Adaptive Domain Randomization for Robust Reinforcement Learning: A Comparative Study of UDR and SimOpt Approaches to bridge the sim-to-Real gap

Pranav Tripathi
Politecnico di Torino
Turin, Italy
s337810@studenti.polito.it

***Abstract*—This project implements and compares two domain randomization techniques for reinforcement learning in the Hopper environment: Uniform Domain Randomization (UDR) and Simulation Optimization (SimOpt). Using Soft Actor-Critic (SAC) as the base algorithm, I demonstrate how adaptive parameter optimization in SimOpt leads to improved policy transfer compared to traditional UDR methods. My implementation includes a comprehensive training framework that automatically optimizes simulation parameters using Bayesian optimization, resulting in more robust policies that generalize better to target environments. The project showcases the effectiveness of SimOpt in handling environment shifts and provides empirical evidence of its superior performance in terms of reward stability and transfer learning capabilities. Through extensive experimentation, I validate that adaptive domain randomization significantly enhances the policy's ability to handle real-world variations and uncertainties.**

I. Introduction

This project aims to train a Reinforcement Learning agent with various state-of-the-art algorithms, with the objective of bridging the Sim-to-Real gap. I present the main paradigms at the base of this framework. The code of the presented methods can be found at the project repository. [1].

A. Reinforcement Learning

Reinforcement learning (RL) is a framework that trains an agent through direct interaction with its environment. RL differs from the two main machine learning paradigms, supervised and unsupervised learning: it does not use examples of the desired behavior to choose its actions nor tries to find a hidden structure.

On each step, the agent sees a partial description (observation) of the environment's current state S_t. Based on the observation, the agent decides on which action A_t to take. The agent also perceives from the environment a single number called reward R_t. The agent's objective is to maximize its cumulative reward, called return. The agent decides which actions to take through a rule called policy π.

The following are other important concepts to define a RL problem. A sequence of states and actions is called a trajectory τ:

$$\tau = (S_0, A_0, S_1, A_1 \cdots) \tag{1}$$

Episodes are subsequences of the agent-environment interaction which are naturally divided based on certain events. The discounted return is defined as follows:

$$G_t = R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots = \sum_{t=0}^{\infty} \gamma^t R_{t+1} \tag{2}$$

with discount factor $\gamma \in (0, 1]$.

RL aims to select a policy that maximizes the expected return $J(\pi)$:

$$\max_{\pi} J(\pi) = \max_{\pi} E_{\tau\sim\pi}\left[\sum_{t=0}^{T} \gamma^t R_{t+1}\right] \tag{3}$$

with $T \in (0, +\infty)$.

The state-value function $V^{\pi}(s)$ gives the expected return if you start in state s and always act according to policy π:

$$V^{\pi}(s) = E_{\tau\sim\pi}[G_t(\tau)|S_0 = s] \tag{4}$$

The action-value function $Q^{\pi}(s, a)$ gives the expected return if you start in state s, take an arbitrary action a, and then always act according to policy π:

$$Q^{\pi}(s, a) = E_{\tau\sim\pi}[G_t(\tau)|S_0 = s, A_0 = a] \tag{5}$$

B. Sim-to-Real transfer

One of the most well-known applications of reinforcement learning is to train robots to do specific tasks. Training an agent relies on gaining experience, which can have various drawbacks when implemented with robots. A solution can be using the Sim-to-Real transfer [2]. The agent is trained in a simulated environment and then is transferred to a real robot in a real environment. The primary motivation behind Sim-to-Real transfer is the reduction of costs and risks associated

with training real robots. Moreover, it can reduce the training time and make data collection easier.

When implementing Sim-to-Real transfer, we face what is called the reality gap. It refers to the mismatch between the results we see in the simulated environment and the real world. These discrepancies can arise from various factors, including differences in sensor noise, and actuation dynamics, and there can be real-world experiences and tasks for which the agent was not trained in the simulation. As a result, a policy that performs well in simulation might not necessarily perform well in the real world.

There are different strategies to address the reality gap. In this project, I focused on Domain Randomization [3]. Domain Randomization involves training the policy on a wide range of simulated environments with randomized parameters. The goal is to expose the agent to enough simulated variability of the parameters at training time so that it learns a robust policy capable of generalizing to the real world. This method assumes that the variations in the simulation cover the possible variations in the real world.

