A sensitivity analysis approach to Domain Randomization

334015 Andrea Vasco Grieco
Politecnico di Torino
Turin, Italy
s334015@studenti.polito.it

331852 Valerio Xefteris
Politecnico di Torino
Turin, Italy
s331852@studenti.polito.it

331568 Paolo Castellaro
Politecnico di Torino
Turin, Italy
s331568@studenti.polito.it

***Abstract*—In this paper we applied Reinforcement Learning to solve the Hopper task. We compared several different algorithms to assess the different performances. In particular, we tested with more naive approaches (REINFORCE and Actor Critic), as well as state-of-the-art algorithms, i.e. Proximal Policy Optimization (PPO) or Soft Actor Critic (SAC). We then applied the Uniform Domain Randomization with different ranges to enable the model to adapt to a target scenario. Finally, we developed an experimental approach to select the most significant parameters to randomize, in order to improve the convergence speed of state-of-art randomization algorithms, whose performance is often affected by the high dimensionality of the parameter space. This approach is based on sensitivity analysis by exploiting Sobol indices and linear regression.**

I. Introduction

A. Reinforcement Learning

Reinforcement Learning (RL) is a branch of Machine Learning where an agent interacts with an environment and aims to learn optimal behaviour by maximizing cumulative rewards. The agent observes the current state s_t of the environment, takes an action a_t and then transitions to a new state s_{t+1} receiving a reward r_t. The main goal is to learn a policy $\pi(a|s, \theta)$ that maps states to actions in order to maximize the return of cumulative rewards, $G_t = \sum_{k=0}^{\infty} \gamma_k r_{t+k}$ where $\gamma \in [0, 1)$ is called discount factor. The reason for this factor is to prioritize immediate over future rewards. The general scheme of RL is represented by the following illustration:

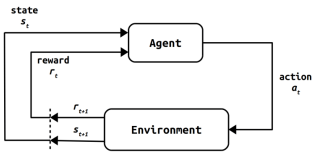


Fig. 1: Reinforcement Learning scheme

B. Hopper environment

The Hopper is a one-legged robot who is tasked to jump forward as long as possible without falling down. Its reward is the result of the sum of three components: the first one for being in a *healthy state*, the second one proportional to how much it goes forward, while the third one is a cost aimed at penalizing large actions, in order to achieve a smoother behavior.

The robot is composed of four main body parts and relative

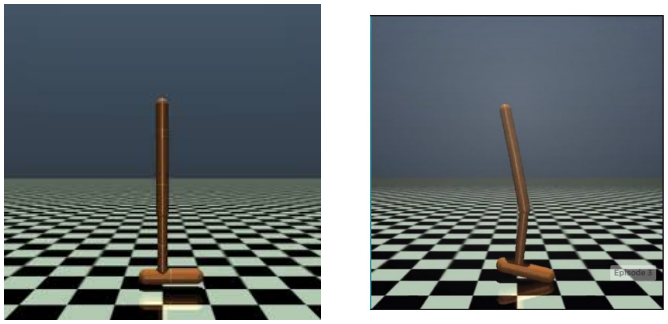


Fig. 2: The Hopper inside Mujoco env [1]

masses, called dynamics paramters. These are the torso at the top (mass m_0), the thigh in the middle (mass m_1), the leg at the bottom (mass m_2), and a single foot on which the entire body rests (mass m_3). The values of their masses can be seen in Table I.

TABLE I: Hopper Dynamics Parameters Values for source env

Parameter	m_0	m_1	m_2	m_3
Value	3.534	3.927	2.714	5.089

The Hopper environment has a continuous state space consisting of 11 dimensions. These correspond to parameters such as joint angles, velocities, and other system configuration aspects. Each element in the state space can take any real value. The action space is continuous too, with 3 dimensions corresponding to the torques applied to the Hopper's joints, These dimensions allow the agent to control its movement within the range [-1, 1] for each of them.
We will also refer to a *target* environment. This is identical to the one we just talked about (*source* environment) with the only difference that the mass of the torso m_0 is reduced by 1 (i.e. $m_0 = 2.534$), in order to simulate an unmodeled phenomenon.

II. REINFORCE Implementation

A. Theoretical Aspects

The REINFORCE [2] algorithm is a Monte Carlo method, applied to the classic policy gradient algorithm, where the agent collects samples of an episode using its current policy

and uses them to adjust the parameters θ of the policy itself. Let $\pi(a|s,\theta)$ be a differentiable policy parameterized by θ, that outputs an action a given some state s. Let $\alpha > 0$ be the step size (e.g.for the Adam Optimizer). The algorithm proceeds by generating episodes. Starting from an initial state S_0, the policy is used to sample an action A_0, to which the environment responds to by providing a reward R_1 and the next state S_1. This process is repeated until it reaches a terminal state S_{T-1} and a terminal reward R_T. After an episode is complete, for each time-step of the episode $t = 0, 1, ..., T-1$, the discounted return is computed as follows: $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$, where γ is the discount factor. Then, the policy parameters are updated as: $\theta_{t+1} \leftarrow \theta_t + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta(A_t|S_t,\theta)$.

