learning process.

3. REINFORCE

The basic idea of this algorithm is to have the agent interact with the environment through complete episodes: at each step the state, the action chosen by the agent, the reward obtained and the next state is recorded. At the end of the episode, all this data is used to update the policy in order to maximize the expected cumulative reward [5]. In our case, the policy is represented by a neural network parameterised by a vector of parameters $\theta \in \mathbb{R}^d$, and the policy function $\pi(a \mid s, \theta)$.

The updating of the policy parameters is done using the gradient of the objective function $J(\theta)$, according to

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{1}$$

where α is the learnign rate and $\nabla J(\theta_t)$ is a stochastic estimate of the gradient of performance with respect to the policy parameters.

The algorithm used in this first part is REINFORCE [2], one of the simplest and best known policy gradient methods. In REINFORCE, the policy is updated at the end of each episode using the following formula:

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla_\theta \log \pi(A_t \mid S_t, \theta_t) \tag{2}$$

where

- A_t is the action at time t,
- S_t is the corresponding state,,
- π is the policy (a normal distribution in our experiments),
- G_t is the cumulative reward discounted from t,
- $\nabla_\theta \log \pi$ is the gradient of the log-probability of the action taken.

The value of G_t is calculated as the sum of the discounted future rewards

$$G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i \tag{3}$$

where $\gamma \in [0, 1)$ is the *discount factor* and T is the final time step (end of episode).

The advantage of this approach is that it allows us to reinforce the actions that produced high rewards, progressively improving the policy [5]. However, a known disadvantage of REINFORCE is the high variance of the update, which can make learning unstable and slow [6].

3.1. REINFORCE with constant baseline

To reduce the variance without introducing bias, we introduced a constant baseline in the update. In this case, the formula becomes:

$$\theta_{t+1} = \theta_t + \alpha (G_t - b) \nabla_\theta \log \pi(A_t \mid S_t, \theta_t) \tag{4}$$

where b is the baseline.This value is subtracted from the cumulative reward, so that only actions that lead to higher than average rewards are reinforced. The baseline then acts as a normalizer, improving the stability of the updates [6].

In our project we experimented with different baseline values: 0 (REINFORCE), 20 e 50.

3.2. Results and considerations

The results (Figure 1) clearly show the impact that the choice of baseline has on learning performance. In particular, we observed that with a baseline of 0, the agent receives positive updates even for unprofitable actions, since each cumulative reward is treated as significant. This results in a strong variance in upgrades, making training unstable:

sometimes you get high rewards, but progress is slow and inconsistent.

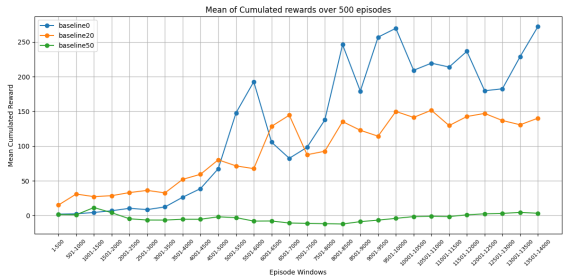

Figure 1. Reward behavior during training with different baseline values.

Setting the baseline to 20 leads to a noticeable change in the agent's behavior. Only actions that produce rewards significantly above this threshold are reinforced, resulting in more stable learning. Although the average rewards are lower than in the baseline-free case, this configuration proves to be the most balanced, allowing the policy to evolve consistently while maintaining a solid trade-off between effectiveness and reliability.

In contrast, a high baseline such as 50 hampers learning: most actions produce rewards below the threshold and are thus penalized, even when beneficial. This leads the agent to misinterpret good behavior as ineffective, causing learning to degrade into randomness.

4. ACTOR CRITIC

Another common approach in Reinforcement Learning is the Actor-Critic algorithm, which integrates two roles within the agent: the **Actor**, responsible for selecting actions, and the **Critic**, responsible for evaluating them. This combination leads to faster and more stable learning compared to reward-only strategies like REINFORCE.

The **Actor** uses a policy function to turn observed states into a probability distribution over actions. Its goal is to learn which actions bring higher rewards over time.

