Bridging the Sim-to-Real Gap: A Comparative Study of Uniform and Automatic Domain Randomization in Reinforcement Learning

Dario Lupo
DAUIN
Politecnico di Torino
Turin, Italy
dario.lupo@studenti.polito.it

Vito Piazzolla
DAUIN
Politecnico di Torino
Turin, Italy
vito.piazzolla@studenti.polito.it

Fabio Veroli
DAUIN
Politecnico di Torino
Turin, Italy
fabio.veroli@studenti.polito.it

***Abstract*—This report explores the application of Uniform Domain Randomization (UDR) and Automatic Domain Randomization (ADR) techniques in reinforcement learning (RL) to address the Sim-to-Real Transfer Problem. This problem arises when models trained in simulated environments fail to generalize to real-world scenarios due to the inherent gap between simulation and reality. We investigate the effectiveness of UDR and ADR in two distinct *Gym* environments: *Hopper* and *Half Cheetah*, using the Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) algorithms. Our experiments demonstrate that ADR outperforms UDR in the more complex Half Cheetah environment, while their performances are almost comparable in the simpler *Hopper* environment. The results suggest that the complexity of the environment and the sensitivity to randomization play crucial roles in determining the effectiveness of these techniques. This work provides insights into the trade-offs between manual and automated domain randomization methods and their applicability in different RL scenarios.**

I. Introduction

Reinforcement Learning (RL) has shown remarkable success in solving complex tasks within simulated environments. However, a significant challenge remains in transferring these learned policies to real-world applications, a problem known as the Sim-to-Real Transfer Problem. Simulated environments are often simplified representations of reality, leading to a reality gap that can cause models trained in simulation to fail when deployed in real-world scenarios. To bridge this gap, Domain Randomization (DR) has emerged as a powerful technique, introducing variability into the training environment to improve the robustness and generalization of RL models.

In this report, we explore two variants of DR: Uniform Domain Randomization (UDR) and Automatic Domain Randomization (ADR). UDR involves manually defining fixed ranges for randomization, while ADR dynamically adjusts these ranges based on the model's performance, creating a curriculum of increasingly challenging environments. We evaluate these techniques in two distinct environments: *Hopper*, a simpler one-legged robot, and *Half Cheetah*, a more complex robot with two legs and one body. Using the PPO and SAC algorithms, we compare the performance of UDR and ADR in these environments, aiming to understand how the complexity of the task and the sensitivity to randomization affect the effectiveness of these methods.

Our findings reveal that ADR performs significantly better than UDR in the *Half Cheetah* environment, where the increased complexity benefits from ADR's dynamic adjustment of randomization ranges. However, in the simpler *Hopper* environment, UDR slightly outperforms ADR, suggesting that fixed randomization ranges are sufficient for less complex tasks. These results highlight the importance of tailoring domain randomization strategies to the specific characteristics of the environment and task at hand. This work contributes to the ongoing effort to improve the generalization capabilities of RL models, bringing us closer to deploying RL solutions in real-world applications.

II. Background

A. Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a highly effective and widely used reinforcement learning algorithm designed for training agents. It alternates between collecting interaction data from the environment and optimizing a surrogate objective function using stochastic gradient ascent [3]. PPO addresses the critical challenge of achieving substantial policy improvement while minimizing the risk of performance degradation. To balance this trade-off, PPO introduces two primary mechanisms: **PPO-Penalty**, which incorporates a penalty term to discourage large deviations from the current policy, and **PPO-Clip**, which uses a clipping function to limit the magnitude of policy updates. These features make PPO robust, efficient, and stable, enabling it to handle high-dimensional action spaces and complex environments effectively. Its simplicity and reliability have made PPO a cornerstone algorithm in modern reinforcement learning.

This study employs **PPO-Clip**, which incorporates clipping within the objective function to constrain the new policy's deviation from the old policy. The clipping mechanism is controlled by the hyperparameter ϵ, which determines the permissible extent of policy change. By preventing overly large policy updates, the clipping serves as a form of regularization,

promoting stability and improving the robustness of the optimization process [4].

PPO explores the environment by sampling actions from the most recent version of its stochastic policy. During training, the policy generally becomes less stochastic over time, as the update mechanism prioritizes actions with higher expected rewards, effectively shifting the policy toward exploitation of previously identified rewards.

B. Soft Actor-Critic

Soft Actor-Critic (SAC) is an off-policy actor-critic algorithm built on the maximum entropy reinforcement learning framework. This framework aims to maximize both the expected cumulative return and the entropy of the policy, promoting successful task performance while encouraging random exploration [5]. This approach inherently addresses the exploration-exploitation trade-off, as higher entropy facilitates more exploration, potentially enhancing learning efficiency [6].