B. Why the baseline?

REINFORCE can be quite volatile and unstable, due to the fact that it increases the probability of each action taken in a given state. In the long run, actions associated with higher returns are favored, but the method also risks increasing the probability of suboptimal actions (even if only by a small amount). This might be problematic since ideally we would want to decrease the likelihood of ineffective actions. A way to determine the effectiveness of an action is using a baseline b as a threshold. If the return is below this baseline then the action is considered bad.

C. Pseudocode and Network Implementation

We implemented the REINFORCE algorithm with two constant baselines $b = \{0, 20\}$, and also with a baseline obtained using the running average of the rewards of the last 1000 episodes, according to the following pseudo-code algorithm:

Algorithm 1: REINFORCE

Input: a differentiable policy $\pi(a \mid s, \theta)$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ (e.g., to 0)
Loop forever
  Generate an episode $S_0, A_0, R_1, \ldots, S_T, A_{T-1}, R_T$ following $\pi(\cdot \mid \cdot, \theta)$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$
    $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $G_t \leftarrow G_t - b$
    $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla \ln \pi(A_t \mid S_t, \theta)$
  end
end

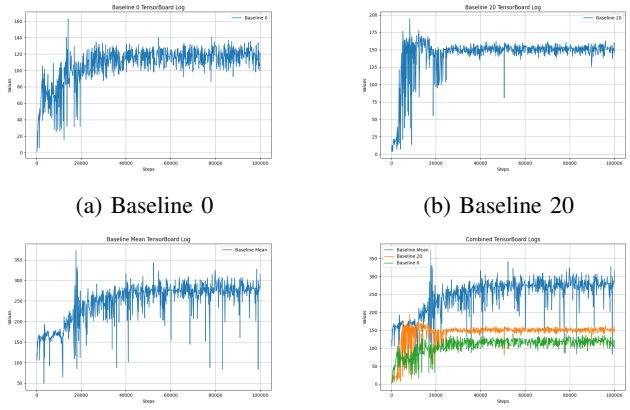
The neural network used consists of two fully connected layers of 64 neurons each and a mean output layer. The loss function is then defined as

$$L := -\sum_t [\log(\pi(a_t|s_t))(G_t - b_t)],$$

the gradient of which is then used to update the policy parameters by backpropagation.

D. Evaluation

We evaluated all three different REINFORCE baselines on 10^5 episodes. As expected from theory, we can see that the algorithms with the baselines outperformed the algorithm without it, while also being faster and more stable. In particular, the agent with no baseline shows higher variability and a generally more unstable learning throughout training, with reward values fluctuating significantly before stabilizing at a relatively low level ($\sim$125). In contrast, the agent with a constant baseline of 20 shows more stability and achieves a better final performance ($\sim$160), though it experiences sharp fluctuations early on before gradually improving. The running average baseline achieves the best performance, with the agent steadily reaching higher reward values ($\sim$300) and displaying the least variance throughout training. It should be noted that



(a) Baseline 0 (b) Baseline 20

(c) Baseline running average (d) Comparative Graph

Fig. 3: REINFORCE with different baselines

none of the policies implemented with REINFORCE were successful in completing the task, as evidenced by the low values of the rewards shown in the graphs; however, in the Hopper scenario, these policies enabled the agent to perform a few jumps.

III. Actor-Critic

The Actor-Critic [2] method combines policy-based and value-based approaches in reinforcement learning. This dual-network architecture consists of an actor, responsible for selecting actions, and a critic, that evaluates the actions by estimating the state values. In this second model, we implemented an Actor-Critic algorithm with the purpose of comparing its performance against the REINFORCE algorithm in terms of expected rewards, variance and convergence speed.

A. Our implementation

In our project two network configurations were tested: a single network containing both the forward module for the actor and the critic, and two separate networks for the actor and the critic, each with a different learning rate. Our observations suggest that on the one hand the configuration allowing distinct parameter modeling for the two networks aids in

better managing individual loss trends. On the other hand, the configuration that is most consistent with the theoretical foundations of the Actor-Critic approach is the one implementing a single learning rate, thus enabling parameter sharing between the networks. This approach has shown greater long-term stability in convergence. We conducted a fine-tuning process using Optuna [3], a Bayesian hyperparameter optimization framework, to determine both the optimal learning rate and the discount factor γ. By setting 50 as number of trials and 1,000 as number of training episodes per trial, our random search found the optimal values for the learning rate and γ to be, respectively, 0.0017 and 0.96. Concerning other potential alterations,the Actor and Critic networks maintained their original architecture, each comprising a single hidden layer with 64 neurons.

