Domain Randomization in Robotic Control

Francesco Giuseppe Gillio
Department of Computer Science
Polytechnic University of Turin

March 26, 2024

Abstract

This research aims to explore reinforcement learning (RL) agents within the Gym Hopper environment with the MuJoCo physics engine for accurate modeling. The Hopper, a one-legged robot, must learn and master hopping and maintaining balance while optimizing horizontal speed. Nowadays, the effective application of reinforcement learning to robotics represents a significant challenge, mainly due to the complexities of learning in real-world environments and accurately simulating physics. To address these challenges, this study investigates a range of policy gradient algorithms, including REINFORCE, Actor-Critic, and Proximal Policy Optimization (PPO), assessing their strengths and suitability for training agents in simulation environments. Furthermore, the project delves into state-of-the-art techniques for Domain Randomization, in particular Domain Randomization Optimization IDentification (DROID), which enhances the robustness and transferability of learned policies across various conditions. Through the application of these methodologies, this research aims to mitigate the "reality gap" often found in robotic applications, thereby improving agent performance and adaptability. This research not only contributes to the understanding of effective RL strategies in robotics but also lays the groundwork for future explorations into the refinement of domain randomization techniques, pushing the boundaries of practical and reliable robotic systems in real-world scenarios.

GitHub: `https://github.com/305909/reino`

1 Introduction

In reinforcement learning (RL), the primary entities consist of the agent and the environment. The environment encompasses the world within which the agent resides and interacts [1]. Over each interaction step, the agent receives a (potentially partial) observation of the world's state and subsequently determines an appropriate action. The agent's actions induce changes in the environment, although the environment may also evolve independently. Additionally, the agent receives a reward signal from the environment, which quantifies the desirability of the current world state. The agent's objective involves maximizing its cumulative reward, known as the return. Reinforcement learning methodologies enhance the agent's acquisition of behaviors that aim to achieve this objective.

States and Observations

A state s offers a comprehensive description of the world's condition, containing no hidden information. In contrast, an observation o provides a partial depiction of a state, potentially omitting certain details. In deep reinforcement learning (RL), states and observations often appear as real-valued vectors, matrices, or higher-order tensors. For instance, a robot might portray its state through its joint angles and velocities.

Action Space

Environments within reinforcement learning (RL) accommodate diverse action types, collectively known as the action space. For example, games such as Atari and Go offer discrete action spaces, where agents select from a finite set of moves. In contrast, environments involving physical robotic control feature continuous action spaces, e.g. real-valued vectors.

Policy

In reinforcement learning (RL), a policy serves as the guiding principle for an agent's decision-making process. When deterministic (μ), the policy dictates actions as direct functions of the state:

$$a_t = \mu(s_t)$$

Alternatively, when stochastic (π), the policy samples actions according to a probability distribution conditioned on the state:

$$a_t \sim \pi(\cdot|s_t)$$

Since the policy essentially embodies the agent's decision-making mechanism, it's common to interchangeably refer to "policy" as "agent", such as stating "The policy aims to maximize reward". Deep RL deals with parameterized policies: functions whose outputs vary according to adjustable parameters (such as neural network weights and biases). The framework often denote the parameters of such a policy by θ or ϕ, and then write this as a subscript on the policy symbol to highlight the connection:

$$a_t = \mu_\theta(s_t)$$

$$a_t \sim \pi_\theta(\cdot|s_t)$$

Trajectories

A trajectory τ comprises a sequence of states and actions in the world: $\tau = (s_0, a_0, s_1, a_1, ...)$. The very first state of the world, s_0, comes from a random draw from the start-state distribution, sometimes denoted by ρ_0: $s_0 \sim \rho_0(\cdot)$. State transitions (what happens to the world between the state s_t and the subsequent state s_{t+1}), follow the natural laws of the environment and hinge solely on the most recent action a_t. These transitions may occur in either a deterministic, $s_{t+1} = f(s_t, a_t)$, or stochastic, $s_{t+1} \sim P(\cdot|s_t, a_t)$. Actions come from an agent according to its policy.

Reward and Return

The reward function R plays a pivotal role in reinforcement learning, depending on the current state of the world s_t, the action taken a_t, and the subsequent state of the world s_{t+1}: $r_t = R(s_t, a_t, s_{t+1})$. The agent attempts to maximize some notion of cumulative reward over a trajectory (the specific interpretation depends on context):

- finite-horizon undiscounted return:

$$R(\tau) = \sum_{t=0}^{T} r_t$$

- infinite-horizon discounted return:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

The discount factor $\gamma \in (0, 1)$ in reinforcement learning serves two key purposes. Intuitively, it reflects the preference for immediate rewards over delayed rewards. Mathematically, it ensures that the sum of rewards over an infinite horizon converges to a finite value under reasonable conditions, enhancing practical computation and decision-making in RL algorithms.