SAC simultaneously learns a stochastic policy π_θ and two Q-functions, Q_{ϕ_1} and Q_{ϕ_2}. The algorithm has two primary variants: one with a fixed entropy regularization coefficient α and another that adjusts α dynamically during training. In this study, the fixed α variant is employed.

The Q-functions in SAC are updated using a modified version of the Twin-Delayed Deep Deterministic Policy Gradient (TD3) algorithm, with the following key differences:

- SAC incorporates an entropy regularization term in the target, encouraging the policy to remain stochastic.
- Target actions are sampled directly from the current policy rather than from a separate target policy, as in TD3.
- SAC, which trains a stochastic policy, relies on inherent stochasticity for smoothing, eliminating the need for explicit target policy smoothing used in TD3.

III. Related work

A. Domain Randomization

The Sim-to-Real Transfer Problem is one of the main challenges in Reinforcement Learning. Simulated environments are simplified models of the real world and often fail to capture the full complexity and variability of reality, introducing a reality gap. This gap often leads to models that perform well in simulation but fail to generalize when deployed in real-world scenarios. Domain Randomization (DR) has emerged as a powerful technique to address this issue, enabling models to bridge the sim-to-real transfer gap effectively. Uniform Domain Randomization (UDR) represents a specific implementation of Domain Randomization (DR), where parameters are randomized according to a uniform distribution across predefined ranges, ensuring exposure to a diverse set of conditions during training.

DR works by introducing variability into the simulated training environment. Instead of training a model in a fixed, deterministic simulation, DR randomizes key parameters such as lighting, textures, object properties, physics (masses, friction constant) and sensor noise. By exposing the model to a diverse range of randomized scenarios during training, DR ensures that the model does not rely on narrow simulation-specific features. Instead, it learns to focus on more fundamental and transferable aspects of the task, such as object shapes, spatial relationships, or physical interactions. This forces the model to learn robust features and strategies that are invariant to these changes, preparing it to handle the unpredictable conditions of the real world. [2]

B. Automatic Domain Randomization

DR requires significant manual effort to tune the right distribution to close the reality gap between the simulated and real-world environments. In this section, we describe how Automatic Domain Randomization (ADR), introduced in [1], automates this process.

The key hypothesis motivating ADR is that training on a maximally diverse distribution over the environment leads to transfer via emergent meta-learning. If the model has some form of memory, it can adapt its behaviour during deployment to improve performance in the current environment over time. As described in [1], this happens when the training distribution is so broad that the model's finite capacity prevents it from memorizing specific solutions for each environment.

In contrast to DR, which requires manually defining and fixing distribution ranges throughout training, ADR automates this process by dynamically adjusting the ranges.

Fig. 1 illustrates the ADR process. ADR automatically creates a curriculum of expanding distributions (randomization ranges). The environment is sampled from these distributions to generate training data, which is then used to optimize the model. The current environment is then used to further evaluate the performance of the model on the current distribution. Periodically, if the model demonstrates good performance, the distribution ranges are widened; otherwise, they are narrowed. This iterative process ensures efficient exploration and improves the model's robustness across environments.

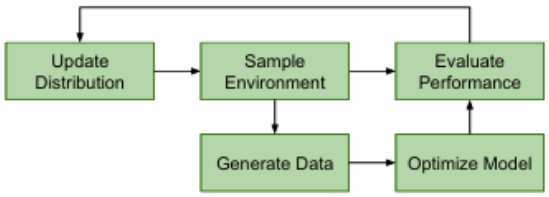

Fig. 1. Top-level diagram of ADR, as described in [1]

ADR is independent of the algorithm used for model training, as it only generates training data. With a sufficiently powerful model architecture and training algorithm, the distribution is expected to expand far beyond what is achievable with manual DR. This is because every improvement in the model's performance results in increased randomization.

ADR offers two key advantages over manual DR:

- **Curriculum-based training**: ADR employs a curriculum that gradually increases the difficulty of training as it progresses. This simplifies the training process because the model first learns to solve the problem in a single environment, and additional environments are only introduced once a minimum performance level is achieved.
- **Reduced need for manual tuning**: ADR eliminates the need to manually tune the randomization parameters. This is particularly critical as the number of randomization parameters grows, making manual adjustments increasingly challenging and time-consuming.

We now introduce the ADR algorithm (Fig. 2 and Fig.3), as originally presented in [1]. For details on the parameters used in our specific study, refer to Section V-C.

