Bridging the Sim-to-Real Gap in Reinforcement Learning: A Comparative Analysis of Algorithms on the OpenAI Gym Hopper Problem

Carlo Marra s334220 | Giovanni Pellegrino s331438 | Alessandro Valenti s328131

Abstract

This paper presents a introductory analysis of reinforcement learning algorithms on the OpenAI Gym Hopper problem, focusing on addressing the sim-to-real gap. We implemented and evaluated various reinforcement learning algorithms, including REINFORCE, Actor-Critic (A2C), and Proximal Policy Optimization (PPO). Our results indicate that while REINFORCE and A2C struggled to solve the Hopper task, PPO demonstrated effective performance. We extended our investigation by applying Uniform Domain Randomization (UDR) to the Hopper's masses during training, enhancing generalization and boosting performance. Finally, we implemented the DROPO off-policy method to identify optimal ranges for environment parameters, utilizing trajectories from a model trained on a target environment. Notably, DROPO accurately identified the target environment parameters, highlighting its potential for minimizing the sim-to-real gap.

1. Introduction

Reinforcement Learning (RL) is a paradigm in machine learning where an **agent** learns to make decisions by interacting with an **environment** to maximize **cumulative rewards**. The interaction is formalized using a *Markov Decision Process* (MDP), defined by a tuple (S, A, P, R, γ), where S represents the set of **states**, A represents the set of **actions**, $P(s'|s, a)$ denotes the **transition probability** from state s to state s' given action a, $R(s, a)$ denotes the **immediate reward** received after transitioning from state s to state s' due to action a, and γ is the **discount factor**, determining the importance of future rewards.
A **policy** $\pi(a|s)$ defines the agent's strategy by mapping states to actions. The agent's goal is to *find an optimal policy* π^* *that maximizes the expected return*, the sum of discounted rewards over time.
In our project, we utilized custom Hopper environments based on the original OpenAI Gym Hopper [2]. The Hopper is a simulated bipedal robot with four main parameters: torso, thigh, leg, and foot. The goal of the Hopper is to learn to move forward efficiently. We used two custom environments, "source" and "target": the "source" environment is used for training, while the "target" environment, which simulates *real-world* conditions, is used for evaluation. The key difference between the two environments is the mass of the torso, which is increased by one kilogram in the "target" environment.
All code implementations are available at GitHub

2. REINFORCE

2.1. Algorithm Description

The **REINFORCE** algorithm [8] is a Monte Carlo policy gradient method used in reinforcement learning for optimizing stochastic policies. It relies on using episode samples to update the policy parameters based on estimated returns. The policy parameters are updated in a direction that increases the expected return from the policy.
The objective of REINFORCE is to **maximize** the expected return $J(\theta)$ of a policy π_θ parameterized by θ:

$$J(\theta) = \mathbb{E}_\pi [G_t]$$

where G_t represents the return (*cumulative reward*) starting from time step t.
The gradient of the objective function can be expressed as:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi [Q^\pi(s, a) \nabla_\theta \ln \pi_\theta(a|s)]$$

By the definition of the return G_t and its relation to the action-value function Q^π, this can be simplified to:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi [G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)]$$

This equation forms the basis for the REINFORCE update rule.

The algorithm 1 initializes the policy parameters randomly. It generates a trajectory by following the current policy. For each time step in the trajectory, the return G_t

Algorithm 1 REINFORCE Algorithm

Initialize the policy parameter θ at random.
while True **do**
  Generate one trajectory using the policy π_θ:
  $S_0, A_0, R_0, S_1, A_1, R_1, \ldots, S_T$
  for $t = 0$ to T **do**
    Estimate the return G_t:
    $G_t = \sum_{k=t}^{T} \gamma^{k-t} R_k$
    where γ is the discount factor.
    Update the policy parameters:
    $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)$
    where α is the learning rate.
  end for
end while

from that time step onwards is calculated. The policy parameters are then updated using the **gradient ascent rule**, where the gradient is scaled by the return G_t and the log-probability of the action taken.
The REINFORCE algorithm updates the policy parameters to increase the likelihood of actions that yield higher returns, progressively improving the policy.