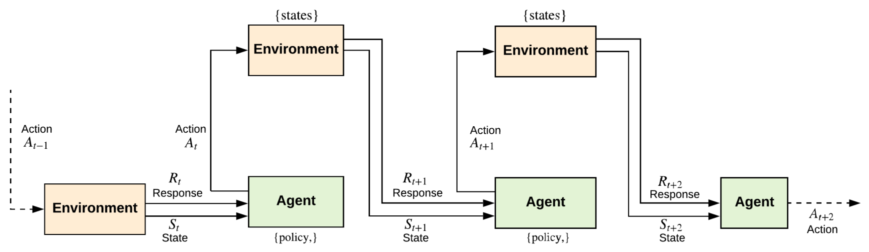

Figure 1: The agent–environment interaction [1].

Moreover, the return assumes significance not only in a global perspective of the trajectory τ, but also in a local context. In particular, at each time step t, the (discounted) return R_t emerges as the cumulative sum of rewards from t until the conclusion of the trajectory:

$$R_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$$

Major textbooks denote the (discounted) return at time step t interchangeably as R_t or G_t.

The RL Problem

Whatever the choice of return measure (whether infinite-horizon discounted, or finite-horizon undiscounted), and whatever the choice of policy, the RL problem involves the selection of a policy which maximizes expected return when the agent acts according to it. The notion of expected return requires some notion about probability distributions over trajectories:

- probability of a T-step trajectory:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$$

- the expected return (for whichever measure):

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)]$$

- the central optimization problem in RL:

$$\pi^* = \arg\max_{\pi} J(\pi)$$

  with π^* referencing to the optimal policy.

Value Functions

It's often useful to determine the value of a state or state-action pair. Value, in this context, refers to the expected return when starting from that state or state-action pair and following a specific policy indefinitely thereafter. Value functions play a critical role in nearly every reinforcement learning (RL) algorithm. The On-Policy Value Function, $V^\pi(s)$ returns the expected return when starting in state s and consistently following policy π:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s]$$

The On-Policy Action-Value Function, $Q^{\pi}(s,a)$ returns the expected return when starting in state s, taking an arbitrary action a (which may not come from the policy), and then forever after act according to policy π:

$$Q^{\pi}(s,a) = \underset{\tau\sim\pi}{\mathbb{E}}[R(\tau)|s_0 = s, a_0 = a]$$

The Optimal Value Function, $V^*(s)$ returns the expected return when starting in state s and always act according to the optimal policy in the environment:

$$V^*(s) = \max_{\pi} \underset{\tau\sim\pi}{\mathbb{E}}[R(\tau)|s_0 = s]$$

The Optimal Action-Value Function, $Q^*(s,a)$ returns the expected return when starting in state s, taking an arbitrary action a, and then forever after act according to the optimal policy in the environment:

$$Q^*(s,a) = \max_{\pi} \underset{\tau\sim\pi}{\mathbb{E}}[R(\tau)|s_0 = s, a_0 = a]$$

Advantage Functions

In reinforcement learning, sometimes the focus shifts from absolute action quality to comparing actions in terms of relative effectiveness, precisely captured by the advantage function. The advantage function $A^{\pi}(s,a)$ corresponding to a policy π quantifies the improvement achieved by selecting a specific action a in state s, over randomly selecting an action according to $\pi(\cdot|s)$, assuming the agent to act according to π forever after. Mathematically:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

The Optimal Q-Function and the Optimal Action

There exists a crucial link between the optimal action-value function $Q^*(s,a)$ and the action chosen by the optimal policy. By definition, $Q^*(s,a)$ provides the expected return for starting in state s, taking (arbitrary) action a, and then acting according to the optimal policy forever after. The optimal policy in state s selects the action that maximizes the expected return. Therefore, from $Q^*(s,a)$, the agent directly determine the optimal action $a^*(s)$, via:

$$a^*(s) = \arg\max_{a} Q^*(s,a)$$

Model-Free and Model-Based RL

One critical branching point in a reinforcement learning (RL) algorithm involves determining whether the agent possesses (or learns) a model of the environment. A model of the environment refers to a function that predicts state transitions and rewards. The primary advantage of possessing a model lies in its ability to enable the agent to plan by anticipating outcomes across a range of possible choices, thereby facilitating explicit decision-making. Agents may then condense the results of this forward-planning into a learned policy. AlphaZero [2] represents a notable example of this approach, showcasing significant improvements in sample efficiency over methods that lack a model. However, a significant disadvantage arises from the fact that agents typically lack access to a ground-truth model of the environment. Consequently, if an agent intends to utilize a model, it must learn the model solely from experience, which introduces several challenges. The most prominent challenge concerns the potential exploitation of bias in the model by the agent, leading to optimal performance with respect to the learned model but suboptimal (or severely deficient) behavior in the actual environment. Model learning inherently presents substantial difficulty, and even considerable investment of time and computational resources may not yield successful outcomes. Although model-free methods sacrifice the potential improvements in sample efficiency provided by a model, they typically offer easier implementation and tuning. As of September 2018, model-free methods enjoy greater popularity and more extensive development compared to model-based methods.