Each environment e_λ is parameterized by $\lambda \in R^d$, where d corresponds to the number of parameters randomized. For the i-th ADR parameter λ_i, i=1,...,d the pair (ϕ_i^L, ϕ_i^H) is used to describe a uniform distribution for sampling λ_i such that $\lambda_i \sim U(\phi_i^L, \phi_i^H)$. The overall distribution is given by:

$$P_\phi(\lambda) = \prod_{i=1}^{d} U(\phi_i^L, \phi_i^H)$$

At each iteration the ADR algorithm (Fig.2) randomly selects one dimension of λ_i to fix to a boundary value ϕ_i^L or ϕ_i^H (a process referred to as "*boundary sampling*"), while the other parameters are sampled from the standard distribution (ϕ_i^L, ϕ_i^H). The model's performance in the sampled environment is evaluated and stored in a buffer corresponding to the selected boundary of the chosen parameter. Once sufficient data is collected in the buffer, the average model performance is computed and compared to predefined thresholds. If the average performance exceeds the high threshold t_H, the boundary range for the selected parameter is expanded. Conversely, if the average performance falls below the low threshold t_L, the boundary range is contracted.

To quantify the amount of ADR expansion, the ADR *entropy* is also defined in [1] as $H(P_\phi) = \frac{1}{d}\sum_{t=1}^{d} \log(\phi_i^L - \phi_i^H)$, where a higher ADR entropy indicates a broader randomization sampling distribution.

The ADR algorithm adjusts P_ϕ by continuously fixing one parameter to a boundary value during sampling. For training data generation, λ is sampled from the updated P_ϕ, and the model is trained in the corresponding environment (Fig. 3).

To integrate ADR with data generation, at each iteration, the ADR algorithm (Fig. 2) is executed with probability p_b (referred to as the *boundary sampling probability*), while training data is generated (Fig. 3) with probability $1 - p_b$.

IV. Experimental Setup

The proposed algorithms were trained and tested using the *Gym* framework (OpenAI). Additionally, the *MuJoCo* physics engine was employed to simulate the environments and render their graphics.

Algorithm 1 ADR

Require: ϕ^0
Require: $\{D_i^L, D_i^H\}_{i=1}^d$
Require: m, t_L, t_H, where $t_L < t_H$
Require: Δ
  $\phi \leftarrow \phi^0$
  repeat
    $\lambda \sim P_\phi$
    $i \sim U\{1, \ldots, d\}$, $x \sim U(0, 1)$
    if $x < 0.5$ **then**
      $D_i \leftarrow D_i^L$, $\lambda_i \leftarrow \phi_i^L$
    else
      $D_i \leftarrow D_i^H$, $\lambda_i \leftarrow \phi_i^H$
    end if
    $p \leftarrow$ EVALUATEPERFORMANCE(λ)
    $D_i \leftarrow D_i \cup \{p\}$
    if LENGTH(D_i) $\geq m$ **then**
      $\bar{p} \leftarrow$ AVERAGE(D_i)
      CLEAR(D_i)
      if $\bar{p} \geq t_H$ **then**
        $\phi_i \leftarrow \phi_i + \Delta$
      else if $\bar{p} \leq t_L$ **then**
        $\phi_i \leftarrow \phi_i - \Delta$
      end if
    end if
  until training is complete

Fig. 2. Automatic Domain Randomization pseudo-code

Algorithm 2 Training Data Generation

Require: ϕ
  repeat
    $\lambda \sim P_\phi$
    GENERATEDATA(λ)
  until training is complete

Fig. 3. Training Data Generation

A. MuJoCo

MuJoCo, short for *Multi-Joint Dynamics with Contact*, is a general-purpose physics engine designed to support research and development in robotics, biomechanics, computer graphics, animation, and machine learning. It enables fast and accurate simulation of articulated structures interacting with their environments, making it a valuable tool for applications requiring detailed physical modeling.

B. Gym

Gym is an open-source Python library designed to facilitate the development and comparison of reinforcement learning algorithms. It provides a standardized API to connect learning algorithms with simulation environments, along with a curated collection of environments that comply with this API. Since its release, Gym's API has become the de facto standard for reinforcement learning research and experimentation.

C. Environment

The environments used in this project are:

- *Hopper*: the Hopper is a two-dimensional, one-legged figure consisting of four main body parts: the torso (at the top), the thigh (middle section), the leg (lower section), and a single foot on which the entire body

rests. The objective is to generate forward motion by applying torques to the three hinges connecting the body parts, enabling the Hopper to hop in the forward (right) direction.

- *Half Cheetah*: the Half Cheetah is a 2-dimensional robot with 9 links and 8 joints connecting them, including two paws. The objective is to apply torques to the joints to make the cheetah run as fast as possible in the forward (right) direction. Positive rewards are given based on the distance moved forward, while negative rewards are applied for backward movement.