2.2. Introduction of the Baseline

While the approach described effectively improves the policy, it can suffer from *high variance*, making the learning process unstable and inefficient. A widely used variation of REINFORCE consist in the introduction of a baseline to reduce the variance of gradient estimates *without introducing any bias* [9], modifying the update rule as follows:

$$\theta \leftarrow \theta + \alpha(G_t - b)\nabla_\theta \ln \pi_\theta(A_t|S_t)$$

From a logical standpoint, the introduction of a baseline can serve as a **reference point**, stabilizing the updates by subtracting a value that ideally captures the average return. This reduces the fluctuations caused by individual episodes' returns without affecting the expected value of the gradient, thereby maintaining unbiased estimates.
The most standard way to use a baseline is to subtract the state-value from the action-value, resulting in an advantage function $A(s, a)$:

$$A(s, a) = Q(s, a) - V(s)$$

Using a constant baseline is a simpler yet effective approach.
A possible effective method for choosing a constant baseline is to calculate the **average return per episode** through preliminary experiments. Using this average return as the constant baseline b ensures that the baseline closely aligns with the agent's typical performance, thus minimizing the variance in gradient estimates.

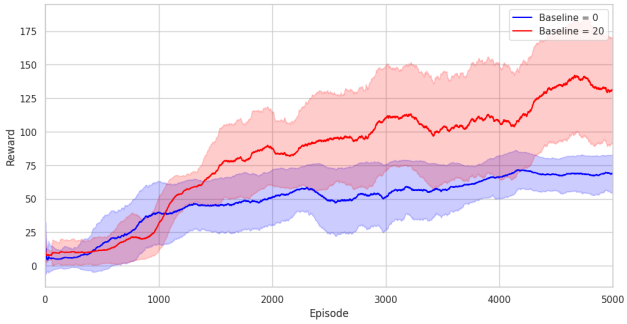

Figure 1. Comparison of REINFORCE algorithm performance with and without a baseline. The plot shows the average return over a window of 250 episodes. Shaded areas indicate the variance in returns.

2.3. Results

The results demonstrated in Figure 1 show that introducing a baseline in the REINFORCE algorithm significantly enhances the learning process. With a baseline value of 20, the algorithm achieves higher average rewards compared to the version without a baseline, which has an average return of 24.887 during training over 5000 episodes. This improvement suggests that the use of a baseline helps the agent to optimize its policy more effectively, resulting in better performance over time.
Although the variance in the returns does not appear to be significantly reduced, the stability of the training process is still noticeably improved. The agent with the baseline reaches **higher rewards much faster**, achieving a better mean reward after just 1000 episodes compared to the version without a baseline. This accelerated learning indicates that *the baseline helps the agent to make more consistent progress* with each policy update.
The chosen baseline of 20 proves effective, closely matching the average return observed during training without a baseline, even though this specific value was not initially selected based on the average return. While using a constant baseline derived from the mean return is beneficial, dynamically adjusting the baseline during training (continuously updating the baseline based on the running average of returns) could offer further advantages by ensuring it remains relevant as the agent's performance evolves.
However, despite these improvements, the REINFORCE algorithm alone **is not sufficient** to fully solve the Hopper environment. This is likely due to the complexity and high-dimensional nature of the Hopper task, which requires more advanced techniques and more sophisticated exploration strategies than what REINFORCE can provide.

3. Actor-Critic

3.1. Algorithm Description

The **Actor-Critic** family of algorithms in Reinforcement Learning [7] comprises two primary components: the Actor and the Critic. The **actor component** is responsible for selecting actions based on the current policy. It updates the policy parameters θ to optimize the policy $\pi_\theta(a|s)$ according to the provided feedback. The **critic component** evaluates the action taken by the actor by computing the value function parameters w. The value function refers to both the **state-action** value function $Q_w(s,a)$ and the **state** value function $V_w(s)$.