For feasibility reasons, I simulated the Sim-to-Real transfer task using a controlled sim-to-sim scenario. The agent was trained in an environment we called the source environment and then tested in the target environment. To mimic the reality gap, discrepancies were manually added. Specifically, the torso mass of the Hopper robot is shifted by 30 percent between the source and target domains.

C. Simulation

The environment used in this project is the Hopper environment from OpenAI Gym. It is designed to simulate a one-legged robot, which consists of a torso, thigh, leg, and foot. The body parts are connected through three hinge joints. The primary objective for the Hopper is to learn to hop forward as rapidly as possible while maintaining stability.

Movement is achieved through the application of torques to the joints. The robot is considered alive as long as it remains upright and maintains balance; if it falls or becomes unstable, it's considered dead and the episode ends.

The simulated environment is a flat, planar surface on which the Hopper operates. In the target environment, the robot's body parts have the following masses: 3.53 kg for the torso, 3.92 kg for the thigh, 2.71 kg for the leg, and 5.089 kg for the foot. In the source environment, the only difference is the mass of the torso, which is reduced by 30 percent compared to the target. The state space is continuous and includes information about the robot and its joints in each state (positions, velocities, angles). The action space is also continuous, representing the three torques applied to the robot's joints.

Three different components are considered in the reward function:

- **Alive Bonus**: The robot receives a reward of +1 for each timestep it remains alive.
- **Forward Progress Reward**: The reward is based on the distance the Hopper moves forward in each timestep; the higher the distance, the higher the reward. It's calculated as the difference between its new position and the older position, divided by the duration of the action.
- **Control Penalty**: This negative reward penalizes excessive actions to encourage smooth and efficient movements, to avoid abrupt accelerations.

II. Related Work

Various papers are cited throughout the paper, but in particular, I want to highlight the following related works:

Domain Randomization for transferring deep neural networks from simulation to the real world by Tobin et al. (2017) [4] is one of the first papers on Domain Randomization. Domain Randomization is explained in sections I-B and III-D to understand my project, but the paper goes into deeper details about how the algorithm works, also with an application that is different from mine. Domain Randomization is not the only state-of-the-art algorithm one can use to overcome the Sim-to-Real gap, but it's the one I focused on in my work.

Soft Actor critic by OpenAI (2019) [5] is the paper proposing the working and flow for the soft actor critic policy. It was a good starting point to get familiar with the policy and how it works.

III. Methodology

I focused my attention on three algorithms in particular: REINFORCE, Actor-Critic, and Soft Actor Critic (SAC). These algorithms represent a progression from basic to more advanced techniques. They're implementations of the policy gradient method, which relies on gradient ascent to optimize the policy. The policy gradient theorem provides a way to compute the gradient of the expected return with respect to the policy parameters θ:

$$\nabla_\theta J(\theta) = E_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)] \tag{6}$$

A. REINFORCE

REINFORCE relies on computing the discounted return G_t and uses the policy gradient theorem for updating the policy parameters. It works because the expectation of the gradient is equal to the actual gradient:

$$\nabla_\theta J(\theta) = E_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)] = E_\pi[G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)] \tag{7}$$

Algorithm 1 REINFORCE (Vanilla Policy Gradient)

Algorithm parameter: step size $\alpha > 0$
Initialize: policy parameter θ
loop
  (for each episode) Generate one episode following policy π_θ: $S_0, A_0, R_1 \ldots, S_{T-1}, A_{T-1}, R_T$ **for** $t = 0, 1, \ldots T-1$ **do**
    $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_\theta \log \pi_\theta(A_t|S_t)$
  end for
end loop

This equation holds true because $Q^\pi(S_t, A_t) = E_\pi[G_t|S_t, A_t]$, as we've seen with equation 5. The process

behind the REINFORCE algorithm is explained in detail in Algorithm 1 [6].

A possible variation of REINFORCE is to subtract a baseline value from the return G_t. With this change, the variance of gradient estimation is reduced while keeping the bias unchanged.

