Reinforcement Learning for Sim-to-Real Transfer

Abstract

This report dives into the use of Reinforcement Learning (RL) for robotic systems, with a focus on solving the challenging task of transferring learned behaviors from simulation to the real world, specifically in the Hopper environment. We test and compare a range of RL algorithms, such as REINFORCE and Actor-Critic, alongside more advanced methods like Proximal Policy Optimization (PPO). One of the key techniques explored here is Uniform Domain Randomization (UDR), which helps make policies more adaptable by introducing a variety of environmental changes during training. Our findings show that UDR significantly improves the ability of policies to perform well in new settings. Overall, this work emphasizes the importance of domain adaptation to ensure that simulated training can successfully translate into real-world performance.]

1 Introduction

In this section we describe the main theoretical concepts of the Reinforcement Learning paradigm, along with an introduction of the Sim-to-Real transfer problem in the context of robotics. The code of the implementation of the various algorithms can be found at the project repository.

Reinforcement Learning Framework

The Reinforcement Learning (RL) framework is centered on the interaction between an agent and its environment, where learning occurs through continuous experience. At each time step, the agent receives an observation that characterizes the current state of the environment and selects an action based on a policy. In response, the environment transitions to a new state and generates a scalar reward signal that reflects the quality of the action taken. The objective of the agent is to learn a policy that maximizes the expected cumulative reward over time, effectively guiding the agent's decision-making process through trial-and-error interactions with the environment.

2 Markov Decision Process and Policy Gradient Methods

The Reinforcement Learning problem can be formally modeled as a Markov Decision Process (MDP), which is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$. Here, $\mathcal{S}$ denotes the set of possible states, $\mathcal{A}$ the set of possible actions, R the reward function, and $\mathcal{P}$ the state transition probability distribution that governs the stochastic evolution of the environment. The MDP dynamics describe the probability of transitioning to a new state and receiving a reward, given the current state and action.

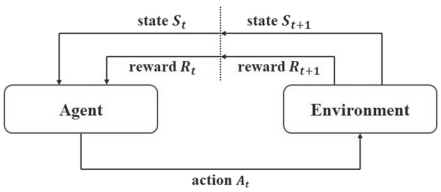

Figure 1: The agent–environment interaction in Markov Decision Process.

$$p(s', r \mid s, a) = \Pr\{S_{t+1} = s',\ R_{t+1} = r \mid S_t = s, A_t = a\} \tag{1}$$

Since the agent has no prior knowledge of the underlying probability distribution $\mathcal{P}$, it can only interact with the environment by observing realizations of state transitions and rewards. The agent's experience can be represented as a trajectory τ, defined as a finite sequence of states, actions, and rewards of the form $\tau = \{S_0, A_0, R_0, S_1, A_1, R_1, \ldots, S_T\}$. When a trajectory reaches a terminal state, it is referred to as an episode. For each trajectory, the objective is to compute the discounted return, which represents the total accumulated reward, discounted over time, and is defined as:

$$G_t = \sum_{k=0}^{T} \gamma^k R_{t+k} \tag{2}$$

where $\gamma \in [0, 1]$ is the discount factor that determines the present value of future rewards. Actions are selected according to a parameterized stochastic policy $\pi_\theta(a \mid s)$, which defines a probability distribution over actions given the current state. The state-value function $v^\pi(s)$ under policy π is defined as the expected return when starting from state s and following policy π. The overall performance of the policy is characterized by the objective function $J(\theta)$, representing the expected return when starting from an initial state s_0:

$$J(\theta) = v^\pi(s_0). \tag{3}$$

The goal of policy-gradient methods is to optimize the policy parameters θ by maximizing $J(\theta)$. This is typically achieved by performing gradient ascent updates:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta), \tag{4}$$

where α is the learning rate. However, estimating the policy gradient $\nabla_\theta J(\theta)$ is non-trivial due to its dependence on the unknown state distribution, which is influenced by the policy itself. The Policy Gradient Theorem provides an analytical expression for the gradient that avoids explicit differentiation of the state distribution:

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) G_t\right], \quad (5)$$

where G_t denotes the discounted return. This formulation enables the direct estimation of the gradient using samples generated from interaction with the environment.