The **temporal-difference error** δ_t **(TD)** is a measure of the discrepancy between the value of the current state and the observed reward plus the predicted value of the next state. The TD error is calculated as follows:

$$\delta_t = r_t + \gamma V_v(s_{t+1}) - V_v(s_t)$$

In our work, we implemented the Advantage Actor Critic (A2C) algorithm, which leverages the computation of the advantages to stabilize training. In particular, we estimated the advantages using the TD-error to remove the *state-action value function* from the formulation. This passage is derived by using the relationship between $Q(s,a)$ and $V_v(s)$ from the Bellman optimality equation:

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$$

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V_v(s_{t+1})]$$

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

The **policy parameters** θ are then updated using the gradient of the log-probability of the current action, scaled by the **Advantage** $A(s,a)$. This update is performed as follows:

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

$$= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \delta_t$$

The parameters of the **value function** v are then updated by using the squared TD error as the loss function of the critic network:

$$v \leftarrow v - \alpha_v \nabla_v (\delta_t)^2$$

In the end, the current action and state are updated to the next action and state, respectively.

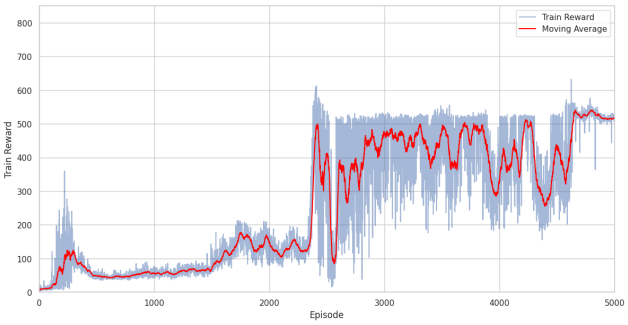

Figure 2. Episode Reward for Actor-Critic Algorithm. The plot shows the episode reward (blue line) and smoothed reward (red line) over 5000 episodes.

Algorithm 2 Action-Value Actor-Critic Algorithm

1: Initialize the state s, policy parameters θ, and value function parameters v to random values.
2: Sample an initial action $a \sim \pi_\theta(a|s)$.
3: **for** each time step $t = 1, \ldots, T$ **do**
4: Sample the reward $r_t \sim R(s,a)$ and the next state $s' \sim P(s'|s,a)$.
5: Compute the advantage A for the value at time t as the TD difference at time t:

$$A_t \approx \delta_t = r_t + \gamma V_v(s_{t+1}) - V_v(s_t)$$

6: Update the policy parameters θ according to:

$$\theta \leftarrow \theta + \alpha_\theta \delta_t \nabla_\theta \ln \pi_\theta(a_t|s_t)$$

7: Update the parameters of the value function:

$$v \leftarrow v + \alpha_v A_t \nabla_v A(s,a)$$

8: Update the current action $a \leftarrow a'$ and the current state $s \leftarrow s'$.
9: **end for**

3.2. Results - Actor-Critic vs REINFORCE

The *performance* of the Actor Critic algorithm on the Hopper problem is presented in Figure 2, which shows the episode reward as a function of the number of training episodes. Initially, there is a *noticeable increase in episode rewards*, suggesting some degree of learning and policy improvement. However, after 3000 episodes, the smoothed reward shows only minor fluctuations, implying that **the policy is not effectively exploring new strategies or improving its performance**.
A significant issue encountered during training was related with **specification gaming**, which is further discussed in section 3.3.

To encourage the agent to explore new strategies, an **en-**

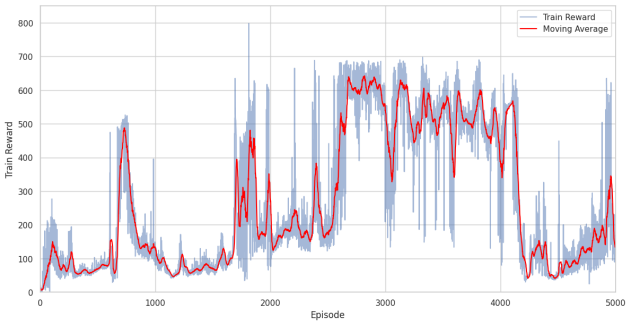

Figure 3. Episode Reward for Actor-Critic Algorithm with Entropy Coefficient. The plot shows the episode reward (blue line) and smoothed reward (red line) over 5000 episodes.