B. Actor-Critic

Algorithm 2 Actor-Critic Algorithm

Algorithm parameters: Step sizes $\alpha_\theta > 0$, $\alpha_w > 0$
Initialize: policy parameter θ and state-value weights w
loop
  (for each episode) Initialize: S_0 (first state of episode)
  $I \leftarrow 1$ **while** S_t is not terminal (for each time step t) **do**
  $A_t \sim \pi_\theta(\cdot|S_t)$
  Take action A_t, observe S_{t+1}, R_t
  if S_{t+1} is terminal **then**
    $V_w(S_{t+1}) \leftarrow 0$
  end if
  $\delta_t \leftarrow R_t + \gamma V_w(S_{t+1}) - V_w(S_t)$
  $w \leftarrow w + \alpha_w \delta_t \nabla V_w(S_t)$
  $\theta \leftarrow \theta + \alpha_\theta I \delta_t \nabla \ln \pi_\theta(A_t|S_t)$
  $I \leftarrow \gamma I$
  $S_t \leftarrow S_{t+1}$
 end while
end loop

The Actor-Critic method combines elements of policy-based and value-based approaches. It addresses some limitations of REINFORCE by using a value function to reduce variance in the policy gradient estimates. The method involves an actor (policy) and a critic (value function).

The critic updates the parameters w of the value function (in our case a state-value function $V_w(s)$). The actor updates the policy parameters θ, in the direction suggested by the critic.

The policy update is guided by the temporal difference (TD) error δ_t, which provides a more accurate and lower variance estimate of the return. The actor's policy is updated using the gradient of the expected return, modulated by the TD error. The critic updates its value function parameters to minimize the TD error, using a mean squared error loss.

The process behind the Actor-Critic algorithm is explained in detail in Algorithm 2 [6].

C. Soft Actor Critic

Soft Actor-Critic (SAC) is a state-of-the-art off-policy actor-critic algorithm that operates under the maximum entropy reinforcement learning framework. The agent aims to maximize both expected reward and policy entropy, encouraging exploration while succeeding at the task by acting as randomly as possible within optimal behavior. To achieve this, SAC uses entropy regularization in its objective function, which prevents premature convergence to suboptimal policies and enables the capture of multiple modes of near-optimal behavior. Compared to traditional methods like REINFORCE and basic Actor-Critic, SAC demonstrates superior sample efficiency, more robust performance, and enhanced stability, particularly in continuous action spaces. For this reason, moving forward, we'll focus on this algorithm.

D. Domain Randomization

As explained in Section I-B, Domain Randomization is one of the possible algorithms to bridge the Sim-to-Real gap. It does so by exposing the agent to randomized parameters to cover the potential variations in the real world. In particular, we implemented Uniform Domain Randomization (UDR), which is how Domain Randomization was originally implemented [4], and SimOpt, which is an offline Bayesian optimization approach to find optimal randomization ranges.

In UDR, each randomization parameter is bounded by an interval and each parameter is sampled within the range in a uniform distribution. SimOpt takes the idea of UDR and expands on it through systematic offline optimization. The main difference is that in SimOpt, the uniform distribution's bounds are optimized across multiple independent trials rather than being manually fixed.

At each trial, the SimOpt algorithm uses a Gaussian Process model to suggest new parameter bounds based on the performance history of previous trials. A full policy is then trained with those bounds, using the same uniform sampling as UDR during training. The target environment reward is evaluated and recorded.

Once enough performance data is collected (e.g., after 10 initial random trials), the Gaussian Process uses this data to model the relationship between parameter bounds and transfer performance. Bayesian optimization with an acquisition function (Expected Improvement) then suggests the most promising bounds to explore next.

If a trial achieves high target environment performance, the Gaussian Process learns that those parameter bounds are beneficial. Conversely, poor performance indicates suboptimal ranges. This iterative process continues across trials, converging toward parameter bounds that maximize sim-to-real transfer.

Both methods aim to improve domain randomization, but SimOpt uses a learning-based offline optimization approach to find optimal randomization ranges, while UDR relies on manually selected fixed ranges.

IV. Experiments

A. REINFORCE vs Actor-Critic

I trained three algorithms:

- REINFORCE
- REINFORCE with constant baseline $b = 20$
- Actor-Critic

and compared their performance. To do so, I trained each algorithm for 1,000,00 episodes in the source environment. After this I tested the three algorithms to check their performance , while also keeping the training data to compare them later on.

B. Soft Actor Critic

To implement SAC, I used the stable-baseline3 library.

First,I implemented the SAC policy with the standard hyperparameters to get an idea of what is the performance with the standard setting . I also calculated the upper bound and lower bound with these settings .