2.1 Reinforcement Learning for Robotics and Sim-to-Real Transfer

Reinforcement Learning (RL) provides a powerful framework for enabling robots to autonomously learn complex behaviors through trial-and-error interactions with their environment. However, applying RL directly on physical robotic platforms is often impractical due to the extensive data requirements, high computational cost, potential safety risks, and wear on hardware associated with long training procedures. A common solution to mitigate these challenges is to perform training in simulated environments, where an approximate model of the real-world dynamics is available. Nevertheless, policies trained solely in simulation often fail to generalize to the real world due to the discrepancies between the simulated dynamics and actual physical systems. This discrepancy, known as the *reality gap*, limits the direct applicability of simulation-trained policies in real-world scenarios. The process of transferring knowledge from simulation to the real environment is referred to as *Sim-to-Real transfer*. The primary objective of Sim-to-Real transfer is to develop policies that are robust to variations and uncertainties encountered in the real world, even when such variations were not explicitly observed during training.

In this context, Domain Randomization (DR) has emerged as an effective strategy to improve policy generalization. DR enhances the robustness of policies by exposing the agent to a wide range of randomized simulated environments, systematically varying key physical and environmental parameters during training. This forces the policy to adapt to a diverse set of scenarios, increasing its likelihood of performing well under previously unseen real-world conditions. In this work, we explore two variants of Domain Randomization: Uniform Domain Randomization (UDR), where parameters are sampled uniformly within predefined bounds, and Automatic Domain Randomization (ADR), which adaptively expands the parameter ranges based on the agent's learning progress.

For practical reasons, we focus on a *Sim-to-Sim transfer* setting, where policies are transferred from a source simulated environment to a target simulated environment with altered dynamics. This allows for controlled evaluation of transfer performance without requiring real hardware deployment.

2.2 Experimental Setup: Hopper Environment

The simulated environment used for training and evaluation is the Hopper environment provided by the OpenAI Gym API [**?**]. The Hopper consists of a planar one-legged robot composed of four main body parts: torso, thigh, leg, and foot, as illustrated in Fig. 2. The control objective is to learn a locomotion policy that enables the robot to perform stable and efficient forward hopping while avoiding falls. The robot's motion is generated by applying continuous torques to the three actuated joints (hip, knee, and ankle), which connect the corresponding body segments. The episode terminates when the robot either falls or reaches a physically unstable configuration, at which point it is considered to have failed; otherwise, it is regarded as alive.

The source and target environments are defined by varying the physical properties of the robot's components to evaluate the policy's generalization capability. Specifically:

- In the *source environment*, the masses of the individual body parts are set as follows: 2.53 kg (torso), 3.93 kg (thigh), 2.71 kg (leg), and 5.09 kg (foot).
- In the *target environment*, only the torso mass is modified to 3.53 kg, while all other parameters remain unchanged.

The observation (state) space includes the set of features perceived by the agent at each time step, which typically consists of body joint angles, angular velocities, linear positions, and velocities. The action space is represented by the joint torques applied at the actuated joints.

The reward function provided to the agent is composed of three terms:

- **Alive Bonus**: a constant reward of +1 for each time step the agent remains in a valid configuration.
- **Forward Progress Reward**: proportional to the forward displacement, calculated as $(x_{t+1} - x_t)/\Delta t$, where Δt denotes the action duration.
- **Control Cost**: a penalty term that discourages excessive torque magnitudes, promoting smoother and more energy-efficient movements.

This reward structure encourages the agent to achieve fast and stable locomotion while penalizing inefficient control strategies.