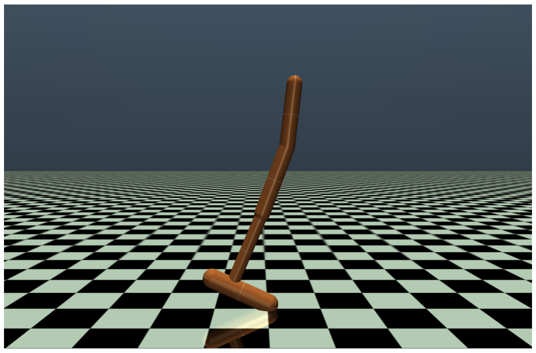

Figure 2: The Hopper environment [3].

What to Learn

Another critical decision point in a reinforcement learning (RL) algorithm concerns the question of what to learn. The typical elements to consider include: policies, either stochastic or deterministic, action-value functions (Q-functions), value functions, and/or environment models.

What to Learn in Model-Free RL

Two main approaches exist for representing and training agents using model-free RL: Policy Optimization and Q-Learning. Policy Optimization methods explicitly represent a policy as $\pi_\theta(a|s)$. They optimize the parameters θ either directly by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$. This optimization almost always occurs on-policy, which means that each update only uses data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximator $V_\phi(s)$ for the on-policy value function $V^\pi(s)$, which assists in figuring out how to update the policy. Examples of policy optimization methods include: A2C, which performs gradient ascent to directly maximize performance, and PPO, which updates the policy by maximizing a surrogate objective function that conservatively estimates the impact of the update on $J(\pi_\theta)$, thereby indirectly maximizing performance. Q-Learning methods focus on learning an approximator $Q_\theta(s,a)$ for the optimal action-value function, $Q^*(s,a)$. They typically employ an objective function grounded in the Bellman equation. This optimization generally occurs off-policy, enabling each update to utilize data collected at any point during training, regardless of the agent's exploration strategy when obtaining the data. The corresponding policy derives from the relationship between Q^* and π^*, where the Q-learning agent selects actions according to:

$$a(s) = \arg\max_a Q_\theta(s,a)$$

2 Environment

The Hopper of MuJoCo [3], a two-dimensional figure with one leg, comprises four primary body parts: a top torso, a middle thigh, a bottom leg, and a single foot supporting the entire body. The objective involves generating forward (rightward) movement through torque application at the three hinges connecting these body segments. This research implements two customized versions of the Gym Hopper environment: source and target. The main distinction between these environments concerns the mass of the Hopper's torso. Specifically, source sets the torso mass at 2.53429174 kg, while target raises it to 3.53429174 kg. The transition from the source to the target environment embodies the essence of sim-to-real transferability. This project aims to create algorithms capable of learning within simulation environments (source) and successfully applying acquired knowledge in real-world situations (target). The reward structure consists of three parts:

1. Healthy Reward: The robot earns a reward of +1 for each time step it remains active.

2. Forward Reward: The robot earns a reward by hopping forward via weight$\times\left(x_{t+1}-x_t\right)/dt$, where dt represents the time occurs between actions. This reward guides the agent to move forward in the x-direction (positive reward).

3. Control Cost: Cost to penalize the hopper for performing excessively large actions.

3 Policy Gradient Methods

In reinforcement learning (RL), policy gradient methods aim to maximize the expected return $J(\theta)$ by repeatedly estimating the gradient:

$$\nabla_\theta J(\theta) = \nabla_\theta \underset{\tau\sim\pi_\theta}{\mathbb{E}}[R(\tau)]$$

The expression for the political gradient comes out in the form:

$$\nabla_\theta J(\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\phi_t\right]$$

The weight parameter ϕ_t governs the gradient update rule and takes various forms across different policy gradient algorithms.

Policy Gradient Method	ϕ_t
REINFORCE	G_t
REINFORCE with baseline	$G_t - \beta(s_t)$
Actor-Critic	$V^\phi(s_t)$ or $Q^\phi(s_t, a_t)$
Advantage-Actor-Critic	$A^\phi(s_t, a_t) = Q^\phi(s_t, a_t) - V^\phi(s_t)$

4 REINFORCE

The REINFORCE method [4], a foundational paradigm in reinforcement learning (RL), offers a direct strategy for updating policy parameters based on environmental rewards. Central to REINFORCE stands the policy network π_θ, which outputs a normal probability distribution over actions from an input state s_t:

$$s_t \xrightarrow{\pi_\theta} \begin{matrix}\mu_\theta(s_t)\\ \sigma_\theta(s_t)\end{matrix}$$

$$a_t \sim \pi_\theta(\cdot|s_t) = \mathcal{N}(\mu_\theta(s_t), \sigma_\theta(s_t))$$

The primary objective of REINFORCE involves maximizing the expected return $J(\theta)$ via gradient descent:

$$J(\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}}[G(\tau)]$$

$$\nabla_\theta J(\theta) = \nabla_\theta \underset{\tau\sim\pi_\theta}{\mathbb{E}}[G(\tau)]$$