Both environments have continuous state and action spaces:

- **State Space**: the state space is a continuous vector representing the figure's physical state, including positions, angles, and velocities of its body parts and joints.
- **Action Space**: the action space is a continuous vector representing the torques applied to each joint, with values bounded between $[-1, 1]$

D. Source and Target Domains

In this project, we simulated the sim-to-real transfer scenario using a simplified sim-to-sim setup, as no experiments were conducted on an actual physical robot. Specifically, we defined two custom domains: the **source** environment, where the policy was trained, and the **target** environment, where the policy was tested. The target environment was designed to represent the real-world setting.

To simulate the reality gap, the source domain of both environment has been generated by shifting the torso mass by 1kg with respect to the target domain.

The specific mass values for the source and target domains in the Hopper and Half-Cheetah environments are presented in Tables I and II, respectively.

TABLE I
HOPPER SOURCE AND TARGET DOMAINS

Hopper				
Domain	***torso***	***thigh***	***leg***	***foot***
source	2.534	3.927	2.714	5.089
target	3.534	3.927	2.714	5.089

TABLE II
HALF CHEETAH SOURCE AND TARGET DOMAINS

Half-Cheetah							
Domain	***torso***	***bthigh***	***bshin***	***bfoot***	***fthigh***	***fshin***	***ffoot***
source	5.360	1.535	1.581	1.069	1.426	1.179	0.850
target	6.360	1.535	1.581	1.069	1.426	1.179	0.850

V. EXPERIMENTAL RESULTS

A. Sim-to-Real Gap

We trained 10 models for 1 million PPO time-steps on both the source and target domains of the Hopper environment. Each model was subsequently tested over 100 test episodes, using the target domain as the test environment in all cases. The learning curves of the models are shown in Fig. 4 and Fig. 5, while the cumulative results are summarized in Tables III and IV.

Training on both domains and testing exclusively on the target domain allowed us to establish the lower and upper bounds of the sim-to-real gap. Specifically:

- The target→target configuration represents the ideal scenario, where the training domain is identical to the test domain. Here, the policy is well-suited to the environment dynamics, resulting in optimal performance.
- The source→target configuration demonstrates the expected performance drop due to the reality gap. In this case, the policy is trained on the source domain, which differs from the target domain, leading to suboptimal generalization. The policy tends to overfit to the source domain's dynamics and struggles to adapt effectively to the unseen conditions of the target domain, resulting in degraded performance.

Although the target→target configuration achieves higher performance, directly training in the target domain is often impractical in real-world scenarios. This is because simulations typically rely on assumptions and approximations of real-world dynamics. These approximations can propagate errors, and even if a model performs well in the simulated environment, it may fail completely in the real-world setting.

Fig. 4. Training performance of 10 models over 1M episodes using PPO on the Hopper Source Domain.

B. Sim-to-Real Transfer using UDR

To address the reality gap problem, a UDR algorithm is applied in the Hopper environment. The agent is trained in a source environment where the masses of the thigh, leg, and foot are randomized uniformly within the range $[0.5 \cdot original\_mass, 1.5 \cdot original\_mass]$, while the torso mass is fixed at $original\_mass - 1$.

a) Randomization Strategy: At the beginning of each episode, random values are assigned to the three masses of the system, sampled from a uniform distribution within the

Fig. 5. Training performance of 10 models over 1M episodes using PPO on the Hopper Target Domain.

predefined range. The torso mass remains constant throughout training. This randomization is applied consistently throughout the entire training process, ensuring that the agent encounters a diverse set of mass configurations in every episode.

b) Training Dynamics: The introduction of Domain Randomization (DR) has a noticeable impact on the training dynamics. As shown in Fig. 6, the learning process does not necessarily progress more slowly compared to a model trained without DR. However, the agent faces challenges in maintaining consistently high and stable performance due to exposure to varying mass configurations during training. This variability arises because the predefined randomization range imposes frequent shifts in dynamics, making it more difficult for the agent to refine a stable policy. Despite this, the agent demonstrates robustness by adapting to these changing conditions, ultimately leading to improved generalization capabilities.

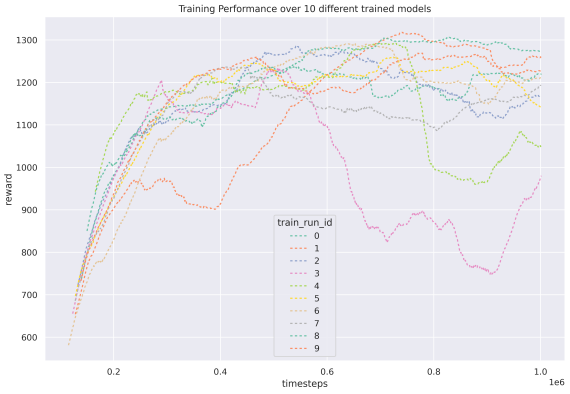

Fig. 6. Training performance of 10 models over 1M episodes with UDR in the Hopper environment.