3 Reinforcement Learning Algorithms

3.1 REINFORCE

The REINFORCE algorithm is a Monte Carlo–based policy gradient method that uses the Policy Gradient Theo-

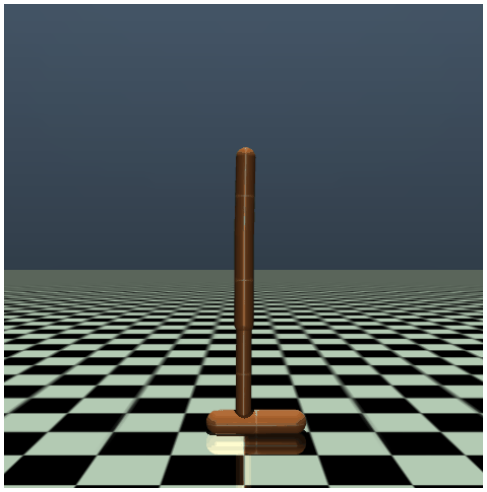

Figure 2: The Hopper robot used for training and evaluation in simulation.

rem (Equation 5) to optimize stochastic policies by estimating unbiased gradients of the expected return (Equation 3). The policy is modeled as a feed-forward neural network with *tanh* activations. The algorithm collects full episodes, computes returns at each time-step (Equation 2), and updates the policy parameters via stochastic gradient ascent (Equation 4), applying the gradient estimate from Equation 5.

Algorithm REINFORCE

Input A differentiable policy parameterization $\pi_{\boldsymbol{\theta}}(A|S)$
Algorithm parameter: Step size $\alpha > 0$
Initialize The policy parameters $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ at random.
for Loop forever (for each episode): **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi_{\boldsymbol{\theta}}$
  $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \sum \gamma^t G_t \nabla \ln \pi_{\boldsymbol{\theta}}(A_t|S_t)$
end for

Figure 3: The REINFORCE algorithm structure.

The analysis of the *rolling_reward_mean_100* plot reveals distinct performance characteristics between the REINFORCE and REINFORCE-Baseline algorithms over 10,000 training episodes. Both reward curves are normalized, ensuring a direct and equitable comparison of their learning progress. While the vanilla REINFORCE algorithm demonstrated a higher peak reward around 5500 steps (approximately 350), it also exhibited significantly higher variance and experienced a considerable degradation in performance towards the end of the training, concluding at a normalized reward of about 160. In contrast, REINFORCE-Baseline, characterized by its smoother learning curve, indicative of reduced variance due to the baseline,(a fixed constant value of 20 was selected based on the average early returns) showed more stable and sustained performance. Although its peak reward was lower (around 280), REINFORCE-Baseline maintained a higher and more consistent normalized reward in the latter stages of training, finishing at approximately 280. This suggests that while vanilla REINFORCE might occasionally achieve superior instantaneous performance, the addition of a baseline in REINFORCE-Baseline contributes to a more stable training process and better long-term performance and convergence within the observed training budget. The results are illustrated in Figure 4

Figure 4: Train Reward for REINFORCE with baseline and without baseline.

Approach	Training Time
No baseline	2230.45 s $\sim$ 37.17 minutes
Baseline = 20	2807.77 s $\sim$ 46.80 minutes

Table 1: Comparison of training times for REINFORCE with and without baseline.

As shown in Table 1, the training time for the REINFORCE algorithm with a baseline (Baseline = 20) is slightly higher than the version without a baseline. This difference arises because, even though the baseline is constant and not updated during training, its inclusion still introduces extra operations in each step of the policy update process, which slightly increases the overall computation time.

3.2 Actor-Critic

Actor-Critic methods combine the benefits of both value-based and policy gradient approaches by maintaining two separate structures: an **actor**, which is responsible for selecting actions, and a **critic**, which evaluates the actions by estimating the value function. These methods utilize the state-value function v^{π}, which maps states to expected returns under policy π:

$$v^{\pi}(s_t) = E_{\pi}\left[G_t \mid s_t\right] \tag{6}$$

Algorithm ACTOR-CRITIC

Input A differentiable policy parameterization $\pi_{\boldsymbol{\theta}}(A|S)$
Input A differentiable state-value function $\hat{V}_{\boldsymbol{w}}(S)$
Parameters: Step sizes $\alpha^{\boldsymbol{\theta}} > 0, \alpha^{\boldsymbol{w}} > 0$
Initialize The policy parameters $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\boldsymbol{w} \in \mathbb{R}^{d}$
for Loop forever (for each episode): **do**
  Initialize S_0 (first state of episode)
  $I \leftarrow 1$
  while S_t is not terminal (for each time step t) **do**
    $A \sim \pi_{\boldsymbol{\theta}}(\cdot|S)$
    Take action A, observe S_{t+1}, R
    $\delta \leftarrow R + \gamma \hat{V}_{\boldsymbol{w}}(S_{t+1}) - \hat{V}_{\boldsymbol{w}}(S_t)$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} \alpha^{\boldsymbol{w}} \delta \nabla \hat{V}_{\boldsymbol{w}}(S_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi_{\boldsymbol{\theta}}(A|S_t)$
    $I \leftarrow \gamma I$
    $S_t \leftarrow S_{t+1}$
  end while
end for

Figure 5: Actor-Critic algorithm

The approximation of the state-value function enables a *bootstrapped estimation* of the one-step return:

$$G_{t:t+1} = r_t + \gamma v(s_{t+1})$$

Although this approach introduces some bias in the gradient estimate $\nabla_\theta J(\theta)$, it significantly reduces the variance during training. Instead of waiting for the full trajectory τ, bootstrapping allows updates based on shorter horizons.

We implemented the Actor-Critic method by using a separate neural network—architecturally identical to the policy network—to estimate the state-value function $V(s)$. This forms the critic component of the method, while the policy network serves as the actor, responsible for updating the policy in the direction suggested by the critic. The actor uses the same loss function as REINFORCE (i.e., $-J(\theta)$), while the critic is trained using the mean squared error (MSE) between its predicted value and the bootstrapped one-step return. The algorithm was trained for 10,000 episodes on the same source environment using identical hyperparameters (learning rate and γ) as REINFORCE. Experimental results (see Fig. 6) demonstrate that the Actor-Critic approach significantly outperforms both REINFORCE and REINFORCE with baseline. While REINFORCE suffers from high variance and unstable convergence, and the baseline version only modestly reduces this variance, Actor-Critic achieves both higher reward performance and more stable learning. The moving average reward curve for Actor-Critic steadily rises to approximately 500, indicating effective exploration and fast convergence, whereas REINFORCE variants plateau at lower performance levels (below 300). This confirms the crucial role of the critic in reducing variance and guiding the policy updates more effectively.

Table 2: Total Cumulative Time for Actor-Critic

Method	Seconds	Minutes	
Actor-Critic	2319.44	38.66	

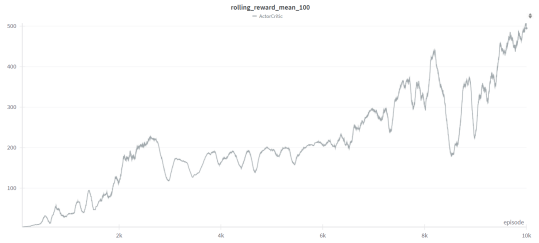

Figure 6: Training Reward for Advantage Actor critic.

3.3 Proximal Policy Optimization

3.3.1 Description

Proximal Policy Optimization (PPO) is a widely used on-policy reinforcement learning algorithm that stabilizes policy updates by limiting how much the policy is allowed to change. Among its variants, we adopted **PPO-Clip**, implemented using the *Stable-Baselines3 (SB3)* library.

Unlike PPO-Penalty, which uses a Kullback-Leibler divergence penalty, PPO-Clip avoids explicit regularization and instead relies on **clipping** the probability ratio in the objective function to stay within a small range. This clipping mechanism helps prevent destructive policy updates.

The objective function used is:

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \right.$$
$$\left. \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s, a) \right)$$

This loss is the minimum between:

- **Unclipped term:** the original advantage multiplied by the probability ratio.
- **Clipped term:** the advantage multiplied by the clipped ratio to prevent too large updates.