Here, the subscript of $\mathbb{E}$ indicates the variables undergoing integration. States and actions receive sequential sampling from the dynamics model $P(s_{t+1}|s_t, a_t)$ and policy $\pi_\theta(a_t|s_t)$, respectively:

$$\underset{\tau\sim\pi_\theta}{\mathbb{E}}[G(\tau)] = \int_\tau P(\tau|\pi_\theta)G(\tau)$$

$$\nabla_\theta \underset{\tau\sim\pi_\theta}{\mathbb{E}}[G(\tau)] = \int_\tau \nabla_\theta P(\tau|\pi_\theta)G(\tau)$$

The chain rule for differentiation transforms the gradient of $P(\tau|\pi_\theta)$ into:

$$\nabla_\theta P(\tau|\pi_\theta) = P(\tau|\pi_\theta)\nabla_\theta \log P(\tau|\pi_\theta)$$

Meanwhile, the probability of a trajectory τ under the policy π_θ decomposes into:

$$P(\tau|\pi_\theta) = \rho_0(s_0)\prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

The logarithm of $P(\tau|\pi_\theta)$ returns:

$$\log P(\tau|\pi_\theta) = w + \sum_{t=0}^{T}(\log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t))$$

Here, $w = \log \rho_0(s_0)$. When computing the gradient of $\log P(\tau|\pi_\theta)$, only the policy term contributes to the gradient since state transition probabilities remain independent of θ:

$$\nabla_\theta \log P(\tau|\pi_\theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

The expression for $\nabla_\theta J(\theta)$ transforms into:

$$\nabla_\theta J(\theta) = \int_\tau \nabla_\theta \log \pi_\theta(\tau)G(\tau)$$

$$\nabla_\theta J(\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}}[\nabla_\theta \log \pi_\theta(\tau)G(\tau)]$$

$$\nabla_\theta J(\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)G(\tau)\right]$$

Taking a step with this gradient pushes up the log-probabilities of each action in proportion to $G(\tau)$, the sum of rewards earned throughout the entire trajectory. However, this approach lacks logical coherence. Agents ought to reinforce actions on the basis of their consequences. Rewards earned prior to an action bear no relevance to the action's merit; only subsequent rewards matter. This intuition finds mathematical validation, expressing the policy gradient as follows:

$$\nabla_\theta J(\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)G_t\right] \tag{1}$$

$$\nabla_\theta J(\theta) \approx \frac{1}{T}\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)G_t$$

$$\text{with} : G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$$

The above expression leads to the policy parameter update rule:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

Here, α denotes the learning rate. Notably, REINFORCE updates policy parameters exclusively at the end of episodes, leading to potentially high variance in gradient estimates and slow convergence. This characteristic may hinder learning stability and efficiency, especially in scenarios with long-lasting episodes or sparse rewards. To mitigate variance, practitioners commonly introduce a baseline subtraction from the return, preserving bias neutrality.

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - \beta(s_t)) \right] \tag{2}$$

However, selecting an optimal baseline $\beta(s_t)$ may pose challenges in complex environments. A standard approach involves whitening, which normalizes returns by subtracting the mean and dividing by the standard deviation across episode time steps:

$$\beta(s_t) = \frac{G_t - \bar{G}(\tau)}{\sigma(G(\tau))}$$

This technique aims to enhance learning efficiency and potentially bolster policy performance by reducing feature correlations. In summary, while REINFORCE provides a direct path to policy gradient optimization, its reliance on Monte Carlo methods often results in heightened variance and slower convergence. These characteristics necessitate careful consideration and adaptation in diverse RL settings.

Algorithm 1 REINFORCE

```
 1: Parameters: $\alpha$, $\gamma$, $\beta(s)$
 2: Initialize the policy $\pi_\theta(a|s)$                ▷ with $\theta \sim \mathbb{R}^d$ at random
 3: for each episode do
 4:     $s_0 \sim \rho_0(\cdot)$
 5:     for each time-step $t \in T$ do
 6:         $a_t \sim \pi_\theta(\cdot|s_t)$
 7:         $f(s_t, a_t) \rightarrow r_t$, $s_{t+1}$
 8:         $s_t \leftarrow s_{t+1}$
 9:     end for
10:     $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
11: end for
```

5 Actor-Critic

The Actor-Critic (A2C) method [5], a robust paradigm within reinforcement learning (RL), combines elements of policy gradient methods with value-based functions to improve learning stability and efficiency. Differently from REINFORCE, which updates policy parameters based on environmental rewards, A2C incorporates a critic network to estimate state-value functions, enhancing gradient precision and policy refinement. Central to A2C stands the policy network (π_θ) and the critic network (V_ϕ), key components of the paradigm. The policy network $\pi_\theta(\cdot|s_t)$, parameterized by θ, outputs a normal probability distribution over actions from the input state s_t:

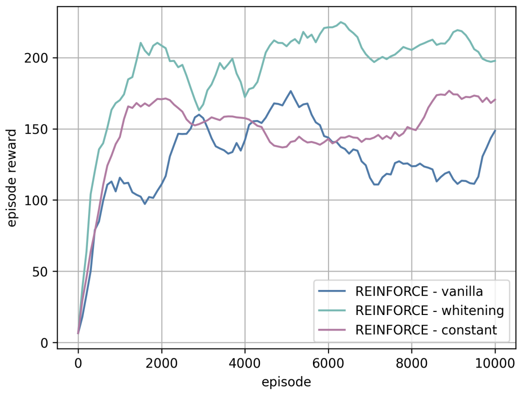

Figure 3: Performance comparison of the REINFORCE model in different baseline configurations.