Despite these challenges, the benefits of DR become evident during testing. When evaluated in the target environment—a setting designed to simulate real-world conditions—the agent demonstrates significantly improved performance. Specifically, the average episode duration consistently reaches the maximum limit of 500 steps, indicating that the agent has learned to survive for as long as possible. Furthermore, rewards obtained in the target environment are notably higher than those achieved by models trained without DR (i.e., trained in a source environment with fixed torso mass variations and then tested in the target environment with original mass values).

Although UDR effectively improves robustness and generalization, it does not address unmodeled effects outside the randomized parameters. For instance, the fixed torso mass in the Hopper environment limits the algorithm's ability to fully bridge the sim-to-real gap. These limitations highlight the need for more adaptive approaches, such as Automatic Domain Randomization (ADR).

C. Sim-to-Real Transfer using ADR

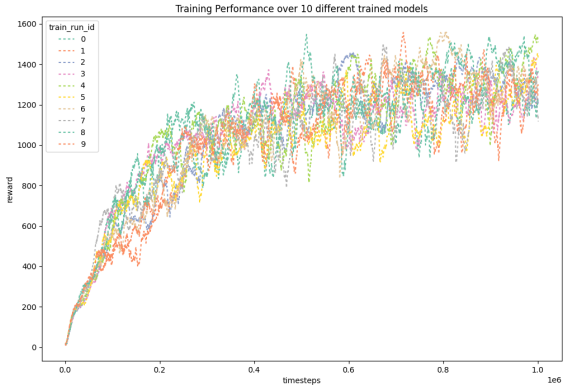

Fig. 7. Training performance of 10 models over 1M episodes with ADR in the Hopper environment.

To implement the ADR algorithm in our project, as structured in Fig. 2, the following strategy was adopted:

- **(Sample Environment Stage)**: Every 3 episodes, a new environment is sampled from the ADR distribution. Each environment has one parameter (mass) fixed to the corresponding lower or upper bound, while the remaining parameters (two masses in the Hopper's case) are sampled uniformly from their corresponding distribution. The torso mass is fixed, similar to UDR.
- **(Evaluate Performance Stage)**: Performance data are collected in the current sampled environment. These performance data are appended to the buffer of the corresponding selected parameter and boundary.
- **(Update Distribution Stage)**: Periodically, if the buffer is full, performance data are averaged. If the average meets the threshold condition, the corresponding parameter bound is expanded (increased or decreased).
- **(Generate Data and Optimize Model Stages)**: At the episode, if ADR is not being executed, a new environment

is sampled from the current ADR distribution, and the model is optimized using the data generated in this environment.
- The algorithm restarts from the Sample Environment Stage and repeats until the training is finished.

The `EvaluatePerfomance` function, referenced in ADR Algorithm (Fig. 2), was implemented by evaluating the performance of the current sampled environment over a single test episode. Although more complex evaluation methods (e.g., evaluating over multiple episodes) could better exploit ADR's capabilities, they would significantly increase computational cost and training time. For this simple implementation of the ADR algorithm, we opted to limit evaluation to one episode.

The boundary sampling probability was not explicitly used as a parameter in this implementation. However, since ADR is executed every 3 episodes and there are approximately 3600–3800 episodes in 1 million timesteps (using PPO), the boundary sampling probability can be considered roughly 33%.

For training in the Hopper environment, we combined ADR with PPO and trained 10 models over 1 million time steps. Unlike UDR, which always select parameters from fixed distributions, ADR begins with only one environment in its distribution. Thus, the lower and upper bounds of each parameter ($[\phi_i^L, \phi_i^H]$) were initially set to 95% and 105% of each mass of source Hopper. The following ADR parameter configuration was used :

- m: 10 (buffer size)
- Δ: 0.1 (update step size)
- $[t_L, t_H]$: $[550, 1150]$ derived from the rewards achieved by Hopper using PPO over 1 million timesteps.

This parameter configuration aimed to maximize ADR updates to test whether starting from a narrow distribution, ADR could achieve results comparable to UDR while generalizing the model to a range of mass configurations.