By using this clipped surrogate objective, PPO-Clip balances effective learning with training stability, avoiding the sensitivity of large updates and ensuring smooth convergence.

3.3.2 Tuning and Training

We selected a set of hyperparameters that we considered most relevant for the PPO algorithm and tuned them using a uniform sampling strategy within specified intervals. Table **??** shows the selected hyperparameters and their corresponding tuning ranges. We performed ten independent training runs, each lasting 3 million time steps, on both the source and target environments to determine the best configuration showed in Tab. 4.

Algorithm	Parameter	Values
PPO	learning_rate	{min: 5×10^{-4}, max: 1×10^{-3}}
	clip_range	{min: 0.25, max: 0.35}
	entropy_coefficient	{min: 0.005, max: 0.02}
	discount factor γ	{min: 0.97, max: 0.99}
	GAE lambda λ	{min: 0.95, max: 0.99}
	n_steps, batch_size	{2048, 4096}, {64, 128}

Table 3: Hyperparameters of Proximal Policy Optimization

Table 4: Best hyperparameters of Proximal Policy Optimization in source and target environments

Environment	Parameter	Value
Source	learning_rate	0.0007081
	clip_range	0.2679
	entropy_coefficient	0.01776
	γ	0.9871
	λ	0.9779
	n_steps	2048
	batch_size	128
Target	learning_rate	0.0008411
	clip_range	0.3329
	entropy_coefficient	0.01748
	γ	0.9865
	λ	0.9694
	n_steps	2048
	batch_size	128

Figure 7 shows the training of the two best models obtained for the source and target environments. We can make some considerations about the behaviour of the two curves:

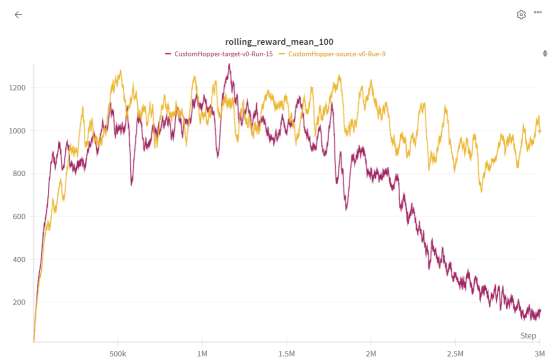

Figure 7: Training curves of the best PPO models on source and target environments

Both the Source and Target models start with a strong learning phase, reaching high rewards. However, the Source model continues to improve and remains stable throughout training, while the Target model begins to deteriorate after around 2 million steps.

This sharp decline in the Target curve is likely due to overfitting or instability from policy updates—common in on-policy methods like PPO when the environment is more sensitive or has higher variability.

3.3.3 Results

As shown in Figure 8 and summarized in Table 5, the model trained in the source environment (s→s) achieves a high average reward of 1731.82**(upper bound)**, indicating successful learning in its native domain. However, when the same model is evaluated in the target environment without any adaptation (s→t), its performance drops drastically to 756.93 **(lower bound)**, which reflects a significant reward reduction of approximately 975 points. This substantial performance gap provides clear evidence of a domain shift between the source and target environments. On the other hand, the model trained and tested within the target environment (t→t) achieves an average reward of 1649.66, close to the source-to-source baseline. This suggests that while the model is capable of stable learning in a consistent environment, it struggles to generalize across domains. Therefore, these findings strongly highlight the importance of applying domain adaptation techniques to bridge the performance gap and improve cross-environment generalization.The lower bound represents a naive approach to the Sim-to-Real problem, whereas the upper bound is the maximum performance achievable by PPO in the target environment."

Figure 8: Evaluation of the two best models in different configurations.