Then, I implemented a hyperparameter optimization procedure to find the best hyperparameters. This was done through the Optuna [7] library, which is the standard approach when implementing stable-baselines3. Optuna is a framework for hyperparameter optimization that can efficiently explore large search spaces and prune unpromising evaluations (or trials). In particular, I implemented 50 trials, where the agent was trained in the source environment for 100,000 timesteps in each trial, resulting in a total optimization budget of 5,000,000 timesteps. The agent was periodically evaluated on the target environment every 10,000 timesteps, with each evaluation consisting of 5 episodes.

As a pruner, I used Optuna's MedianPruner, which periodically compares the agent's intermediate performance against the median of all previous trials' performances at the same step. This allows the algorithm to terminate unpromising trials early, significantly reducing the overall optimization time while maintaining search quality.

As a pruner, I periodically compared the agent's performance to its previous evaluations to ensure that the return was consistently improving (within a threshold)

The values used in the hyperparameter tuning are listed in Table I. The best combination of hyperparameters is the one highlighted in bold.

TABLE I
SEARCH SPACE FOR SAC

Hyperparameters	Search Space
Batch size	{32, 64, 128, 256,**512** ,1024}
Buffer Size	{ 100000,**500000** ,1000000}
Gamma	{0.99, 0.995, **0.999**}
Learning rate	{1e-2, 3e-3, 5e-3, 1e-3, **3e-4**, 5e-4}

With the best hyperparameters, I trained the agent in three different scenarios:

- trained in the source environment and tested in the source environment (Source $\rightarrow$ Source)
- trained in the source environment and tested in the target environment (Source $\rightarrow$ Target)
- trained in the target environment and tested in the target environment (Target $\rightarrow$ Target)

For every scenario, the training lasted 1,000,000 time steps. At the end of the training, I tested the three agents for 50 episodes.

C. Uniform Domain Randomization

To implement Uniform Domain Randomization (UDR), I systematically varied the masses of the Hopper's active body parts—the thigh, leg, and foot—while maintaining a fixed torso mass to preserve the essential core dynamics. At the start of each training episode, the agent was exposed to a novel environment where the dynamic parameters were sampled from a uniform distribution centered on the default source values.

The randomization bounds were established by applying a symmetric scaling factor to the nominal masses. Specifically, we defined the lower and upper bounds by subtracting and adding a proportional variation percentage to the original mass values, respectively. This method generated a continuous interval of possible dynamics around the baseline, requiring the agent to adapt to a spectrum of heavier and lighter physical configurations.

Rather than relying on automated hyperparameter optimization frameworks like Optuna, I conducted a manual sensitivity analysis to identify the most effective randomization intensity. I trained and evaluated agents across three distinct levels of mass variation: a conservative range, a moderate range, and an aggressive range. Each model underwent 1,000,000 time steps of training in the randomized source environment before being tested in the target environment. This empirical study revealed that the moderate 0.3 variation provided the optimal balance, maximizing the agent's ability to transfer its learned policy to the real-world conditions.

The values in bold in Table III were the ones giving the best return after 100,000 time steps.

TABLE II
MASSES OF THE HOPPER'S RANDOMIZED BODY PARTS

Body Part	Mass (in kg)
Thigh	3.92
Leg	2.71
Foot	5.089

TABLE III
SEARCH SPACE FOR UDR

Hyperparameters	Search Space
Mass Variation	{0.1, **0.3**, 0.5,}

I then trained the agent in the source environment for 1,000,000 time steps using the UDR algorithm with the parameters mentioned in the table 3 . I tested on all 3 separately and found the best mass variation to be 0.3

D. SimOpt

We implemented SimOpt as a Bayesian Optimization process to actively search for simulation parameters that maximize the agent's performance on the target environment. Unlike UDR, which indiscriminately widens the distribution to cover potential reality gaps, SimOpt treats the simulation parameters (masses) as hyperparameters to be tuned. We use a Gaussian Process (GP) regressor to model the relationship between these physical parameters and the agent's true performance on the target domain.