The **Critic** evaluates the Actor's choices by estimating the expected cumulative reward from the current state. [7]

The core of the algorithm lies in their interaction. After receiving feedback, the Critic computes the *temporal difference error*—the difference between expected and actual outcomes. This value is used to:

- allow the **Critic** to update its value predictions;
- guide the **Actor** to adjust its policy by reinforcing better-than-expected actions and discouraging worse ones.

Mathematically, the Critic updates its ϕ parameters by minimizing the error between the estimated value and the received value [7]:

$$\phi_{t+1} = \phi_t + \alpha_\phi \cdot \delta_t \cdot \nabla_\phi V_\phi(s_t) \tag{5}$$

where the TD error is defined as:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{6}$$

In parallel, the Actor updates its θ parameters by reinforcing the actions that the Critic has judged positively:

$$\theta_{t+1} = \theta_t + \alpha_\theta \cdot \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A(s_t, a_t) \tag{7}$$

where $A(s_t, a_t)$ represents the *advantage* of the action, i.e. the difference between the return obtained and the average estimate of the value in that state. The model parameters are updated by minimizing a combined loss function, defined as:

$$\mathcal{L}_{tot}(\theta) = \alpha_1 \cdot \mathcal{L}_{actor}(\theta) + \alpha_2 \cdot \mathcal{L}_{critic}(\theta) \tag{8}$$

Where:

- θ represents the parameters shared between actor and critic-
- $\mathcal{L}_{\text{actor}} = -\log \pi_\theta(a_t \mid s_t) \cdot \hat{A}_t$, with $\hat{A}_t$ the estimated advantage.
- $\mathcal{L}_{\text{critic}} = \left(V_\theta(s_t) - \hat{R}_t\right)^2$, i.e. the quadratic error between the estimated value and the expected return.
- α_1: indicates how much importance is given to the updating of the policy, i.e. the agent's behaviour.
- α_2: indicates how much importance is given to the estimation of the value function, i.e. the accuracy of the evaluation of the states.

To find the optimal balance between these last two terms, we used **Optuna**, a framework for the automatic optimisation of hyperparameters based on Bayesian Optimisation.

The Actor-Critic approach has the advantage of combining the strengths of policy-based and value-based methods. Thanks to this synergy, the agent is able to adapt better even in complex and dynamic environments.

Results

This structure resulted in significant performance benefits. In particular, the agent was able to converge much more quickly towards an effective policy compared to REINFORCE. Actor-Critic performs more frequent and stable updates, fostering a more fluid and directional learning dynamic. The values found for α_1 and α_2 are 0.0134 and 0.0097. The agent, after a short initial exploration phase,

showed a significantly better average reward trend (Figure 2), reaching higher values in less time, with peaks of up to 500. Furthermore, the trajectory of the cumulative average reward appeared much smoother, with fewer fluctuations and greater consistency in the increase in performance episode after episode.

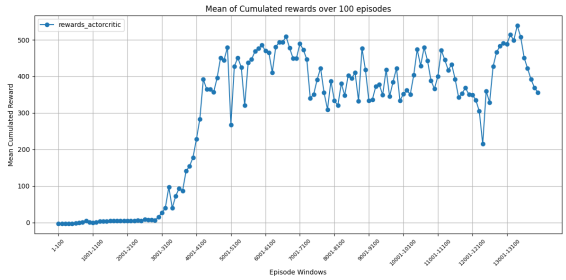

Figure 2. Reward trend with Actor-Critic.

After reaching peak levels (between 5,000 and 7,000 episodes), there is a phase of fluctuation and slight descent, probably due to the variability of the environment or a phase of further exploration. However, the rewards remain high and stable overall, confirming the robustness and effectiveness of the learned policy. The use of Critic as a guide for Actor reduced the variance of updates and accelerated the learning process, highlighting the superiority of this approach over REINFORCE.

5. Proximal Policy Optimization (PPO)

Proximal Policy Optimisation (PPO) is an on-policy reinforcement learning algorithm, belonging to the family of policy gradient methods, which attempts to perform the largest possible update step on policy, using currently available data, while avoiding a performance collapse by limiting deviation from the current policy.

PPO exists in two main variants: PPO-Penalty and PPO-Clip. The last one is based on a clipped objective function, which removes the incentive for the new policy to deviate excessively from the old policy. The main advantage of PPO-Clip is that it avoids the use of explicit constraints, as opposed to PPO-Penalty, which applies a constrained update with respect to KL (Kullback-Leibler) divergence.