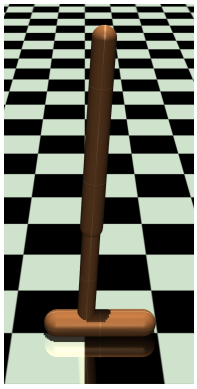

Figure 4. Render of the Actor-Critic agent in evaluation mode demonstrating the Hopper standing perfectly still at timestep 1500.

tropy coefficient (set to 0.1) was added to the loss function. This adjustment discouraged the agent from being overly certain about its action choices, rewarding diverse behaviour. As shown by the results in Figure 3, the addition of the entropy coefficient did lead to increased rewards and *increased exploration*. This is evidenced by the higher variance in episode rewards post-modification, suggesting that the agent was exploring a wider range of actions and policies. Despite these improvements, the *Advantage Actor-Critic* algorithm still **failed to produce agents that exhibited the intended behavior of the robot**. This indicates that, while Actor-Critic methods provide a more stable and efficient learning process compared to REINFORCE, further refinement and optimization are needed to handle complex tasks effectively.

3.3. Specification Gaming Issue

In our study, we observed a specification gaming issue during the training of the Actor-Critic algorithm on the Hopper environment. Specification gaming problems arise when an agent discovers strategies that technically satisfy the given reward function but do not achieve the overarching goal of the environment, exploiting loopholes or unintended aspects of the reward structure to maximize its reward without performing the intended task effectively [3].
Our Actor-Critic policy learned to **maximize its reward** by remaining **stationary** (Figure 4), thereby exploiting the reward structure that included survival without adequately penalizing inactivity.
This issue underscores the importance of designing robust reward functions that accurately reflect desired outcomes.
In our case, the issue probably originated from the *inability* of the A2C algorithm to solve the Hopper problem.

4. PPO

4.1. Algorithm description

Proximal Policy Optimization (PPO) is a state-of-the-art reinforcement learning algorithm designed to improve policy stability and performance. It aims to avoid the collapse of policy performance during training. PPO introduces a novel objective function that limits the magnitude of policy updates, ensuring steady improvements without drastic changes. In this paper, we discuss the results obtained from the **PPO-Clip** implementation available in the *stable-baselines3* GitHub OpenAI repository [5], used as default algorithm.

The **objective function** it maximizes is defined as:

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

where:

- $r_t(\theta)$ represents the ratio of the new policy's action probability to the old policy's action probability:

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

- $\hat{A}_t$ is the estimated advantage function at time t.
- The clip function limits the policy update by constraining the ratio $r_t(\theta)$ to stay within the range $[1-\epsilon, 1+\epsilon]$
- ϵ is a hyperparameter that controls how much the policy is allowed to change at each step.

The objective function L^{CLIP} and its implications can be visualized in Figure 5

4.2. Hyperparameter Tuning

In this section, we detail the hyperparameter tuning process, in which we used the *wandb* (Weights&Biases) library [1]. We conducted experiments to *optimize* the performance of PPO in both the source and target environments,

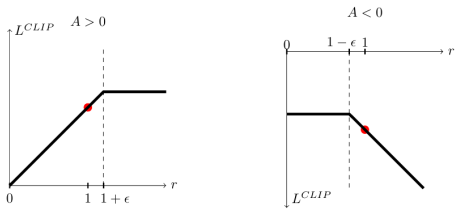

Figure 5. Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., r = 1. (Image and description taken from [6])

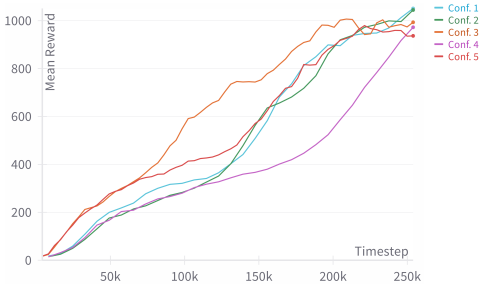

Figure 6. Wandb plot of the top 5 PPO model configurations trained on the **target** environment

with the goal of identifying the best-performing **hyperparameters**.
The following list of hyperparameters where chosen to perform the *wandb* sweep:

- **learning_rate**: [5e-3, 1e-3, 5e-4] (rate at which the model's parameters are updated during training)
- **n_steps**: [2048, 4096, 8192] (number of steps to run for each environment before updating the policy)
- **batch_size**: [64, 128, 256] (number of samples used in one iteration of model training)
- **ent_coef**: [1e-2, 1e-3, 1e-4] (coefficient for the entropy term)
- **gamma**: [0.990, 0.999] (discount factor for future rewards)

The models were trained using a total of **250k timesteps** and the results are summarized in **Table** 1 (tuning on the source) and Table 2 (tuning on the target). It can be observed that, both in the hyperparameter tuning of the source and the target, most of the parameters converge to *similar values* in their respective tables. Only the top 5 algorithms were considered because certain configurations lead to model **collapse** (for instance, a very high learning rate

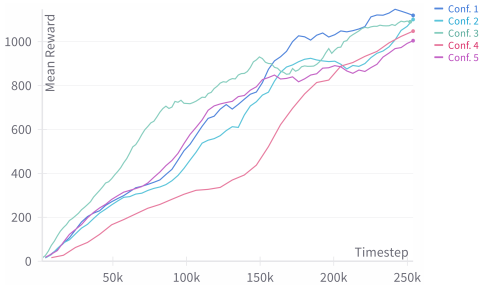

Figure 7. Wandb plot of the top 5 PPO model configurations trained on the **source** environment

and a very low batch size can cause excessive and noisy updates, gradient explosion, and thus a lack of convergence). In Figure 7 and Figure 6, it is possible to observe the performance of the top 5 configurations over time (timesteps).

4.3. Results - Upper and Lower Bound

We then trained a model using the best configuration for each environment, as found in 4.2. Both agents were trained for 1200k timesteps to find convincing upper and lower bounds for the subsequent Domain Randomization process, which is detailed in Section 5. The resulting policies were evaluated on 50 episodes, producing the following results:

- source $\rightarrow$ source: 1791.179 ± 58.981

  This testing configuration involves both training and testing the model within the source environment, providing a baseline performance measure.

- source $\rightarrow$ target (**lower bound**): 655.041 ± 4.234

  Here, the model is trained in the source environment and tested in the target environment. This setting emulates the standard sim-to-real transfer, where the policy struggles to replicate the same results shown during training. This value is taken as the lower bound on the performance obtained by a PPO policy transferred on the target environment.

- target $\rightarrow$ target (**upper bound**): 1747.241 ± 6.091

  In this configuration, the model is both trained and tested within the target environment, providing an upper bound on performance in the target setting.

The lower performance in the **source** $\rightarrow$ **target** configuration compared to the **target** $\rightarrow$ **target** configuration is expected due to the challenges associated with *domain adaptation*. The results highlight the difficulties encountered when directly transferring simulation-based models from their source environment to their intended target environment. In our case, the discrepancy in the dynamics of

Table 1. Best 5 Source Parameters Tuning

Conf.	learning_rate	n_steps	batch_size	ent_coef	gamma	Average reward ± std
1	1×10^{-3}	4096	256	1×10^{-4}	0.990	1657.577 ± 31.660
2	5×10^{-4}	4096	128	1×10^{-2}	0.999	1250.197 ± 4.380
3	1×10^{-3}	4096	64	1×10^{-2}	0.999	1076.443 ± 0.610
4	1×10^{-3}	2048	64	1×10^{-2}	0.999	1059.368 ± 4.590
5	1×10^{-3}	8192	128	1×10^{-2}	0.999	1044.853 ± 0.810

Table 2. Best 5 Target Parameters Tuning

Conf.	learning_rate	n_steps	batch_size	ent_coef	gamma	Avg reward ± std
1	1×10^{-3}	4096	256	1×10^{-2}	0.990	1470.484 ± 162.070
2	1×10^{-3}	8192	256	1×10^{-2}	0.999	1404.382 ± 49.130
3	1×10^{-4}	8192	64	1×10^{-2}	0.999	1109.542 ± 5.040
4	1×10^{-3}	8192	64	1×10^{-2}	0.999	1054.739 ± 0.620
5	1×10^{-3}	4096	64	1×10^{-2}	0.990	964.165 ± 22.02

the simulation between the source and target environments emulated the sim-to-real gap.