B. Policy Update

To update the policy, we computed the bootstrapped discounted returns, the temporal difference error (TD error), and the standardized advantages, adding a small ϵ to the standard deviation to ensure numerical stability.

- **Bootstrapped Discounted Returns (TD Target)**: The TD target ($\hat{R}_t$) incorporates the immediate rewards and the estimated value of the next state, adjusted by a discount factor (γ) and a done flag to identify and handle terminal states. It is computed as:

$$\hat{R}_t = r_t + \gamma V(s_{t+1})(1 - d_t)$$

  where r_t is the reward at time step t, $V(s_{t+1})$ is the estimated value of the next state, and d_t is the done flag, indicating whether the episode has ended. We found that maintaining a high value of γ more closely aligns with our objective of maximizing cumulative rewards. In fact, it effectively extends the horizon of future rewards considered in each update, thereby supporting better long-term decision-making in the learning algorithm.
- **Temporal Difference (TD) Error**: The TD error (δ_t) is the difference between the TD target and the current value estimate of the state. It measures the discrepancy between the predicted value and the observed return:

$$\delta_t = \hat{R}_t - V(s_t)$$

  where $V(s_t)$ is the estimated value of the current state.

The losses were then evaluated using the following formulas:

- **Actor Loss**:

$$\text{Actor Loss} = -\mathbb{E}\left[\log \pi(a_t|s_t;\theta) \cdot \delta_t\right]$$

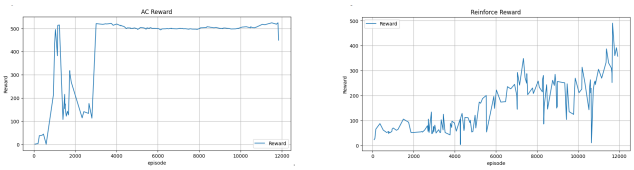
  where $\log \pi(a_t|s_t;\theta)$ represents the log probability of the action taken given the state and policy parameters.
- **Critic Loss**:

$$\text{Critic Loss} = \mathbb{E}\left[\left(V(s_t) - \hat{R}_t\right)^2\right]$$

It is important to note that we decided to use episodes as temporal metric for updates, although it is more common to use timesteps for the critic's update and episodes for the actor's update.

C. Evaluation

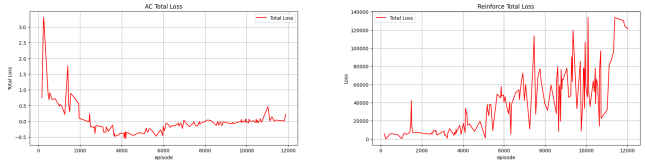
We decided to evaluate both REINFORCE and Actor-Critic over approximately 10,000 episodes and compare their performance.



(a) Actor-Critic Reward (b) Reinforce Reward

Fig. 4: Comparison between actor-critic and reinforce rewards

As shown in Fig. 4a and 4b, the variance during the training process is significantly higher for REINFORCE, while Actor-Critic demonstrates much greater stability after an initial exploration period. Additionally, the average reward achieved by Actor-Critic noticeably surpasses that of REINFORCE, although the difference is not excessive and as we've seen previously, Reinforce reaches a plateau in rewards much more slowly compared to Actor Critic.



(a) Actor-Critic Loss (b) Reinforce Loss

Fig. 5: Comparison between actor-critic and reinforce loss

The loss functions also exhibit distinct behaviors, with Actor-Critic showing more stable and fast convergence as expected (see Fig. 5a and 5b).

D. Testing

As we can see from Table II, we evaluated both models on a source environment and subsequently on a target environment over the course of 1,000 episodes. The results from both tests were well-aligned with the previous considerations. The Actor-Critic model lead to a lower standard deviation and achieved higher rewards compared to the REINFORCE model.

TABLE II: Model performance comparison

Model	Test Source	Test Target
Actor-Critic	Average: 511.72 Std: 9.58	Average: 502.36 Std: 9.36
REINFORCE	Average: 279.87 Std: 32.82	Average: 276.82 Std: 26.03

Remark. However, it is important to note that both models are still naive implementations and do not achieve the goal of fully learning the task.

IV. PPO

To enable our Agent to effectively learn the Hopper task, we chose a more robust and stable algorithm; Proximal Policy Optimization (PPO) [4], which is known for generally delivering solid results in reinforcement learning problems. For this step, we relied on the Stable-Baselines3 [5] library to implement the base algorithm.