$$s_t \xrightarrow{\pi_\theta} \begin{matrix} \mu_\theta(s_t) \\ \sigma_\theta(s_t) \end{matrix}$$

$$a_t \sim \pi_\theta(\cdot|s_t) = \mathcal{N}(\mu_\theta(s_t), \sigma_\theta(s_t))$$

Meanwhile, the critic network, parameterized by ϕ, estimates the value of the state s_t as $V_\phi(s_t)$:

$$V_\phi(s_t) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G(\tau)|s_0 = s_t]$$

Here, $G(\tau)$ denotes the sum of discounted rewards starting from state s_t and consistently following policy π_θ. The advantage function $A(a_t, s_t)$ in A2C quantifies the advantage of taking action a_t in state s_t over the estimated value of that state:

$$A(s_t, a_t) = Q_\phi(s_t, a_t) - V_\phi(s_t)$$

$$Q_\phi(s_t, a_t) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G(\tau)|s_0 = s_t, a_0 = a_t]$$

The advantage gauges whether an action outperforms or underperforms the policy's default behavior. Thus, $A(s_t, a_t)$ ensures that the gradient term $\nabla_\theta \log \pi_\theta(a_t|s_t)A(s_t, a_t)$ directs the policy towards increasing $\pi_\theta(a_t|s_t)$ only when $A(s_t, a_t) > 0$. Many policy gradient methods for episodic environments employ G_t as an empirical estimate for $Q(s_t, a_t)$:

$$A(s_t, a_t) = G_t - V_\phi(s_t)$$

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$$

The primary objective of A2C involves optimizing the policy network through gradient descent and the critic network via regression. The policy update aims to maximize the expected advantage:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)A(s_t, a_t)\right] \tag{3}$$

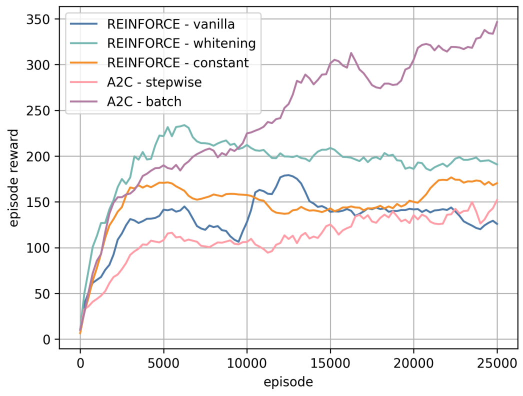

Figure 4: Performance comparison of the A2C model and the REINFORCE model in different configurations.

$$\nabla_\theta J(\theta) \approx \frac{1}{T}\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)$$

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J(\theta)$$

Concurrently with the policy update, the critic network minimizes the mean squared error (MSE) between discounted return G_t and critic estimation $V_\phi(s_t)$, so that the value network always approximates the value function of the most recent policy:

$$\nabla_\phi L(\phi) = \mathop{\mathbb{E}}_{\tau\sim\pi_\phi}\left[\nabla_\phi (G_t - V_\phi(s_t))^2\right]$$

$$\nabla_\phi L(\phi) \approx \frac{1}{T}\sum_{t=0}^{T} \nabla_\phi (G_t - V_\phi(s_t))^2$$

$$\phi \leftarrow \phi + \alpha_\phi \nabla_\phi L(\phi)$$

Algorithm 2 ACTOR-CRITIC

```
**Parameters:** $\alpha_\theta$, $\alpha_\phi$, $\gamma$
**Initialize** the policy $\pi_\theta(a|s)$                    ▷ with $\theta \sim \mathbb{R}^d$ at random
**Initialize** the state-value function $\hat{V}_\phi(s)$       ▷ with $\phi \sim \mathbb{R}^d$ at random
**for** each episode **do**
    $s_0 \sim \rho_0(\cdot)$
    **for** each time-step $t \in T$ **do**
        $a_t \sim \pi_\theta(\cdot|s_t)$
        $V_\phi(s_t)$
        $f(s_t, a_t) \rightarrow r_t, s_{t+1}$
        $s_t \leftarrow s_{t+1}$
    **end for**
    $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J(\theta)$
    $\phi \leftarrow \phi + \alpha_\phi \nabla_\phi L(\phi)$
**end for**
```

6 Proximal Policy Optimization

The Proximal Policy Optimization (PPO) method [6], a state-of-the-art paradigm within reinforcement learning (RL), introduces a "proximal" objective function to maintain policy updates closely aligned with previous policies, thereby ensuring training stability. Central to PPO stands the actor network (π_θ), parameterized by θ, which outputs probability distribution over actions from the input state s_t:

$$a_t \sim \pi_\theta(\cdot|s_t)$$

Meanwhile, the critic network V_ϕ, parameterized by ϕ, estimates the state-value function $V_\phi(s_t)$:

$$V_\phi(s_t) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta}[G(\tau)|s_0 = s_t]$$