The gradient of the policy is estimated as:

$$\hat{g} = \mathbb{E}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{9}$$

where $\hat{A}_t$ is an estimate of the advantage function [8].

PPO introduces an objective function with a clipped term to avoid too drastic updates:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{10}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the ratio of probability of the new policy to the previous one, and ϵ is a hyperparameter controlling the magnitude of the update.

To stabilize training, the objective function also includes a term for the value function and an entropy bonus:

$$L_{\text{tot}}(\theta) = \hat{\mathbb{E}}_t \left[L^{CLIP}(\theta) - c_1 L_{\text{VF}}(\theta) + c_2 S[\pi_\theta] \right], \tag{11}$$

where L_{VF} represents the quadratic error between the estimated value function and the target, and S is the entropy term for incentive exploration [8]. For our experiments we don't consider the term of the entropy. We evaluated PPO in different environments trying to understand how the results changed depending on where the train and then the test were performed. We then considered the two environments:

- **Source**: environment where the dynamics are known and controllable.
- **Target**: environment with altered dynamics (e.g. modified torso masses).

To evaluate the robustness of the trained policies, three training and testing scenarios were defined:

- **Source** $\rightarrow$ **Source**: the policy is trained and tested in the same environment. This represents the ideal situation.
- **Source** $\rightarrow$ **Target**: the policy is trained in the source environment and tested in the target. It is a *lower bound* of robustness.
- **Target** $\rightarrow$ **Target**: the policy is trained and tested directly in the target environment, and represents the optimal performance.

5.1. Implementation and Optimization of hyperparameters

Unlike the REINFORCE and Actor-Critic algorithms implemented with PyTorch, we use the PPO agent from the stable-baselines3 library. This library simplifies the process: once the agent is created and linked to the training environment, it can be trained for a specified number of timesteps, with options to add callbacks for saving outputs or testing during training. The tuning of the hyperparameters is a crucial aspect in order to achieve good performance with PPO. Among the most important parameters there are the *clip range* ($epsilon$), the learning rate and the value of $gamma$ (discount factor). The *clip range*, as we have seen, is a limit to the variation of the policy during the update. The *learning rate* influences how fast the policy changes, too high a value may cause instability, too low may slow down learning. The value of $gamma$ determines how much the agent "values" future rewards over immediate rewards. In general, the optimization of these hyperparameters was performed by grid searches on discrete sets

of reasonable values, monitoring the average reward trend over test episodes to identify the best configurations. We obtained the following results:

- **clip_range**: 0.19877024509129543
- **learning_rate**: 0.0008
- **gamma**: 0.992

5.2. Results

We used 3 different seeds (10,20,40) to calculate the average rewards, thus obtaining more robust results. Each configuration was trained for 2400000 timesteps and tested every 20 train episodes (Figure 3).

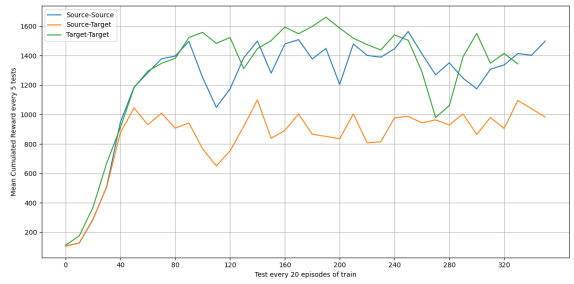

Figure 3. Reward trend in the three training and testing configurations.

We also did an experiment on a seed by making a complete train for the different environments and then tested it crosswise, as shown in Figure 4.

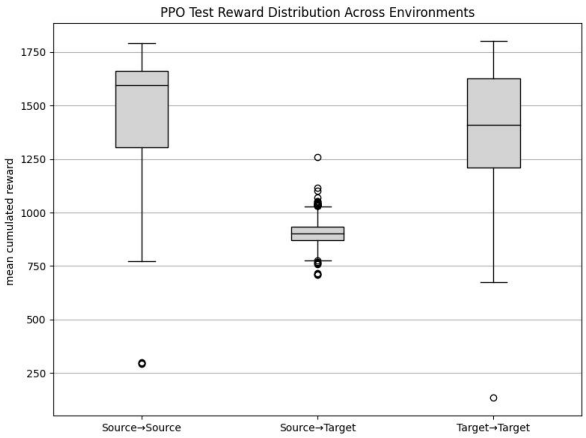

Figure 4. Test reward trend in the three configurations

As shown in the graphs, the configuration in which the agent is trained on the *Source* environment and then tested directly on the *Target* environment shows a clear reduction in performance compared to the configurations in which the agent is trained and tested on the same domain. This phenomenon can be traced back to a domain shift, i.e. a significant discrepancy between the distribution of data (observations, dynamics, reward) present during training and that encountered during execution in the target domain.