As shown in Tables III and IV, ADR is able to partially close the reality gap, outperforming the models trained exclusively on the source domain. Moreover, ADR achieves performance comparable to UDR, despite starting from a narrow distribution that only expands upon achieving sufficient performance. Both methods were tested across various target domains by varying Hopper's masses within the randomization ranges. Results indicate that both ADR and UDR produce models capable of generalizing effectively across varied environments, demonstrating their ability to train models that are not constrained to specific parameter values.

Figure 8 illustrates the evolution in ADR's entropy during training. As expected, ADR began with a narrower distribution, reducing randomization levels when performance was low, and expanding randomization as higher rewards were achieved.

However, due to the limited training time (1 million timesteps), we were unable to configure ADR to reach the broader entropy achieved by UDR. This limitation reduced the diversity of tested environments. Furthermore, we were unable to outperform UDR because ADR requires a balance between buffer size and update step size to prevent performance degradation caused by overly frequent updates.

Based on the results obtained, we hypothesize that improved alternation between data generation and model optimization, along with a more refined evaluation stage that incorporates additional performance statistics, could allow ADR to outperform UDR as entropy increases. This is consistent with findings from prior studies (e.g., [1]), though additional experimentation is required to validate this hypothesis.

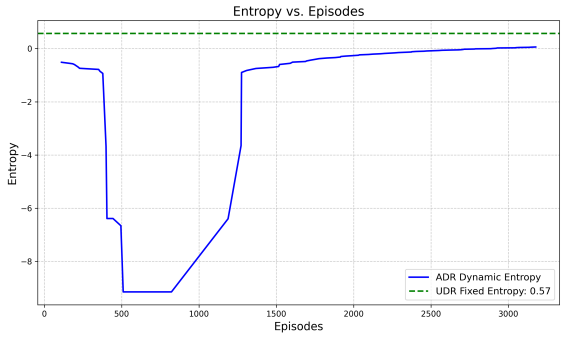

Fig. 8. ADR's Randomization Level during training

ADR's performance is strongly influenced by its parameters. Specifically, the update step size, initial parameters, and thresholds can significantly impact the learning curve of the algorithm used in combination with ADR. Fine-tuning these parameters and gaining a solid understanding of the environment and its reward system are critical for optimizing performance. Other parameters such as the buffer size, the boundary sampling probability, and the evaluation function also affect training time and computational overhead.

TABLE III
REWARD RESULTS OBTAINED BY TESTING EACH OF 10 SEPARATE MODELS OVER 100 EPISODES FOR HOPPER ENVIRONMENT

Train-env/ Test-env	**Test Reward**		
	avg	***std***	***median***
source/target	868.01	260.36	760.93
target/target	1638.79	158.66	1716.74
source-UDR/target	1302.08	59.24	1309.06
source-ADR/target	1294.87	320.45	1277.13

TABLE IV
EPISODE LENGTH RESULTS OBTAINED BY TESTING EACH OF 10 SEPARATE MODELS OVER 100 EPISODES FOR HOPPER ENVIRONMENT

Train-env/ Test-env	**Test Episode Length**		
	avg	***std***	***median***
source/target	243.90	62.61	217.00
target/target	476.25	57.13	500.00
source-UDR/target	499.70	2.04	500.00
source-ADR/target	364.39	94.08	363.00

D. UDR vs ADR on Half Cheetah Environment

To provide a comprehensive comparison between UDR and ADR, we evaluated both algorithms in the *Half-Cheetah* environment. Unlike the previous experiment with the *Hopper* environment, the training algorithm was switched from PPO to SAC. The experimental setup mirrored the *Hopper* environment, with the torso mass reduced by 1 kg and other parameters randomized according to the respective domain randomization algorithms.

For each algorithm, 10 models were trained over 200,000 time steps using the SAC algorithm. The following ADR parameter configuration was used :

- m: 1 (buffer size)
- Δ: 0.1 (update step size)
- $[t_L, t_H]$: $[600, 1450]$ derived from the rewards achieved by Hopper using PPO over 1 million timesteps.
- $[\phi_i^L, \phi_i^H]$: 95% and 105% of the source Hopper masses.

.

As demonstrated in Table V, Fig. 9 and Fig. 10, the ADR algorithm outperforms UDR during both training and testing phases. Specifically, ADR achieves a higher average reward, lower standard deviation, and higher median reward compared to UDR. During training, ADR not only attains higher rewards more quickly but also demonstrates greater consistency across training runs. This superior performance can be attributed to ADR's ability to dynamically adjust the randomization range, enabling the agent to learn more effectively how to adapt to weight variations.

In contrast, UDR struggled when applied to more complex environments like Half Cheetah, which involve multiple randomized parameters. Training from scratch in such scenarios proved challenging for UDR, as it required significantly more time to achieve results comparable to ADR.