A. A brief explanation of PPO

PPO enhances the performance of a standard policy gradient method by also considering the strength of the impact that each update has on the model's stability. If we examine its loss function :

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

we notice how the ratio of the current policy to the old policy is multiplied by the advantage factor. Intuitively, this means that:

- If $A(s,a) > 0$, the action taken is better than the expected action. So we would like the probability of selecting that action to be increased. However, if $r_t(\theta)$ is too large, this could lead to aggressive updates, introducing instability.
- If $A(s,a) < 0$,the action taken may not be as good and consequently we want the probability of selecting that particular action to be decreased. But if $r_t(\theta)$ is too low in this case (i.e. we are significantly decreasing the probability that that action will be reselected), this could still lead to drastic updates and therefore instability.

By minimizing the PPO loss function we are considering the safest update possible while ensuring improvements in performance. To enhance the algorithm's stability, a clipping mechanism was introduced for the policy ratio, parameterized by the hyperparameter ϵ. This ensures that policy updates are gradual and controlled, enabling the agent to explore its environment effectively while maintaining a stable learning trajectory.

B. Our Implementation

For our custom PPO implementation, we employed a three-layer multilayer perceptron (MLP) with ReLU activations for both the value function and policy networks. Each hidden layer comprised 64 neurons. Regarding hyperparameters, we utilized the following settings: learning rate of 0.00028, discount factor of 0.998, clip range of 0.22, batch size of 64, epochs of 10, and number of steps of 2048. Notably, we conducted fine-tuning for the learning rate, discount factor, and clip range using Optuna, training each trial for 700,000 timesteps. This decision was motivated by our observation that the base algorithm achieved satisfactory results around this timestep threshold.

C. Evaluation

We decided to evaluate the performance of PPO on the Hopper task by training the model with the new parameters for 5,000,000 timesteps to reach a plateau in the expected reward and stabilize the results. Additionally, we evaluated the model using the best-performing model, as determined by the highest expected reward obtained during training via custom checkpoints. The values of the average reward and the relative standard deviation are shown in Table III.

TABLE III: Model performance

Test Source-to-Source	Test Source-to-Target	Test Target-to-Target
Average:1656.23 Std:176.16	Average:1080.00 Std:246.32	Average:1525.98 Std:164.51

The results align with the theoretical expectations regarding the model's performance, leading to greater stability compared to the naive algorithms previously discussed and significantly improving performance. These results fall perfectly within the expected reward range presented in the algorithm's documentation (1567 $\pm$ 339 for the Hopper task in a sim-to-sim scenario [5]).

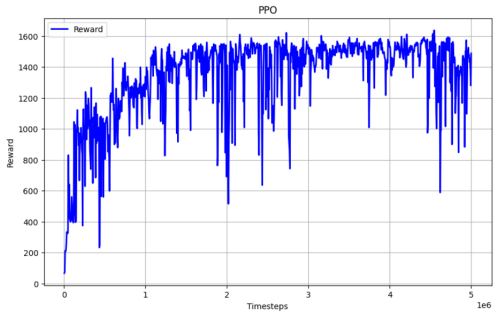


Fig. 6: PPO cumulative reward

As we can see, the variance in the outcomes is still significant, probably due to the stochastic nature of the action sampling. However the results from the source-to-target experiments clearly demonstrate the model's inability to generalize effectively from the source to the target domain, resulting in a substantial performance decline when tested on it. Given these considerations, the use of domain randomization is not only appropriate, but essential to improve performance, as our results will show.

V. Uniform Domain Randomization (UDR)

Domain Randomization is a technique used in reinforcement learning, in particular to solve the sim-to-real transfer problem. The main idea behind it is to expose the agent to a wide range of different simulated environments during training by randomizing some, or all, the simulation parameters. In particular, in the Uniform Domain Randomization [6] these parameters are sampled uniformly within a predefined range. The main goal is to help the agent to learn more robust policies, that generalize better to unseen or unexpected real-world conditions. Different parameters can be randomized, such as the environmental ones (e.g. gravity and friction) or, as in our experiments, the dynamics parameters (e.g. masses). The choice of the range used to sample is not trivial and it is problem dependent. In our case the range that gave the best results was 3, thus every mass m for our UDR

was uniformly sampled from a minimum of $m - 3$ to a maximum of $m + 3$. This ensures that the agent experiences a wide spectrum of possible environments without focusing too much on certain values, encouraging it to learn policies that are robust to changes or uncertainty in the environment. For this reason, it makes the agent less sensitive to small variations in the real-world scenarios that are difficult to reproduce in a virtual simulation. A possible downside of using UDR comes from the curse of dimensionality. In fact, as the number of randomized parameters increases or the range is wider, it becomes computationally expensive to train across such a plethora of environments. In fact, sampling in high-dimensional spaces is quite ineffective.