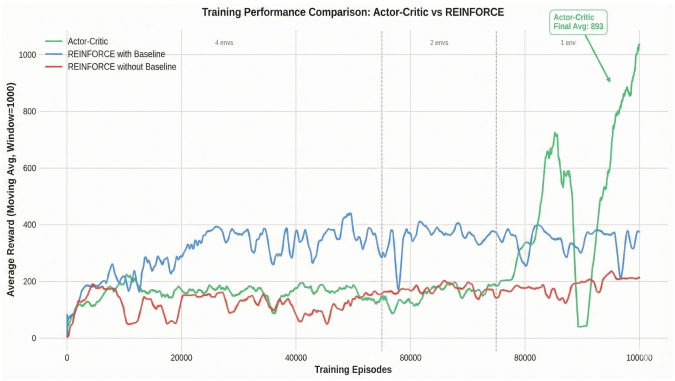

Fig. 1. Comparison between REINFORCE and Actor-Critic

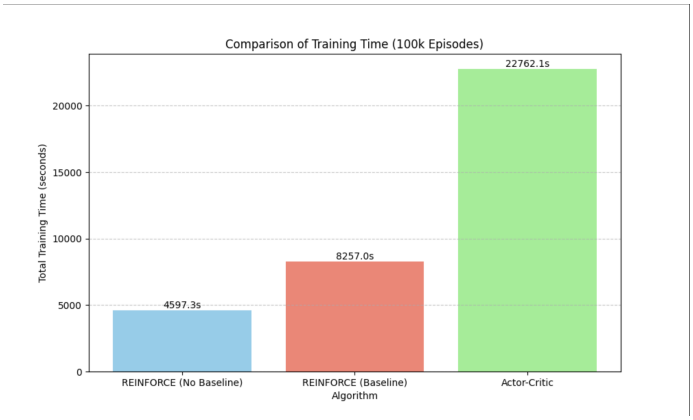

Fig. 2. Training time comparison between REINFORCE and Actor-Critic

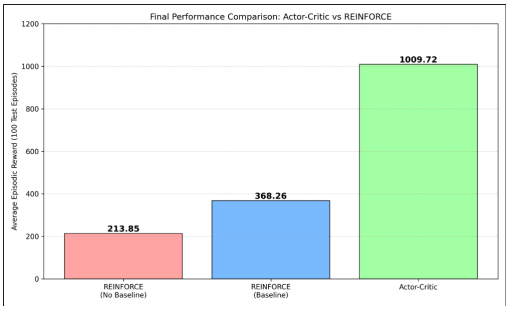

Fig. 3. Test performance for Reinforce and Actor Critic

This approach iterates through a cycle of training and evaluation: we select a set of physics parameters (suggested by the GP), train a SAC agent in this specific simulation configuration for 1,000,000 timesteps, and then evaluate its zero-shot transfer performance on the target environment. The resulting reward is fed back into the optimizer to refine its belief about which simulation parameters yield the most robust policies.

This implementation involves the following key components for SimOpt:

- Number of Initial Points: The set of random parameter configurations evaluated to initialize the GP model (10 trials).
- Number of Optimization Iterations: The number of sequential steps where the optimizer actively selects new parameters to maximize the Expected Improvement (EI) (20 trials).
- Parameter Space: The search bounds for the mass variation of the thigh, leg, and foot, centered around the source values (0.3)
- Training Budget: The fixed duration for training each candidate policy to ensuring fair comparison (1M steps per trial).

It is to be noted that I fix the number of steps per trial to be fixed for 1 million time step (2000 episodes). When after all trials were finished , I trained the model with best parameters found in the most rewarding trial for 5000 episodes . Then it was tested and all the results for comparison were obtained.

V. Results

A. REINFORCE vs Actor-Critic

I ran the Actor-Critic and reinforce for testing in the target env for 100 episodes.The resulting graph is shown in Fig. 1 1.

B. Soft Actor Critic

For every scenario (Source → Source, Source → Target and Target → Target), the agent was tested for 50 episodes after 1,000,000 time steps of training. The results can be seen In Table V.

TABLE IV
SAC Results (with default parameters)

Training and evaluation environments	Return
Source → Source	1996.55
Source → Target	1015.89
Target → Target	1361.44

TABLE V
SAC Results(with paramters found through optuna)

Training and evaluation environments	Return
Source → Source	1332.97
Source → Target	1441.52
Target → Target	1239.66

There's a gap in performance between Source → Target and Target → Target. This was to be expected: the Sim-to-Real gap is caused by a discrepancy between the two domains. The same happens in this case: the agent trained in the source domain when tested in the target environment encounters different dynamics not experienced before. In this case, the "different dynamics not experienced before" are the shift of the torso's mass in the source env (reduced by 30 percent)

Training directly on the target environment led to a higher return, but this is impractical in reality for the reasons explained in section I-B.