In contrast, the configurations **Target** $\rightarrow$ **Target** and **Source** $\rightarrow$ **Source** show high and consistent returns.

PPO proves to be an effective and stable algorithm for control in continuous environments, thanks to the introduction of the clipped function that avoids destructive updates while maintaining a simple structure. However, direct training on the target environment (i.e. the real world) is often impractical due to high costs, safety limitations, poor reproducibility and time required to collect data. Moreover, the real world can present noise and complexity that is difficult to model or control. For this reason, training in simulation (source domain) is resorted to, while trying to bridge the gap with the target environment.

6. Domain Randomization.

Domain Randomization (DR) is a powerful technique for improving the transferability of policies from simulation to real-world environments (sim2real). The key idea is to expose the agent to a wide range of dynamics by randomizing certain parameters of the simulator during training. This prevents the policy from overfitting to a single environment configuration and encourages robust behavior. Formally, we define the source domain $\mathcal{D}_S$ (e.g., the simulator) and the target domain $\mathcal{D}_T$ (e.g., the real world or a perturbed simulator). At each episode, a randomization configuration $\xi \in \Xi$ is sampled, and a trajectory $\tau \sim \mathcal{D}_S(\xi)$ is collected. The policy π_θ is optimized to maximize the expected return over the space of randomized configurations:

$$\max_\theta \ \mathbb{E}_{\xi\sim\Xi}\left[\mathbb{E}_{\tau\sim\mathcal{D}_S(\xi)}\left[R(\tau)\right]\right]. \tag{12}$$

Originally proposed in visual domains [9] [10], DR has been successfully extended to physical dynamics [11], such as mass, friction, damping, and joint limits. This allows policies to become robust to differences between simulation and the real world or to varying conditions within the simulation itself.

6.1. Uniform Domain Randomization.

In Uniform Domain Randomization (UDR), each randomized parameter is sampled uniformly from an interval $[p_i^{\min}, p_i^{\max}]$. In our MuJoCo Hopper implementation, we apply UDR by uniformly randomizing the masses of the thigh, leg, and foot links at the beginning of each episode, while keeping the torso mass fixed at 30% less than the target value to simulate an unmodeled shift. For example, with $\delta = 0.5$, the mass m_i of link i is sampled from $[0.5m_i, 1.5m_i]$ for $i \in \{2, 3, 4\}$. This forces the policy to learn robust behavior across a distribution of dynamics.

6.2. Implementation and Results.

We experimented with three values of δ: 0.2, 0.5, and 0.8, corresponding to increasing levels of variability. For each δ, we trained the policy using three different random seeds: 10, 20, and 40, to assess the stability and consistency of the method. Importantly, we used the same hyperparameters previously optimized for PPO in the source environment: clip range, learning rate, and gamma. This ensures a fair comparison and isolates the effect of domain randomization from other training variables.

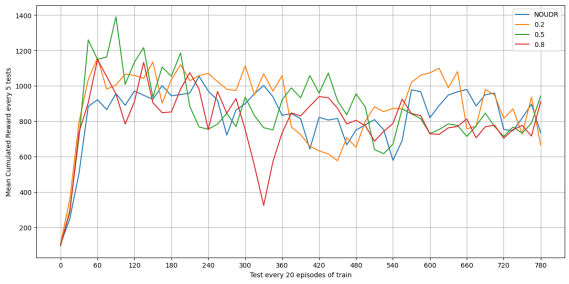

Figure 5. Comparison between different values of delta.