A. Range tuning and Evaluation

A grid search methodology was employed to determine the appropriate range of mass variations for our domain randomization technique. By systematically varying masses by factors of 1 to 5, we assessed the impact on UDR performance. To ensure physically meaningful results, a lower bound of 0.5 was imposed on mass values thus avoiding negative values. Based on these experiments, a variation factor of 3 was selected as the optimal setting. The results are in Table IV.

TABLE IV: UDR ranges performance

Model	Test Source-to-Target
range 1	Average: 660.58 Std: 174.85
range 2	Average: 903.63 Std: 127.19
range 3	Average: 1558.85 Std: 287.85
range 4	Average: 970.20 Std: 189.34
range 5	Average: 1058.31 Std: 219.17

As our findings demonstrate, the incorporation of UDR into the PPO algorithm enables it to achieve performance levels comparable to those of a PPO model trained exclusively on the target domain. While there is still some increased variance in the mean reward, likely due to the model's exposure to highly divergent parameterizations, it is clear that the model has successfully adapted and generalized to the new environment.

VI. Selection of the most important parameters

In this part we tried to solve a frequent problem in state-of-the-art algorithms of domain randomization such as DROPO. This is given by the high dimensionality of the parameters space. In fact, when adaptive domain randomization methods are used, their performance can be affected by the number of the parameters on which the randomization takes place. Our goal is to try to find a method to choose a subset of n parameters (with n given).

A. General description of the method

We tried to solve the problem by means of the sensitivity analysis. Specifically, the goal was to find which parameters affected the performance the most within the simulation, to conclude that these were the most important to be randomized. Our intuition was that the randomization of such parameters was more likely to be effective in compensating for an unmodeled phenomenon. For instance, one could imagine to add random noise parameters to the values, which obviously do not affect the model. At the very minimum, we want our model to be able to identify these parameters and avoid trying to optimize them. Similarly, if other values do not affect the simulation much we want a lower score, to indicate such a behavior.
Our setting supposes that there is access to a trained model, which successfully solves the task and that the simulation adopted is sufficiently close to the reality. Another important assumption is that we have an algorithm capable of reliably learning a task: that is, its general behavior must be consistent after each training.

B. Sensitivity analysis

Sensitivity analysis can be used to refer to different concepts depending on the setting. However, for our purposes we can think of it as 'The study of how the uncertainty in the output of a model (numerical or otherwise) can be apportioned to different sources of uncertainty in the model input', according to [7]. In particular, we considered the factor fixing (FF) and the factor prioritization (FP) settings. In the former one, the goal is to find the subset of indices which, when fixed, reduce the variance the most. Conversely, in the second setting, we try to rank the importance of the factors directly, simply by choosing the parameters who influence the model the most. However, the methods used for FP do not typically consider interactions, and tend to work well mainly on linear models. For the FF case we used the Sobol' sensitivity indices, while for the the FP we simply adopted a linear regression and the Sobol' first order indices. Notice that the linear regression is quite naive and it is not meant as an effective solution but simply as a baseline. We will now briefly introduce the two algorithms.

1) Linear Regression: the linear regression tries to fit the coefficients of a linear function to best approximate a function $f\colon \mathbb{R}^n \to \mathbb{R}$. Let $\mathcal{S} \coloneqq \{x_1, \dots, x_m\}$ be a sample of m records such that $x_i \in \mathbb{R}^n$ for every $i \in \{1, \dots, m\}$. Let $y_i \in \mathbb{R}$ be the response variable corresponding to the i-th record, meaning that $y_i = f(x_i)$. Then a linear regression is the function $g(x) = b^* + w^* \cdot x$, where the bias $b^* \in \mathbb{R}$ and the vector of the weights $w^* \in \mathbb{R}^n$ satisfy:

$$w^*, b^* = \operatorname*{argmin}_{w \in \mathbb{R}^n, b \in \mathbb{R}} \sum_{i=1}^{n} (y_i - b - w \cdot x_i)^2.$$

2) Variance-based SA: A more accurate way to study the relevance of the parameters is given by variance-based methods. In particular, let $Y = f(X)$ for some random vector X of dimension n. Then, we can study the importance of the i-th component of X, by considering the first order sensitivity index for the i-th component

$$S_i = \frac{V(\mathbb{E}[Y|X_i])}{V(Y)}$$

Other important quantities are the total indices S_{Ti} corresponding to the i-th component

$$S_{T_i} = 1 - \frac{V(\mathbb{E}[Y|X_{-i}])}{V(Y)}$$

where $X_{-i} = (X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n)$ is the vector of dimension $n-1$ whose components are the random variables of X excluding the i-th. Another important, more general, quantity is given by the indices corresponding to the components $i_1, \ldots, i_k$ of X