Here, $G(\tau)$ denotes the sum of discounted rewards starting from state s_t and consistently following policy π_θ. The advantage function in PPO, akin to AC, quantifies the benefit of taking action a_t in state s_t over the estimated state value:

$$A(s_t, a_t) = Q_\phi(s_t, a_t) - V_\phi(s_t)$$

PPO ensures policy enhancement by optimizing the objective function through a clipped surrogate objective. The policy update objective under current distribution aims to maximize the expected advantage:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{s_t \sim \pi_\theta}\left[\min\left(r_t(\theta)A(s_t, a_t), \eta\right)\right]$$

$$\eta = \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A(s_t, a_t)$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

In the formulations above, r_t denotes the ratio of updated policy to previous policy, and ϵ acts as a hyperparameter governing policy alteration extent. The critic network undergoes an update to minimize the mean squared error (MSE) between discounted return G_t and critic estimation $V_\phi(s_t)$:

$$\nabla_\phi L(\phi) = \mathop{\mathbb{E}}_{s_t \sim \pi_\phi}\left[(G_t - V_\phi(s_t))^2\right]$$

The overarching PPO algorithm involves iterative policy and critic network updates employing stochastic gradient descent (SGD). Parameter θ and ϕ updates ensue using respective gradients.

PPO iterates through these updates to steadily refine policy and critic network capabilities in estimating state-value functions, thereby bolstering agent decision-making proficiency within reinforcement learning contexts.

7 Domain Randomization

Domain Randomization (DR) stands as a technique for enhancing the robustness and generalization of policies trained in simulated environments [7]. The objective of domain randomization involves introducing enough simulated variability at training time, enabling the model to generalize to real-world data during testing. By exposing the agent to a variety of different scenarios during training, DR helps bridge the sim-to-real gap, ensuring the learned policy performs effectively in real-world settings. The subsequent sections provide an overview of two variants of Domain Randomization: Uniform Domain Randomization (UDR) and Domain Randomization Optimization IDentification (DROID).

Algorithm 3 PPO

```
Parameters: $\alpha_\theta$, $\alpha_\phi$, $\gamma$, $\epsilon$
Initialize the policy $\pi_\theta(a|s)$                                ▷ with $\theta \sim \mathbb{R}^d$ at random
Initialize the state-value function $\hat{V}_\phi(s)$                  ▷ with $\phi \sim \mathbb{R}^d$ at random
for each episode do
    $s_0 \sim \rho_0(\cdot)$
    for each time-step $t \in T$ do
        $a_t \sim \pi_\theta(\cdot|s_t)$
        $V_\phi(s_t)$
        $f(s_t, a_t) \rightarrow r_t, s_{t+1}$
        $s_t \leftarrow s_{t+1}$
    end for
    $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J(\theta)$
    $\phi \leftarrow \phi + \alpha_\phi \nabla_\phi L(\phi)$
end for
```

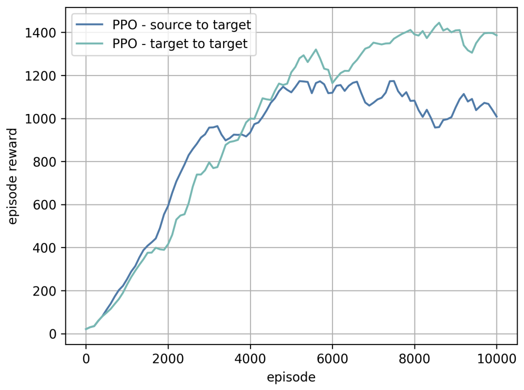

Figure 5: The sim-to-real gap (PPO).

8 Uniform Domain Randomization

Uniform Domain Randomization (UDR) enhances the robustness of an agent's policy by exposing it to variety of environmental conditions during training [7]. This research implements UDR on the link masses of the Hopper robot, specifically targeting the masses of the thigh, leg, and foot, while fixing the torso mass at a value different from the target environment by -1 kg. The torso mass accounts the discrepancy between the simulation environment and the real-world. Implementing UDR involved designing uniform distributions for each of the three link masses. Each training episode entails drawing values of these masses from their respective distributions. Specifically, for each link i, the environment initializes the boundaries of the uniform distribution $\mathcal{U}_\phi$ and samples the value of the physical parameter θ_i at the beginning of each episode:

$$\theta_i \sim \mathcal{U}_\phi((1-\phi)\cdot\theta_i^{(0)}, (1+\phi)\cdot\theta_i^{(0)})$$

where $\theta_i^{(0)}$ represents the original mass of the i-th link of the Hopper robot, and ϕ the variation factor. Here, $\mathcal{U}_\phi$ represents a continuous uniform distribution between $(1-\phi)\cdot\theta_i^{(0)}$ and $(1+\phi)\cdot\theta_i^{(0)}$. This approach forces the agent to adapt its policy in order to achieve satisfactory performance across environments with varying dynamics. UDR aims to train the agent to maximize its reward under different conditions, ensuring resilience to environmental variations. Since the choice of distribution acts as a hyperparameter, a grid search algorithm evaluates various distributions to return the optimal ϕ variation factor.