5. Uniform Domain Randomization

5.1. Randomizing the Environment

Domain Randomization is a technique used to address the Sim-To-Real Gap, it involves exposing the agent to a *variety* of simulated environments during training. This approach enhances the robustness and generalization capabilities of the trained policies, allowing them to **perform better** in real-world scenarios.

When moving from simulated environments to the real world, there are inherent differences which policies cannot account for. The main idea behind Domain Randomization is to try to make up for these differences by *changing the dynamics of the simulation* each time an episode starts during training. In our case, the environmental parameters that were considered for randomization were the **link masses** of the Hopper robot. The torso mass was left unchanged while the other three (thigh, leg, and foot) were drawn from a uniform distribution. We experimented with three ranges of values for the variables:

- ± 10% of the original value
- ± 25% of the original value
- ± 50% of the original value

We trained a policy for each range and then evaluated the results, obtaining the following results:

- ± 10% → Avg. Reward: 1557.972 ± 204.390
- ± 25% → Avg. Reward: 825.830 ± 11.829
- ± 50% → Avg. Reward: 986.061 ± 15.688

We ultimately opted for the **widest** of the three ranges to expand training and evaluate the effects of randomization.

The Upper and Lower Bound results shown in section 4.3 highlighted the fact that extensive training on a single environment leads to **highly specialized policies** that perform poorly on the target environment. To further explore the problem, we trained another PPO agent with the same set of hyperparameters for **400k timesteps**. As shown in **Table 4**, this shortly trained policy is less affected by the environment transfer, and achieves better rewards than the previously found lower bound.

Training a model with Uniform Domain Randomization is *computationally intensive*. The agent is exposed to changing environments and necessitates a larger number of episodes and updates to reach a given reward threshold, when compared with the standard source environment training. We observed this behaviour by training two policies, one of which using a randomized environment, up until the same reward threshold on the target environment. The results are summarized in **Table 5**. We confirmed that Randomization requires **longer training**, and that training on a single environment displays a steeper learning curve on the source environment.

Taking these results into account, we opted to train on the randomized environment for 3 000k timesteps.

5.2. Results - Overcoming the Gap

The final results of our experiments are shown in **Table 3**. The final model trained with randomization proved its efficacy on the target environment, nearly *doubling* the average reward threshold set by the lower bound. Uniform Domain Randomization demonstrated to be helpful in bridging the Sim-To-Real Gap, enhancing the robustness of the trained policy and enabling it to retain strong performances when transferred on the target environment. While these results prove the potential of Domain Randomization techniques in improving the *transferability* of simulation-based policies, they also highlight some troublesome aspects about the implemented technique.

Despite significant improvements compared to the lower

Table 3. Static vs Randomized Environment Comparison

Training Environment	Timesteps	Training Time (s)	Mean Reward - Source	Mean Reward - Target
Source - **Lower Bound**	1200k	1407.280	1791.179 ± 58.981	655.041 ± 4.234
Target - **Upper Bound**	1200k	1423.645	1733.712 ± 31.713	1747.241 ± 6.091
Randomized (± 50%)	3000k	3977.273	1793.762 ± 54.042	1297.988 ± 248.285

bound, the results still exhibit a substantial standard deviation (± 19.13% of the mean reward value), suggesting that **additional training may be necessary** to achieve a more stable and converged policy. When uniformly sampling parameters for training the policy, the effectiveness of individual values is overlooked. This means that all environments, regardless of their usefulness for policy reinforcement, have an equal probability of being sampled during training.
Uniform Domain Randomization is a promising starting point, but its simple design leads to inefficient training and limited results.
In Section 6, we focus on the problem of selecting optimal ranges for randomizing dynamics parameters.