$$S_{i_1,\ldots,i_k} = \frac{V(\mathbb{E}[Y|X_{i_1,\ldots,i_k}])}{V(Y)}.$$

Let us briefly explain the meaning of such quantities. Observe that, whenever we want to assess the importance of a variable we can fix it and observe the extent by which the value of Y drops. Therefore we want to chose the value of i which minimizes $\mathbb{E}[V(Y|X_i)]$. Since $V(Y) = \mathbb{E}[V(Y|X_i)] + V(\mathbb{E}[Y|X_i])$ and $V(Y)$ is independent of i, minimizing $\mathbb{E}[V(Y|X_i)]$ is the same of maximizing $V(\mathbb{E}[Y|X_i])$ and therefore S_i. Therefore, this is a measure of how much fixing X_i affects the performance. Conversely, the total indices S_{T_i} are a measure of the remaining variance if we fix all the variables except for the i-th. A similar reasoning can be done for $S_{i_1,\ldots,i_k}$: they measure the variance remaining after we fix the indices $i_1, \ldots, i_k$. Notice that the total indices do not take into account any interaction and can only be reliably used to perform FP. The meaning is to assign the biggest value, in terms of ranking, to the indices whose variance is greater when everything else is fixed.

However, these quantities cannot be easily computed directly as they would require the knowledge of the underlying probability distribution, which in our case is not known. An effective way of approximating them is via Monte Carlo (MC) and quasi-Monte Carlo (QMC) methods. The second ones have better convergence properties than the first ones since they are based on low-discrepancy sequences, thus making them more suitable for high dimensional spaces, reducing the problems of the curse of dimensionality. This means that given a volume $\mathcal{V}$ they aim at finding a sequence $\mathcal{S}$ of N points such that the discrepancy has low values. The formula for the discrepancy is given by

$$\sup_{\mathcal{J} \subseteq \mathcal{V}} |\operatorname{card}(\mathcal{S} \cap \mathcal{J}) - \operatorname{Volume}(\mathcal{J}) \cdot N|$$

where $V(\mathcal{J})$ is the volume of the subset $\mathcal{J}$ of $\mathcal{V}$. Intuitively, it tries to ensure that the points are equally distributed over $\mathcal{V}$. Suitable sequences are the Latin Hypercube sequence (LHS) and the Sobol sequence.

C. Sensitivity analysis for the Hopper task

We chose the same target environment as in the previous sections. Recall that it was obtained by decreasing the torso mass by one with respect to the source environment. We started by training a model with the PPO algorithm using $5 \cdot 10^6$ timesteps. We then computed the linear regression and the sensitivity indices. To do so, we chose the function $f \colon \mathbb{R}^5 \to \mathbb{R}$ which assigns the value of the reward function achieved by a policy with parameters tuple $(m_1, m_2, m_3, m_4, m_5)$. The first three masses are the ones corresponding, respectively, to *thigh, leg* and *foot*. The other two are simply noise masses, sampled in the range $[1, 7]$. These last ones do not affect the model in any way, and we use them to verify whether our models correctly identify them as irrelevant. To compute the sensitivity indices and the regression coefficients we assigned values sampled in the range of $[\max\{m_i^* - 3, 0.1\}, m_i^* + 3]$, where m_i^* is the original mass in the source environment. The reason behind the $\max$ is simply to ensure that all the masses are positive. To reduce the number of samples needed we chose to sample according to QMC methods. In particular, we chose a Latin Hypercube sampling strategy for the linear regression, and we used the Sobol sequence from [8] and computed the indices with Saltelli's improvement [9] for the sensitivity indices, adopting a radial sampling. Recall that Sobol' sequence requires 2^k samples for some $k \in \mathbb{N}$ to cover the space uniformly. Finally, we would expect the performance of the model to increase as each variable m_i gets closer to the center of the interval, corresponding to the value of training environment. Vice versa, it should be lower on the left and right part of interval. This was indeed the case, as shown in Figure 7a. To perform the linear regression we tried to linearize the model by using the distance of the mass m_i from the original quantity m_i^*. That is, we used as data the parameters $\delta_i := |m_i - m_i^*|$. This was quite effective in achieving our goal, as shown in Figure 7b. We would therefore expect to find negative coefficients for all the relevant parameters while they should be around 0 when irrelevant. Moreover, the importance of the parameters should be proportional to the absolute values of their coefficients, ordered from highest to lowest.