Hyperparameters	**UDR**
Variation Factor (ϕ)	$\{0.25, \mathbf{0.5}, 0.75\}$

Table 1: Grid Search for UDR

9 Domain Randomization Optimization IDentification

Domain Randomization Optimization IDentification (DROID) stands as a state-of-the-art approach in reinforcement learning aimed at refining simulation environments to enhance the robustness and transferability of learned policies [8]. In contrast to traditional Uniform Domain Randomization, which introduces stochastic variations to simulation parameters uniformly, DROID focuses on iteratively adjusting these parameters to closely approximate real-world dynamics. This section presents a comprehensive overview of DROID, detailing its objectives, methodology, and computational framework.

Objective

DROID attempts to address the sim-to-real gap inherent in reinforcement learning by optimizing the fidelity of simulation environments. By iteratively adjusting physical parameters such as masses within the simulation, DROID aims to minimize the discrepancy between the distributions of trajectories observed in simulation ($\mathcal{D}_{\text{sim}}(\theta)$) and those in the real world ($\mathcal{D}_{\text{real}}$). This optimization process significantly enhances the adaptability and generalization capabilities of reinforcement learning policies, enabling them to perform effectively across varying real-world conditions.

Problem Formulation

The algorithm efforts to minimize the Wasserstein distance W between the real-world trajectory distribution $\mathcal{D}_{\text{real}}$ and the simulated trajectory distribution $\mathcal{D}_{\text{sim}}(\theta)$. Initially, DROID sets θ to an initial set of physical parameters $\theta^{(0)} = \{\theta_{\text{torso}}, \theta_{\text{thigh}}, \theta_{\text{leg}}, \theta_{\text{foot}}\}$ that govern the simulation dynamics. This setup establishes an initial simulation data distribution $\mathcal{D}_{\text{sim}}(\theta^{(0)})$ from which iterative improvements begin. The algorithm proceeds through M iterations, refining θ to progressively minimize the discrepancy. The iterative refinement process (Algorithm 4) ensures that DROID systematically adjusts simulation parameters to minimize the discrepancy between simulated and real-world trajectory distributions.

Mathematical Foundation

The Wasserstein distance $W(\mathcal{D}_{\text{real}}, \mathcal{D}_{\text{sim}}(\theta_i))$ measures the discrepancy between the real-world trajectory distribution $\mathcal{D}_{\text{real}}$ and the simulated trajectory distribution $\mathcal{D}_{\text{sim}}(\theta)$.

$$W(\mathcal{D}_{\text{real}}, \mathcal{D}_{\text{sim}}(\theta)) = \inf_{\gamma \in \Gamma(\mathcal{D}_{\text{real}}, \mathcal{D}_{\text{sim}}(\theta))} \mathop{\mathbb{E}}_{(s,a)\sim\gamma}[c(s,a)]$$

$\Gamma(\mathcal{D}_{\text{real}}, \mathcal{D}_{\text{sim}}(\theta))$ represents the set of joint distributions $\gamma(s, a, s', a')$ with marginals $\mathcal{D}_{\text{real}}$ and $\mathcal{D}_{\text{sim}}(\theta)$, and $c(s, a)$ denotes the cost function, quantifying the dissimilarity between state-action pairs (s, a) in the real and simulated environments. This formulation enables Wasserstein distance to consider not only statistical moments but also the spatial arrangement of trajectories, crucial for accurately assessing and minimizing the discrepancy between simulated and real-world dynamics. By minimizing $W(\mathcal{D}_{\text{real}}, \mathcal{D}_{\text{sim}}(\theta))$, DROID drives its iterative optimization process to refine simulation parameters, aligning two-dimensional trajectory distributions and enhancing simulation fidelity. This approach aims to closely align the simulated environment with real-world dynamics. The systematic adjustment of simulation parameters in DROID aligns with the foundational objective of reducing the domain gap between simulation and reality in reinforcement learning. In the fine-tuning process, grid search algorithm evaluates various learning rates η to optimize performance. Among the several values, an optimal configuration returns a learning rate η set at $1e-4$. This choice appears pivotal as it balances the gradient descent efficiency with stability, crucial for the iterative adjustment of simulation parameters.

Hyperparameters	**DROID**
Learning Rate (η)	{1e-3, 5e-3, 5e-4, **1e-4**}

Table 2: Grid Search for DROID

Algorithm 4 DROID

```
**Initialize** $\theta = \theta^{(0)}$
**Collect** $\mathcal{D}_{\text{real}} = \{(s_i, a_i)\}_{i=1}^{N}$
**Collect** $\mathcal{D}_{\text{sim}}(\theta) = \{(s'_j, a'_j)\}_{j=1,\theta}^{N}$
**for** $m = 0, 1, 2, \ldots, M$ **do**
    $b \leftarrow W(\mathcal{D}_{\text{real}}, \mathcal{D}_{\text{sim}}(\theta))$
    **for** each $\theta_i \in \theta$ **do**
        $\theta_i \leftarrow \theta_i + \eta$
        $\mathcal{D}_{\text{sim}}(\theta_i) = \{(s'_j, a'_j)\}_{j=1,\theta_i}^{N}$
        $\mathcal{L}_i = W(\mathcal{D}_{\text{real}}, \mathcal{D}_{\text{sim}}(\theta_i))$
        $\nabla \leftarrow \nabla_{\theta_i} \mathcal{L}_i$
        $\theta_i \leftarrow \text{clip}(\theta_i, 0.01, 10.0)$
    **end for**
    $\mathcal{D}_{\text{sim}}(\theta) \leftarrow \{(s'_j, a'_j)\}_{j=1,\theta}^{N}$
**end for**
```