Table 4. PPO Source Training Length Comparison

Timesteps	Avg source reward ± std	Avg target reward ± std
400k	1521.819 ± 39.412	914.541 ± 12.492
1200k	1791.179 ± 58.987	655.041 ± 4.231

Table 5. Static vs Randomized Time Comparison

Environment	Training Time (s)	Mean Reward - Source
Source	163.82	1330.87
Randomized (± 50%)	231.81	1250.53

6. DROPO

6.1. Algorithm Description

The **Domain Randomization Off-Policy Optimization** (DROPO) [10] algorithm is designed to infer domain randomization distributions using an offline dataset of trajectories. This method optimizes the domain randomization distributions to maximize the likelihood of observing real-world data in simulation, thereby facilitating more accurate sim-to-real transfer.

The DROPO algorithm (**Algorithm** 3) begins with an initial guess for the dynamics distribution parameters, denoted as $\phi_{\text{init}} = (\mu_{\text{init}}, \Sigma_{\text{init}})$. These initial parameters are informed by replaying the trajectory in the simulation environment.
Next, a **set** Ξ of K **dynamics parameter vectors** $\{\xi_k\}_{k=1}^{K}$ is sampled from the **current distribution** $p_\phi(\xi)$. For each **parameter vector** ξ_k, the simulator state is set to the **original real state** s_t. The **real action sequence** $a_{t..t+\lambda-1}$ is executed, and the **next state** $s_{t+\lambda}^{\xi_k}$ is observed in the simulator.

Algorithm 3 DROPO Algorithm

Initialize parameters $\phi_{\text{init}} = (\mu_{\text{init}}, \Sigma_{\text{init}})$
Fill a dataset $\mathcal{D}$ with trajectories from the target domain
while not converged **do**
  Sample a set Ξ of K dynamics parameters from current distribution, where $\xi \sim p_\phi(\xi)$
  for all $s_t, a_{t\dots t+\lambda-1}, s_{t\dots t+\lambda-1}$ in $\mathcal{D}$ **do**
    for all ξ_k in Ξ **do**
      Set simulator parameters to ξ_k and the simulator state to s_t
      Execute the real action sequence $a_{t..t+\lambda-1}$
      Observe next state $s_{t+\lambda t}^{\xi_k} \sim p(s_{t+\lambda}^{\text{sim}} \mid s_t, a_{t\dots t+\lambda-1}, s_{t\dots t+\lambda-1}, \xi_k)$
    end for
    Compute the mean $\bar{s}_{t+\lambda}^{\phi}$ and covariance $\Sigma_{t+\lambda}^{\phi}$
    Evaluate the log-likelihood $\mathcal{L}$ of $s_{t+\lambda}$ under $s_{t+\lambda} \sim \mathcal{N}(\bar{s}_{t+\lambda}^{\phi}, \Sigma_{t+\lambda}^{\phi})$
  end for
  Compute total log-likelihood $\mathcal{L} = \sum_t \mathcal{L}_t$
  Update ϕ towards maximizing $\mathcal{L}$
end while

The **next-state distribution** $p_{\text{sim}}(s_{t+\lambda}^{\text{sim}} | s_t, a_{t..t+\lambda-1}, \phi)$ is modeled as a Gaussian distribution with mean $\bar{s}_{t+\lambda}$ and covariance $\Sigma_{t+\lambda}$, estimated using the following equations:

$$\bar{s}_{t+\lambda}^{\phi} = \frac{1}{K} \sum_k s_{t+\lambda}^{\xi_k}$$

$$\Sigma_{t+\lambda}^{\phi} = \widehat{\text{Cov}}(p_{\text{sim}}(s_{t+\lambda} \mid s_t, a_{t..t+\lambda-1}, \phi)) + \text{diag}(\epsilon)$$

The value ϵ is a **hyperparameter** to regularize the likelihood computation and compensate for observation noise. Assuming the **true real-world observation** $s_{t+\lambda}$ originates from the next-state distribution, the **log-likelihood** $\mathcal{L}_t$ is calculated as:

$$\mathcal{L}_t = -\frac{1}{2}\Bigg(\log\det \Sigma_{t+\lambda}^{\phi} + \\ + (\bar{s}_{t+\lambda}^{\phi} - s_{t+\lambda})^T \Sigma_{t+\lambda}^{\phi^{-1}} (\bar{s}_{t+\lambda}^{\phi} - s_{t+\lambda}) \Bigg)$$