Similar to the Hopper experiment, the trained models were evaluated on target environments with parameter values both inside and outside the randomization ranges. For environments with parameters inside the range, ADR achieved rewards comparable to its performance in the base domain (2000–2200 average reward), whereas UDR performance dropped by nearly 50%. In environments with parameters outside the randomization range, ADR consistently outperformed UDR, highlighting its superior generalization capability.

In this setup, we explored an "extreme" scenario where the step size was set to 0.1 and m=1, resulting in boundary updates every 3 episodes if threshold conditions were met. Figure 11 illustrates the progression of ADR entropy during training. In this setup, the entropy was initialized at a higher level to verify whether entropy would decrease in the early stages if performance was low. Unlike in the Hopper environment, ADR's entropy for Half-Cheetah eventually almost converged to the UDR fixed entropy. These results suggest that ADR's ability to adaptively increase entropy during training leads to performance improvements that are consistent with those observed in the Hopper environment. Furthermore, the model demonstrated strong generalization across a wide range of target environments. These findings partially support the hypothesis that ADR's increasing entropy can lead to superior outcomes compared to UDR.

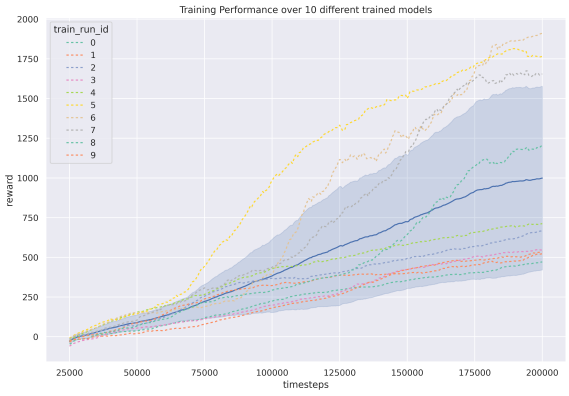

Fig. 9. Training performance of 10 models over 200k episodes with UDR in the Half Cheetah environment.

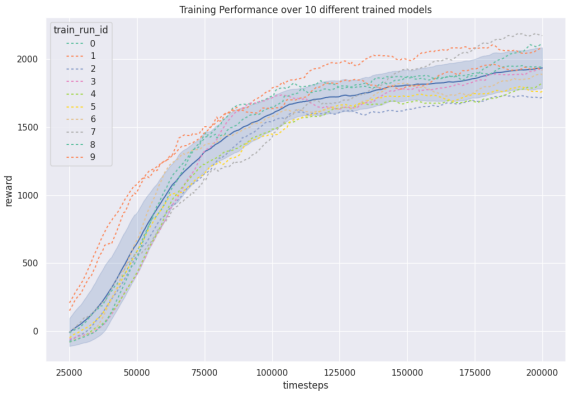

Fig. 10. Training performance of 10 models over 200k episodes with ADR in the Half Cheetah environment.

TABLE V
REWARD RESULTS OBTAINED BY TESTING EACH OF 10 SEPARATE MODELS OVER 100 EPISODES FOR HALF CHEETAH ENVIRONMENT

Train-env/ Test-env	**Test Reward**		
	avg	***std***	***median***
source-UDR/target	1126.68	645.47	716.48
source-ADR/target	2096.76	221.11	2074.29

The superior performance of ADR compared to UDR in the *Half Cheetah* environment, but not in the *Hopper* environment, can be attributed to several factors:

- Complexity of the Environment: The *Half Cheetah* environment is inherently more complex, with 9 links and 8 joints, compared to the simpler *Hopper*, which has only 4 main body parts. The increased complexity might benefit more from ADR's dynamic adjustment of randomization

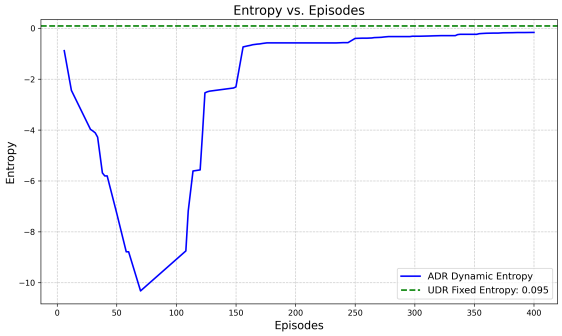

Fig. 11. ADR's Randomization Level during training for the Half Cheetah environment.

ranges, allowing it to explore a broader range of scenarios and better adapt to the diverse challenges posed by the environment. On the other hand, the Hopper environment is simpler, and UDR's fixed randomization ranges might be sufficient to cover the necessary variability for effective training. ADR's dynamic adjustments in this case might introduce unnecessary complexity, potentially leading to over-randomization or slower convergence, which could explain its slightly worse performance compared to UDR.