The following sections try to reduce this gap between the lower bound (Source → Target) and upper bound (Target → Target).

It should also be seen that when we use the best found hyperparameter through optuna search , the source to target performance improves , showing that the found parameters are performing better in the target environment . But the target to target performance decreases , one of the possible reasons can be the fact that the found parameters were found by training on source environment. , and in target to target we were training it on the target environment.

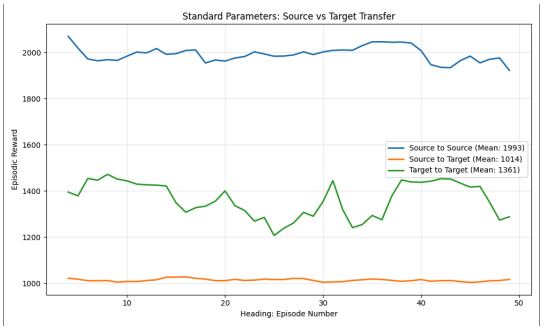

Fig. 4. Optuna parameters: Source vs Target Transfer

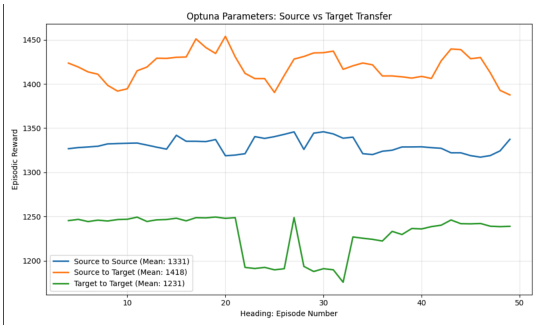

Fig. 5. Standard parameters: Source vs Target Transfer

C. Uniform Domain Randomization

We trained the agent in the randomized source environment for 1,000,000 time steps using the manually selected UDR hyperparameters (mass variations of 0.1 ,0.3 and 0.5). We tested the best performing agent (0.3) in the target environment for 50 episodes. The average return was 1540.53, resulting in a significant improvement compared to the lower bound of SAC trained on the source environment without UDR.

As we've seen, UDR is an adequate starting point in bridging the performance gap caused by Sim-to-Real. However, there are two crucial drawbacks:

- the high effort needed in manual tuning
- the environment is randomized from the beginning of training, instead of a gradual inclusion of randomized parameters

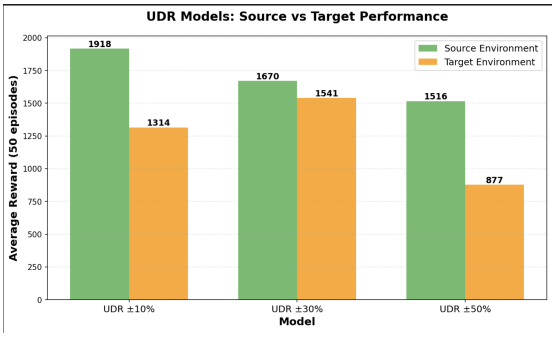

Fig. 6. Performance between different UDR models

D. Simulation Optimization

We can compare the final performance of our optimized SimOpt agent against the manual UDR baseline. Both agents were evaluated for 50 episodes on the target environment after completing their respective training regimes.

The manual UDR approach, with its fixed 0.3 variation, achieved a robust mean return of 1540.53. It proved to be a reliable method, consistently producing safe policies that transfer well. In comparison, the SimOpt-optimized agent achieved a mean return of 1401.14, but with a significantly higher maximum peak reward of 1753.74 .

As seen from the comparison SimOpt exhibits higher peaks but also greater variance compared to the steadier performance of UDR. This behavior aligns with the nature of the optimization process: SimOpt aggressively searches for high-reward configurations, potentially finding more agile but riskier policies. The fact that SimOpt surpassed the highest peak of any other model (including baselines and UDR) by a margin of nearly 200 points demonstrates its potential to discover superior policies that manual tuning might miss.

Comparing the two, UDR offers guaranteed safety straight out of the box, while SimOpt offers peak performance via automated search. SimOpt successfully improved upon the potential of the agent, unlocking dynamics that allow for higher scores, even if the average stability trails slightly behind the conservative UDR baseline. It can get more stable if we increase the number of trials and episodes per trail in the training.