The results of the delta tuning are presented in Figure 5. We found that a delta value of 0.5 consistently achieved better performance than the other two settings across all testing episodes in the target environment. This suggests that an intermediate level of variability in the physical parameters strikes an effective balance, enhancing the agent's adaptability without significantly disrupting its learned behaviors. A lower delta of 0.2 produced more stable results but generally lower rewards, indicating limited generalization. In contrast, $\delta = 0.8$ led to both reduced performance and increased variability between runs, with some instances showing clear signs of policy collapse. Overall, UDR with delta value 0.5 consistently outperforms PPO when evaluated on the target domain.

6.3. Comparison Between UDR and PPO.

Compared to the baseline PPO agent, the UDR-trained agents consistently achieved better performance when evaluated on the target environment with the shifted torso mass. The UDR policy shows improved robustness, as it is less sensitive to specific dynamics and more adaptable to variations. This contrasts with PPO, which tends to overfit to the fixed source dynamics and fails to generalize under dynamic changes. However, UDR comes with tradeoffs: it requires careful manual tuning of the randomization ranges and introduces greater variability in training, which can slow down convergence and make learning less stable. Despite these limitations, UDR offers a powerful approach to building resilient agents for deployment in environments with uncertain or shifting dynamics.

7. DORAEMON

DORAEMON (DOmain RAndomization via Entropy MaximizatiON) is a domain randomization technique designed to improve the transfer of reinforcement learning policies from simulation to the real world. Traditional domain randomization introduces variability in the simulator to help the agent generalize, but choosing how much and what kind of randomness to add is often manual and inefficient.

DORAEMON automates this by gradually increasing the randomness (entropy) of the simulated environments. It does this while ensuring that the agent can still perform well, only expanding the range of variability when the agent's success rate is above a threshold. This adaptive exploration ensures that the agent learns to handle diverse environments without being overwhelmed by too much randomness too early.

7.1. Algorithm

To dynamically adapt the training distribution towards regions of the environment parameter space that lead to better generalization, we define a measure of success-weighted performance improvement across a set of recent interactions. In particular, let $T = \tau_1, \ldots, \tau_K$ denote the K most recent trajectories and let $E = e_1, \ldots, e_K$ be the corresponding set of environment parameters used to generate those trajectories. Each environment parameter vector $e_k \in \mathbb{R}^M$ encodes, for instance, the M link masses used in simulation, bounded component-wise by $(e_{\min}, e_{\max})$.

We define a success indicator function:

$$\texttt{Success}(\tau) = \begin{cases} 1 & \text{if } \texttt{Reward}(\tau) \geq \texttt{threshold} \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

Let ϕ denote the beta distribution over the (normalized) environment parameters. The corresponding probability density for a normalized parameter vector e_k under distribution ϕ is given by:

$$\nu_k(\phi) = \prod_{i=1}^{M} \frac{\phi.\texttt{pdf}\left(\frac{(e_k)i - \texttt{emin},i}{e_{\max,i} - e_{\min,i}}\right)}{e_{\max,i} - e_{\min,i}} \tag{14}$$

To evaluate how well a new distribution ϕ_{new} improves over an existing one ϕ_{old}, we define the following Success Rate Ratio function:

$$G_{ha}(\phi_{\text{old}}, \phi_{\text{new}}) = \frac{\sum_{k=1}^{K} \texttt{Success}(\tau_k) \cdot \frac{\nu_k(\phi_{\text{new}})}{\nu_k(\phi_{\text{old}})}}{\sum_{k=1}^{K} \frac{\nu_k(\phi_{\text{new}})}{\nu_k(\phi_{\text{old}})}} \tag{15}$$

G_{ha} measures the reweighted success rate of the new distribution ϕ_{new} using importance sampling over the observed trajectories and environment configurations. Intuitively, if ϕ_{new} assigns higher probability density to environment configurations that led to successful outcomes in

Algorithm 1 Doraemon Dynamic Distribution Update Algorithm

Require: actual distribution ϕ_{actual}, success-rate threshold α, freedom degree for masses δ, KL threshold ϵ, update distribution `step`.