- Sensitivity to Randomization: The dynamics of the *Half Cheetah* environment may be less sensitive to specific ranges of randomization, enabling ADR to explore a broader parameter ranges without destabilizing the learning process. Conversely, the Hopper environment appears to be more sensitive to the exact randomization ranges. In this case, ADR's dynamic adjustments might result in ranges that are too narrow or too broad, leading to instability in the learning process and causing suboptimal performance compared to UDR.
- Training Dynamics: ADR training might be more effective in the *Half Cheetah* environment, where the agent needs to learn more complex behaviors (e.g., running forward efficiently). The gradual increase in difficulty provided by ADR could help the agent learn more robust policies. In the *Hopper* environment, the task is simpler (hopping forward), and UDR's fixed randomization ranges might provide enough variability for the agent to learn a stable policy. ADR's additional complexity might not offer significant benefits and could even slow down the learning process.
- Algorithm Interaction: The interaction between ADR and the underlying reinforcement learning algorithm (PPO in *Hopper* and SAC in *Half Cheetah*) might be more effective in the *Half Cheetah* environment, leading to better exploration and exploitation of the environment. In the *Hopper* environment, the interaction between ADR and PPO might not be as effective, potentially leading to suboptimal exploration and slower convergence.

VI. Conclusion

The performance gap between ADR and UDR in the *Hopper* and *Half Cheetah* environments likely derives from differences in the complexity and dynamics of these tasks, as well as how effectively each algorithm's randomization strategy addresses the unique challenges of each environment. The *Half Cheetah* environment, characterized by its multi-joint coordination and continuous motion dynamics, benefits significantly from ADR's adaptive adjustment of randomization ranges. By targeting progressively challenging variations, ADR likely enhances the agent's ability to generalize and adapt to dynamic scenarios in this more intricate setting.

In contrast, the *Hopper* environment, with its simpler single-leg dynamics and less complex control requirements, may not require the nuanced adjustments provided by ADR. In this case, UDR's fixed randomization strategy appears sufficient to capture the range of variability necessary for effective training, reducing ADR's relative advantage.

A. Further Development

This study is based entirely on simulated environments, and the lack of real-world experiments is acknowledged. While the sim-to-sim setup serves as a reasonable approximation, validating these techniques in real-world settings would significantly strengthen the findings and offer more robust evidence of their effectiveness in addressing the sim-to-real gap.

We also note that performance evaluation in ADR was limited to a single episode due to computational constraints, which may have restricted the ability to fully explore ADR's capabilities. Evaluating performance over multiple episodes could offer a more comprehensive understanding of ADR's potential and improve the reliability of the results. A longer training period could better evidence ADR's capabilities, giving the model enough time to optimize its policy over different environment configurations with ever-increasing randomization ranges.

Additionally, the effective use of Automatic Domain Randomization (ADR) requires a deep understanding of the system being modeled. Parameters such as buffer size, update step size, performance thresholds, initial randomization ranges, and boundary sampling probability can significantly impact ADR's performance. These parameters must be carefully selected and tuned to ensure ADR generates meaningful variability and adapts effectively to the task at hand. Misconfiguring these parameters could lead to inefficient training or failure to generalize to real-world conditions.

Future studies can also verify the effectiveness of ADR compared to UDR with different fixed entropies of UDR. This would allow a more nuanced comparison between the strategies, particularly in tasks where the optimal entropy might vary. A possible expansion of ADR could involve dynamically changing the threshold based on the model's increasing performance, to define new levels for increasing or decreasing the entropy. This approach could help address the challenge of choosing two fixed thresholds for performance

evaluation, as different stages of training may correspond to different performance values.

REFERENCES

[1] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang, "Solving Rubik's Cube with a robot hand," arXiv:1910.07113, Oct. 2019. [Online]. Available: https://arxiv.org/abs/1910.07113.

[2] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World," arXiv:1703.06907, Mar. 2017. [Online]. Available: https://arxiv.org/abs/1703.06907.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," arXiv:1707.06347, Jul. 2017. [Online]. Available: https://arxiv.org/abs/1707.06347.

[4] OpenAI, "Proximal Policy Optimization (PPO)," Spinning Up in Deep RL. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ppo.html.

[5] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," arXiv:1801.01290, Jan. 2018. [Online]. Available: https://arxiv.org/abs/1801.01290.

[6] OpenAI, "Soft Actor-Critic (SAC)," Spinning Up in Deep RL. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/sac.html.