(a) (b)

Fig. 7: Performance when randomizing only the first mass (corresponding to thigh)

We also adopted some measures to optimize the speed of the process we took several decisions. Our first try consisted in simply taking an average of 100 runs for each tuple of masses in order to evaluate the model's performance. However, this resulted in really long computation times while computing sensitivity indices, due to the way Sobol's sequence is computed. To address this we first decided in favor of using a deterministic policy instead of the normal stochastic one. With a deterministic policy the agent always chooses the most likely action instead of sampling according to the

probabilities on which it is trained. This makes it ineffective after performing domain randomization, but it significantly reduces the variability of the reward. By doing so, we were able to only require one evaluation per mass tuple, significantly reducing the risk of encountering non-significant performance due to the sampling of less likely actions. Furthermore, since the model behaved always according to the same policy it made the comparison between scenarios more reliable. The second improvement we added consists in computing the evaluation runs in parallel, splitting the samples in batches. These would then be assigned to a different process each, so that the work would be evenly split. Since CPUs on personal computers allow to have about 10 parallel processes we managed to significantly reduce the required time. Overall, we were able to reduce the computation time by a factor of about 1000 compared to our first attempt.

Finally, to evaluate the effectiveness of this ranking we trained UDR randomizing each one of the masses by itself and finding the average performance, as well as all the couples. If the ranking was correct we would expect the models trained by randomizing the masses with higher scores according to our FP and FF methods to have a better performance on the target environment.

D. Results

The results of the regression and sensitivity analysis can be found in Table V. The column of UDR implies the performance on the target environment by randomizing only the i-th mass with a range of 3 as in the previous chapter.

The linear regression was performed on 100000 samples,

Param.	Regression coeff.	$\mathbf{S_i}$	$\mathbf{S_T}$	UDR
m_1	-165.27 ± 3.58	0.448 ± 0.03	0.858 ± 0.02	1209
m_2	-73.47 ± 3.74	0.050 ± 0.02	0.453 ± 0.01	912
m_3	-61.7 ± 3.58	0.072 ± 0.02	0.412 ± 0.01	884
m_4	0.79 ± 3.58	0.002 ± 0.01	0.230 ± 0.01	-
m_5	-0.005 ± 3.58	-0.011 ± 0.01	0.231 ± 0.01	-

TABLE V: Results of the tests, the intervals correspond to a probability of 95%

even though similar results were found with significantly less observations. First, notice that it correctly identified the noise masses (m_4 and m_5) by assigning coefficients close to 0. Moreover, we can see that it correctly identified the most important parameters and the ranking correctly predicts the performances of the UDR. However, there is no reason to assume that the model would be linear in its behavior. In fact, even assuming the differentiability of f, which would guarantee some local linearity, that would be a unreasonably strong assumption. This is indeed confirmed by the low value of R^2 that we found, corresponding to approximately 0.097.

Let us now analyze the results of the sensitivity analysis. We can see that both the total and the first order indices correctly predict the first mass to be the most relevant. Then, we can see that the noise masses are assigned the lowest value, the corresponding S_T are not zero. Notice that this behavior is expected, since the model is affected by some unpredictability due to the randomization of the starting point in Mujoco. For this reason, the numerator of the fraction, given by $V(\mathbb{E}[Y|X_{-i}])$ is not 0 and these values are effectively measuring what portion of the variance $V(Y)$ remains after fixing all the parameters. However, the ranking is still correct since the noise parameters received the lowest values.

The first order indices swapped the ranking of masses m_2 and m_3. This is not a big problem though, as they predicted correctly that these masses would not be significant, by assigning them a value of approximately 0. Furthermore, the results obtained with UDR are not exactly reliable between different trainings and they are affected by a high variance, so the ranking can still be considered effective.

We also computed the second-order sensitivity indices $S_{ij}^{(2)}$. Given indices i and j these are computed by

$$S_{ij}^{(2)} \coloneqq S_{ij} - S_i - S_j.$$

These measure the relevance of the interaction between parameters i and j, net of their respective variances. If $N = 3$ as in our case it is trivial to show that

$$S_{ij}^{(2)} = 1 - S_{T_k} - S_i - S_j$$

where i, j and k are distinct indices and correspond to the three different parameters. However, we did not adopt this last method to compute them, but obtained them by direct calculations. The results are in Table VI. Notice that even if we omitted them here for brevity, also the interactions between all the noise masses and the real ones were coherently close to 0. We can see here that also in this case Sobol indices

Params	$\mathbf{S_{ij}^{(2)}}$	UDR
(m_1, m_2)	0.081 ± 0.036	1293
(m_1, m_3)	0.038 ± 0.035	1202
(m_2, m_3)	0.018 ± 0.027	698

TABLE VI: Results of the second order parameters and comparison with UDR performed only on the two parameters. The intervals correspond to a probability of 95%.

were quite informative. In fact, they correctly predicted that the interactions between all the masses taken in couples would be small and this is in fact the case. Notice that sometimes the performance (especially for masses m_2 and m_3) is lower than in the previous tests. However, we do not consider this variation to be significant, as the training of the model is not always exactly reliable and it is subject to high variance.