$\phi_{\text{old}} \leftarrow \phi_{\text{actual}}$
Collect K trajectories T and K environment parameters E
Compute generalization metric: $G \leftarrow G_{ha}(\phi_{\text{old}}, \phi_{\text{old}})$
if $G < \alpha$ **then**
    Initialize candidate set: $\mathcal{C} \leftarrow \emptyset$
    Initialize second candidate set: $\mathcal{S}_c \leftarrow \emptyset$
    for $s \in \{-\texttt{step}, +\texttt{step}\}$ **do**
        for $i = 1$ to n **do** ▷ for each dimension of ϕ
            Generate new distribution: $\phi' \leftarrow \phi_{\text{old}}$
            $\phi'[i] \leftarrow \max(1.0, \phi_{\text{old}}[i] + s)$
            Compute divergence: $D \leftarrow KL(\phi_{\text{old}}, \phi')$
            Estimate success: $G' \leftarrow G_{ha}(\phi_{\text{old}}, \phi')$
            if $D < \epsilon$ **and** $G' \geq \alpha$ **then**
                Compute entropy: $H \leftarrow \mathcal{H}(\phi')$
                Add candidate: $\mathcal{C} \leftarrow \mathcal{C} \cup \{(\phi', H)\}$
            else if $D < \epsilon$ **and** $G' < \alpha$ **then**
                Add second candidate: $\mathcal{S}_c \leftarrow \mathcal{S}_c \cup \{(\phi', G')\}$
            end if
        end for
    end for
    if $\mathcal{C} \neq \emptyset$ **then**
        Select $\phi^* \in \mathcal{C}$ with maximum entropy
        Update distribution: $\phi_{\text{new}} \leftarrow \phi^*$
    else if $\mathcal{S}_c \neq \emptyset$ **then**
        Select $\phi^* \in \mathcal{S}_c$ with maximum success rate
        Update distribution: $\phi_{\text{new}} \leftarrow \phi^*$
    end if
end if

the past, then G_{ha} will be high. This principle underlies the distributional adaptation step in the DORAEMON algorithm (Algorithm 1), which iteratively seeks to increase entropy while preserving an optimal success rate.

7.2. Implementation and Results

We trained policies under three configurations: baseline PPO (no DR), UDR with fixed randomization magnitude ($\delta = 0.5$), and DORAEMON with both the parameters suggested in [12] and a set of parameters obtained via Optuna-based grid search. The parameters suggested in the reference paper are $\epsilon = 0.05$, $step = 2$, $\delta = 0.5$, $\alpha = 0.5$ while the set of parameters obtained with Optuna are $\epsilon = 0.09$, $step = 2.50$, $\delta = 0.18$, $\alpha = 0.54$.

The masses were sampled from a uniform distribution for UDR, and from a beta distribution for DORAEMON.

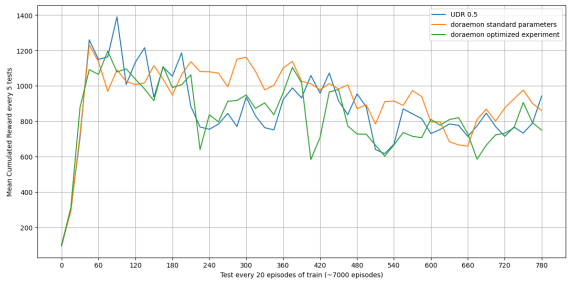

Figure 6. Comparison of UDR, DORAEMON with the parameters of the reference paper, and DORAEMON with parameters manually optimized.

All methods used the same PPO hyperparameters (clip range, learning rate, and γ) optimized for the source environment. For each method, three training runs were conducted using three different random seeds.

DORAEMON's performance is sensitive to its hyperparameters (e.g., $\epsilon, \delta, \alpha, step$). Over-aggressive expansion of the domain can lead to temporary instability in training, as observed in the Optuna configuration, which showed some sudden drops in test rewards (Figure 6). In contrast, the configuration from the paper maintained a strong and stable learning curve with the highest overall test-time reliability, and was therefore used for the following comparisons.

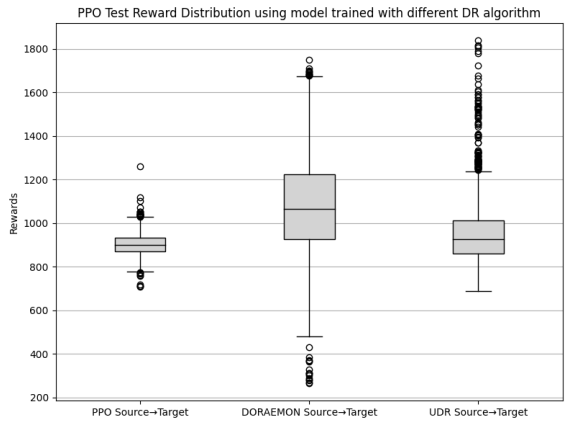

Figure 7. Distribution of episode rewards achieved in the target environment by policies trained on the source environment using PPO (mean=1082.89), UDR (mean=973.59), and DORAEMON (mean=904.37).