Figure 6 presents a comparison of the performance trends of the PPO model in the target environment across different training configurations. The graph shows the average rewards for three scenarios: training in the source environment (lower bound), training in the target environment (upper bound), training in the source environment with Uniform Domain Randomization (UDR), and training in the source with DROID configuration. Each configuration results from parallel training in three different environments for 25,000 episodes with different seeds. The results demonstrate that UDR implementation leads to a higher average reward than the configuration without randomization, indicating performance improvement in the target environment due to UDR. Furthermore, the graph shows that training with the DROID configuration outperforms standard Uniform Domain Randomization. DROID achieves this by optimizing parameter randomization, significantly reducing the discrepancy in sim-to-real transfer. This improvement underscores DROID's effectiveness in enhancing performance and reducing the domain gap between simulation and real-world conditions.

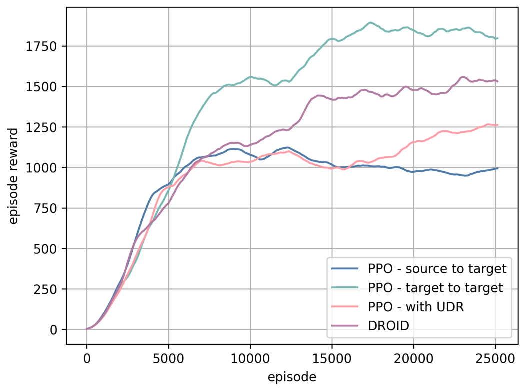

Figure 6: Comparison of PPO model performance trends in the target environment across different training configurations.

10 Conclusion

This research delves into the complexities of applying Reinforcement Learning (RL) to robotics, focusing specifically on addressing the Sim-to-Real challenge. The study aims to investigate the efficacy of Domain Randomization (DR) techniques in enhancing policy robustness and transferability across different environments. Through the exploration of various RL algorithms such as REINFORCE, Actor-Critic, and Proximal Policy Optimization (PPO), the research highlights their respective strengths and applications in training agents to perform tasks in simulated environments. The study evaluates each algorithm based on its ability to optimize policies effectively within the context of a Hopper robot tasked with hopping forward.

The core contribution of this report lies in the comprehensive analysis of two DR techniques: Uniform Domain Randomization (UDR) and Domain Randomization Optimization IDentification (DROID). UDR involves varying the masses of the robot's body parts uniformly, whereas DROID introduces an iterative optimization process to tailor simulation parameters for closer alignment with real-world dynamics. UDR demonstrates its effectiveness in enhancing policy resilience by exposing the agent to diverse environmental conditions during training. This approach aims to mitigate the "reality gap" between simulation and real-world scenarios, thereby improving the agent's adaptability. On the other hand, DROID represents a state-of-the-art strategy that iteratively adjusts simulation parameters to minimize the discrepancy between simulated and real-world trajectories. By optimizing simulation fidelity, DROID significantly enhances policy performance and transferability, as evidenced by superior results compared to traditional UDR.

This study underscores the importance of DR techniques in bridging the gap between simulated and real-world environments for reinforcement learning in robotics. Future research might explore further refinements to DR methodologies and their integration with state-of-the-art RL algorithms, ultimately advancing the field towards more robust and practical robotic applications.

References

[1] OpenAI. Introduction to Reinforcement Learning. Available at: https://spinningup.openai.com/en/latest/index.html, 2018.

[2] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv preprint arXiv:1712.01815*, 2019.

[3] Emanuel Todorov, Tom Erez and Yuval Tassa. MuJoCo: A Physics Engine for Model-Based Control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vilamoura-Algarve, Portugal, 2012, pp. 5026-5033, doi: 10.1109/IROS.2012.6386109, 2012.

[4] Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8(3), 229-256. GitHub: https://github.com/DLR-RM/stable-baselines3, 1992.

[5] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv preprint arXiv:1602.01783*. GitHub: https://stable-baselines3.readthedocs.io/en/master/modules/a2c.html, 2016.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*. GitHub: https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html, 2017.

[7] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba and Pieter Abbeel. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World. *arXiv preprint arXiv:1703.06907*, 2018.

[8] Ya-Yen Tsai, Hui Xu, Zihan Ding, Chong Zhang, Edward Johns and Bidan Huang. DROID: Minimizing the Reality Gap Using Single-Shot Human Demonstration. *arXiv preprint arXiv:2102.11003*, 2019.