The **total log-likelihood** $\mathcal{L}$ of the dataset D is then obtained by summing the individual log-likelihoods:

$$\mathcal{L} = \sum_t \mathcal{L}_t$$

The **parameters** ϕ are adjusted to maximize the total log-likelihood $\mathcal{L}$. Any gradient-free optimization method can be employed for this purpose, we used the **Covariance Matrix Adaptation Evolution Strategy** [4] (CMA-ES). During optimization, the parameters are normalized and modeled as a truncated normal distribution bounded within two standard deviations from the mean.
Once the **domain distribution** $p_\phi(\xi)$ has converged, it can be used to parametrize the randomization process for training policies. During training, **new dynamics parameters** $\xi \sim p_\phi(\xi)$ are sampled at the start of each training episode. Given our simulation only setting, we simulated real-world trajectories using a *semi-converged* policy. Specifically, we trained a PPO agent for **250k timesteps** on the target environment, representing our real-world scenario. Subsequently, we collected trajectories by running this policy on the target environment for 5,000 timesteps.
The *initialization* of the parameters was performed as follows:

- **Mean Initialization**: For each of the four parameter masses, the bounds were set from a minimum of 0.1 to a maximum of 8. Then, for optimization convenience, they were normalized to the interval [0, 4].
- **Standard Deviation Initialization**: The standard deviations were initialized based on the width of the parameter range. The initial standard deviation was set to one-eighth of the width.

We set $K = 10$ and a budget of 100 evaluations for the CMA-ES optimization method.

6.2. Results

The DROPO algorithm was able to converge to a nearly single point estimation of the target environment parameters. This result was anticipated since the dynamics of the environment used for the DROPO optimization were the same as those of the target environment.
The DROPO algorithm optimized the parameters to the following Gaussian distributions:

Table 6. DROPO Results

Parameter	Target Mean (Kg)	Optimized Mean $\pm$ std
Torso	3.53429174	3.52546 $\pm$ 0.07653
Thigh	3.92699082	3.89995 $\pm$ 0.0031
Leg	2.71433605	2.7274 $\pm$ 0.02431
Foot	5.0893801	5.30127 $\pm$ 0.00466

The results summarized in **Table** 6 demonstrate the effectiveness of the DROPO algorithm in accurately identifying and **converging on the target environment parameters**. The algorithm successfully narrowed down the distributions to values very close to the actual target parameters, with relatively *small* standard deviations, indicating high confidence in the estimations.
Our implementation demonstrated remarkable efficiency by converging in 26.862 seconds. This is particularly notable given that the algorithm had to go through the entire trajectory for each of the K parameter samples to compute the loss function. The ability to achieve convergence swiftly despite the relative low number of function evaluations the optimizer was allowed to perform highlights the robustness and efficiency of the DROPO algorithm. This efficiency ensures that the algorithm can be applied to complex real-world scenarios without prohibitive computational costs, making it a valuable tool for sim-to-real transfer in reinforcement learning applications.

References

[1] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com. 4

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. 1

[3] DeepMind. Specification gaming: The flip side of ai ingenuity. Accessed: 2024-07-14. 4

[4] Nikolaus Hansen. The cma evolution strategy: A tutorial, 2023. 8

[5] 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. 4

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. OpenAI. 5

[7] Richard S. Sutton and Andrew G. Barto. Actor critic methods. In *Reinforcement Learning: An Introduction*, Adaptive computation and machine learning series, chapter 13.5. The MIT Press, Cambridge, MA, second edition, 2018. Accessed: 2024-07-14. 3

[8] Richard S. Sutton and Andrew G. Barto. Reinforce: Monte carlo policy gradient. In *Reinforcement Learning: An Introduction*, Adaptive computation and machine learning series, chapter 13.3-13.4. The MIT Press, Cambridge, MA, second edition, 2018. Accessed: 2024-07-14. 1

[9] Daniel Takeshi. Going deeper into reinforcement learning: Fundamentals of policy gradients, 2017. Accessed: 2024-07-14. 2

[10] Gabriele Tiboni, Karol Arndt, and Ville Kyrki. Dropo: Sim-to-real transfer with offline domain randomization. *Robotics and Autonomous Systems*, page 104432, 2023. 7