E. Comments

Our results show that sensitivity analysis is quite effective in predicting the most important variables to randomize. In particular, the linear regression did provide a reasonable ranking but it might not be suited for more complex problems, due to the limitations of such a simple model.

The first-order sensitivity indices are probably the most effective way of computing the importance of a single parameter when taken alone, since the way they are constructed is more robust. In our setting, based on sensitivity analysis alone,

one could develop a strategy to choose which parameters to randomize to be effective with their resources. This means that, it provides an overview of which parameters to randomize and what performance increase one could exepect. The choice could be done as follows: in case the resources were sufficient only to randomize one parameter, then the choice would be to choose only m_1. Then, the model tells us that there is no reason to train a two parameters, since the overall performance is not expected to increase significantly and it would only waste resources.

We did not compute the third order indices but we would expect this value, denoted by $S^{(3)}_{m_1 m_2 m_3}$, to be high. In fact, since our parameters are sampled independently, it holds true that

$$S_{T_i} = S_i + \sum_j S^{(2)}_{ij} + \ldots + S^{(n)}_{i_1 \ldots i_n}$$

meaning that in our case, ignoring the noise masses, we would get $S^{(3)}_{m_1 m_2 m_3} = S_{T_{m_1}} - S_{m_1} - S^{(2)}_{m_1 m_2} - S^{(2)}_{m_1 m_3}$. Therefore, we would get $S_{m_1 m_2 m_3} \approx 0.291$ which is a significant value. This correctly predicts the significant improvement we observed when randomizing all the three masses obtaining values above 1500.

F. Advantages and limitations

We will now analyze the advantages and limitations of this approach.

The main advantage consists in the fact that it is possible to perform the prediction with only one training and without prior randomization. This would allow to guide the randomization of robots whenever it is possible to use the same simulation, if they need to be applied in different settings. Moreover, if it is possible to know a number k of parameters that are suitable to randomize in advance one could simply perform sensitivity analysis to estimate $S_{i_1 \ldots i_k}$ for all the values.

There are two possible limitations. The first one being that computing $S_{i_1 \ldots i_k}$ is costly, since it requires $\binom{n}{k}$ sensitivity indices. The potentially high computational cost can however be reduced by exploiting parallelization.

The second problem is one that were not able to tackle. Our assumption requires our RL model to be sufficiently reliable about the way it solves the task. This was approximately the case for the Hopper, even though some results were not exactly consistent. For instance, the source to target with no randomization we initially found higher values than with the randomization of masses 2 and 3, which is not what is expected and is the result of MC methods while training. However, there were at least comparable results but we cannot safely assume that this will always be the case.

VII. Conclusion

Throughout our paper we explored several approaches to reinforcement learning, applied to the Hopper task. We began with naive algorithms such as REINFORCE and Actor-Critic and then proceeded with a more sophisticated one, i.e. Proximal Policy Optimization (PPO). At this point, we solved the sim-to-sim problem by applying Uniform Domain Randomization. We then developed a new approach to determine the most important parameters, based on two different sensitivity analysis approaches. The most basic one adopted a linear regression and evaluated the parameters importance based on the absolute value of the coefficients. The other one adopted sensitivity indices to predict the amount of variance of the reward explained by each parameter. To test it we also introduced two noise variables, to show that our techniques were capable of identifying them as unimportant. We found significant results and the predictions made by this approach were reasonably accurate.

Further research is still required to improve this technique, possibly to test it with other sensitivity analysis algorithms, in order to find the most effective ones. In fact, the computation of these indices can still be quite computationally expensive, even if it is more feasible than making multiple trainings, with the corresponding randomizations. For this reason, a possible future development is to find a reliable approach to reduce the number of samples required.

It should also be noted that the Hopper only has 4 parameters, one of which was kept fixed. Further research should also verify whether the proposed approach can be applied or adapted to more complex systems, and identify the possible limitations.

A full code implementation can be found here.

References

[1] Gym Library, "Hopper environment," 2023.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, second edition ed., 2020.

[3] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2623–2631, 2019.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[5] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[6] 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)*, pp. 3803–3810, IEEE, 2018.

[7] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto, *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. USA: Halsted Press, 2004.

[8] S. Joe and F. Y. Kuo, "Constructing sobol sequences with better two-dimensional projections," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2635–2654, 2008.

[9] A. Saltelli, P. Annoni, I. Azzini, F. Campolongo, M. Ratto, and S. Tarantola, "Variance based sensitivity analysis of model output. design and estimator for the total sensitivity index," *Computer Physics Communications*, vol. 181, no. 2, pp. 259–270, 2010.

[10] I. Sobol', "Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates," *Mathematics and Computers in Simulation*, vol. 55, no. 1, pp. 271–280, 2001. The Second IMACS Seminar on Monte Carlo Methods.