Scenario	Mean Reward	Standard Deviation
Source → Source (s→s)	1731.82	± 103.26
Source → Target (s→t)	756.93	± 84.23
Target → Target (t→t)	1649.66	± 97.24

Table 5: Final Evaluation Results of Different Training and Testing Scenarios

4 Uniform Domain Randomization

4.1 Description

The ability to generalize is essential for enabling agents to handle real-world situations that can change in unexpected ways. A major difficulty lies in the mismatch between training data (source environment) and test data (target environment), caused by differences in environmental settings, parameters, and other factors. To overcome this issue, Domain Randomization (DR) can be applied during training by introducing random changes to environment parameters, thereby enhancing data diversity. However, to be effective, DR must carefully select which parameters to vary and ensure these variations are uniformly distributed.

4.2 Implementation

We exploit *Uniform Domain Randomization* (UDR), based on a uniform distribution that is used to vary the parameters during training. This approach aims to achieve a better coverage of the state space, allowing trained agents to generalise more effectively to new scenarios. Specifically, to approach correctly the source → target scenario, we have taken the optimal hyperparameters given by the tuning of PPO on the source environment in Tab. 4 and searched for the optimal intervals for randomizing the masses of some elements of the agent; after that, we tested the policy on the target environment. Since the two environments differ only for the mass of the *torso*, the parameters to be randomized will be the masses of the remaining parts: *thigh*, *leg* and *foot*. Uniform Domain Randomization associates each body part mass to a uniform distribution $U(a_i, b_i)$, from which their values are sampled at each episode, as shown in the algorithm in Fig. 9.

Algorithm UDR

Require: $\{m_i\}_{i=1}^{d}$ {Parameters to be randomized}
Require: $\{(a_i, b_i)\}_{i=1}^{d}$ {Distribution bounds}
  repeat
    for $i \in d$ **do**
      $m_i \sim U(a_i, b_i)$
    end for
    Generate Data (m_i)
    Update policy
  until Training is complete

Figure 9: UDR Algorithm.

Figure 10: UDR Tuning on Source Environment

fig. 10 indicates that the agent experiences a rapid improvement in performance during the early phase of training, attaining average rewards in the range of 800 to 1200 within the first one million steps. However, after approximately 1.5 million steps, a significant decline in performance is observed. One possible reason for this degradation is that the agent continues to sample body part masses from unchanged distributions, leading to repeated exposure to the same configurations which, as the time steps progress, do not provide sufficient stimulus for learning. As a result, the agent repeatedly encounters familiar scenarios that no longer contribute effectively to the learning process. Another potential issue is that, after many iterations, the agent might overfit to this broad yet fixed distribution of environments, ultimately reducing its ability to generalize to the specific target environment.

4.3 Results

We tested on the target environment the obtained policy, in order to be able to compare it with the upper bound and lower bound previously showed in Fig. 8. As expected the policy with the best results actually performed really close to the optimal target → target policy of the PPO and UDR policy fall within the bounds. This means that UDR is able to over come the shift in the target environment represented by the different value of the mass of the torso.

Figure 11: Smoothed reward vs episode for UDR training

Setting	Mean Reward	Std Deviation
Source → Target (UDR)	1620.1	370.3

Table 6: Evaluation source-to-target

5 Conclusion

In this report, we explored several Reinforcement Learning algorithms and observed their limited robustness when adapting to slightly different environments. To tackle the Sim-to-Real problem, we employed Domain Randomization techniques, specifically implementing Uniform Domain Randomization (UDR) and Automatic Domain Randomization (ADR) and We obtained promising results using the proposed approach. .

References

[1] Hossein Ahmadzadeh, Fatemeh saleh. *RL-PoliTO-MLDL-Project*. GitHub repository, https://github.com/hossein-ahmadzadeh/RL-PoliTO-MLDL-Project

[2] Lilian Weng. *Policy Gradient Algorithms*. Blog post, April 8, 2018. https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

[3] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. *Stable-Baselines3: Reliable reinforcement learning implementations*. Journal of Machine Learning Research, 22(268):1–8, 2021. http://jmlr.org/papers/v22/20-1364.html

[4] OpenAI et al. Solving rubik's cube with a robot hand. CoRR, abs/1910.07113, 2019. arXiv:1910.07113.http://arxiv.org/abs/1910.07113