7.3. Comparison between PPO, UDR, and DORAEMON

Compared to both PPO and UDR, DORAEMON achieved the best overall performance when evaluated on the target environment with an unmodeled torso mass shift. While PPO without DR underperforms (due to its overfitting to the source dynamics) as it fails to adapt to changes

in the target dynamics, UDR improves robustness by exposing the policy to randomized dynamics during training. However, UDR uses fixed randomization ranges, which can lead the agent explore too much and struggle to learn, or explore too little and fail to adapt well.

DORAEMON addresses this challenge by gradually expanding the domain variation based on an entropy-based criterion. This results in higher mean test rewards in the target domain and higher variance (Figure 7), as a reflection of its broader generalization ability. UDR performance is lower than DORAEMON in most randomized domains, but for a few ones it is higher (Figure 7). These few values are not statistically significant and are therefore considered outliers. Its variance is also lower than that of DORAEMON. Finally, PPO has the lowest mean (and median) test reward and the lowest variance, with very few outliers.

8. Conclusions

When comparing the various methods, a clear progression emerges in the Hopper's jumping ability. The REINFORCE algorithm struggles to even maintain balance, producing erratic and negligible movement. With REINFORCE with baseline, the agent shows more stability and remains upright while slowly shifting to the right, but the behavior remains inadequate as the agent fails to execute the desired jumping motion. Transitioning to the actor–critic method introduces higher learning peaks, yet these are not sufficient for consistent jumping, underscoring the difficulty of achieving stable and reliable performance in such complex environments.

With PPO trained and tested in the source domain, the Hopper shows good performance, exhibiting significant improvements in jumping behavior. However, when this policy is transferred to the target domain, its performance deteriorates: while the agent initially manages to jump, it often loses balance and falls shortly after, highlighting the impact of domain mismatch. Introducing UDR with $\delta = 0.5$ leverages moderate variability to significantly improve adaptability, enabling smoother and more stable jumps.
The Doraemon method achieves the best performance overall, producing coherent, robust, and reactive jumping behavior that generalizes well to the target environment.

While foundational algorithms like REINFORCE and Actor-Critic established the basis for policy gradient methods, they are often outperformed by more robust approaches such as Proximal Policy Optimization. To enhance generalization in simulation, domain randomization has proven effective. Uniform Domain Randomization increases environmental variability to improve adaptability, while our extension, Doraemon, extends this idea with a more structured strategy that yields better performance and stability.

Despite the benefits, these techniques involve inherent stochasticity, requiring multiple training runs and careful hyperparameter tuning to achieve consistent results. Though computationally demanding, this process is essential for developing reliable robotic behaviors.

References

[1] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, 1994. 1

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018. 1, 2

[3] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. 1

[4] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3–4, pp. 279–292, 1992. 1

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

[6] P. Kormushev, S. Calinon, and D. G. Caldwell, "Reinforcement learning in robotics: Applications and real-world challenges," *Robotics and Autonomous Systems*, vol. 61, no. 3, pp. 273–284, 2013. 2

[7] R. S. Sutton and A. G. Barto, "Actor-Critic Methods," in *Reinforcement Learning: An Introduction*, 2nd ed., Cambridge, MA: MIT Press, 2018, ch. 13, pp. 271–278. 3

[8] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017).*Proximal Policy Optimization Algorithms*. 4

[9] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world,"Canada, 2017, pp. 23–30. 5

[10] F. Sadeghi and S. Levine, "(CAD)2RL: Real single-image flight without a single real image," in *Proceedings of Robotics: Science and Systems (RSS)*, Cambridge, MA, USA, 2017. 5

[11] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization," Australia, 2018, pp. 3803–3810. 5

[12] Gabriele Tiboni, Pascal Klink, Jan Peters, Tatiana Tommasi, Carlo D'Eramo, and Georgia Chalvatzaki, *Domain Randomization via Entropy Maximization*, arXiv preprint arXiv:2311.01885, 2023. 7