VI. Discussion

The project began with a fundamental comparison between REINFORCE (with and without baseline) and Actor-Critic algorithms. In the initial experiments (Section 4.1), we observed that the variance inherent in REINFORCE—even with a baseline—made convergence slower and less stable compared to the Actor-Critic approach. This is expected, as Actor-Critic methods leverage a learned value function to reduce the variance of the policy gradient updates. The "critic" effectively stabilizes the learning process, allowing the agent to settle into effective behaviors more reliably than the Monte-Carlo based returns of REINFORCE.

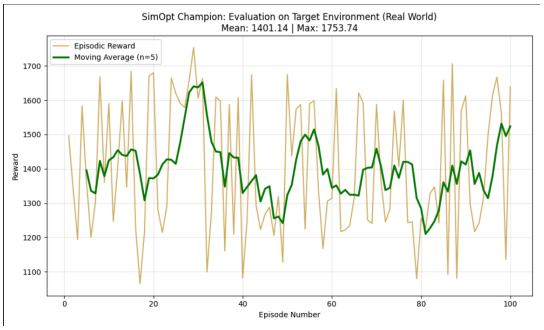

Fig. 7. Simopt Performance on Target Environment

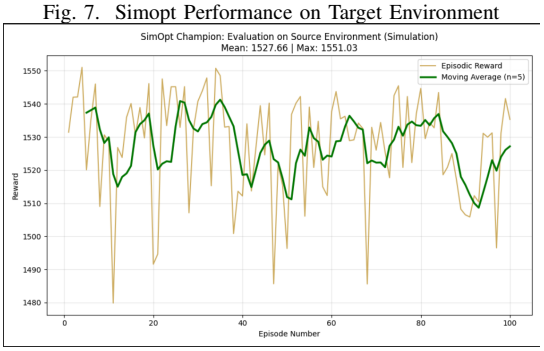

Fig. 8. Performance of Simopt on Source Environment

Building on the stability of Actor-Critic, I transitioned to using Soft Actor-Critic (SAC) for our domain adaptation experiments involving SimOpt (Simulation Optimization) and UDR (Uniform Domain Randomization). In this phase, I aimed to bridge the sim-to-real gap . I established two key performance benchmarks:

Lower Bound (Source to Target): The performance of an agent trained on the Source and tested on the Target .

Upper Bound (Target to Target): The performance of an agent trained and tested directly on the Target environment).

Also during the implementation of SAC , I found the best performing hyperparameter by training it on source environment and testing it periodically in the target environment. I implemented 50 trials , with each trial lasting for 2000 episodes that is 1 million time step unless it was pruned .

I expected better performance from Simopt in comparison to UDR , but the avg reward after testing for 50 episodes showed that UDR had more stable and greater average return.Still the maximum reward was greater for Simopt showing its potential to perform better.

Future Improvements for SimOpt can be achieved by Increasing Optimization Trials: The current SimOpt run was limited in the number of iterations (20 iterations). Increasing this would allow the Bayesian Optimization process to explore the parameter space more thoroughly and converge on tighter, more accurate bounds. Other solution can be to have more evaluations per episode . When tested for 100 episodes (graphs and value not shown here),I could see that Simopt was catching to UDR in average return as well.

VII. Conclusion

In this paper, I observed the performance of some of the main Reinforcement Learning algorithms, showing how increasing the complexity of the algorithms can yield much better results. I then saw how even more complex algorithms like SAC are not robust enough to adapt to a slightly different environment. I tried to bridge this gap through one of the most known techniques to address the Sim-to-Real gap: Domain Randomization. The UDR algorithm provided the expected results: a performance increase, but with difficult tuning to find the right parameters. The SimOpt algorithm provided results better than UDR, but I think it's possible to get an improvement through a few tweaks to our application of the algorithm, mainly giving it more trails to explore the space and more number of episodes per trial for greater training.

References

[1] RL MLDL 2025 Repository. https://github.com/pranavtripathi6844/Reinforcement_Learning

[2] Wenshuai Zhao, Jorge Pena Queralta, and Tomi Wester- ˜ lund. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. arXiv preprint arXiv:2009.13303, 2020.

[3] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018.

[4] 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," 2017.

[5] OpenAI et al., "https://spinningup.openai.com/en/latest/algorithms/sac.html

[6] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. Adaptive Computation and Machine Learning series. The MIT Press, second edition, 2018.

[7] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," 2019.