Sim-to-Real Transfer in Reinforcement Learning with Bayesian Domain Randomization

Tommaso Massaglia
s292988@studenti.polito.it

Farzad Imanpour Sardroudi
s289265@studenti.polito.it

Alireza Talakoobi
s289641@studenti.polito.it

***Abstract*—The main goal of reinforcement learning is to learn a policy that is able to correctly solve a task by providing to the algorithm some information about the environment the agent is operating into, the current state of the agent and a reward function; considering this task, it is evident how training a robot control policy that is able to operate in the real world would pose an issue in data collection as typically that data is very expensive to collect. To solve this issue the literature and studies of recent years moved towards bridging the so called 'reality gap', in practice, training an agent only -or mostly- in simulation and getting as output a policy that is transferable to the real world, of which one of the most efficient methods in solving it resulted to be domain randomization. After giving an introduction of the landscape surrounding the reinforcement learning problem, we will focus on one of the state-of-the-art approaches to solve the reality gap by analyzing the so called Bayesian Domain Randomization (BayRn), which outputs a more robust policy by optimally randomizing the environment.**

Source code can be found here

***Index Terms*-Reinforcement Learning, Sim2Real, Gaussian Processes, Bayesian Optimization, Mujoco, Hopper**

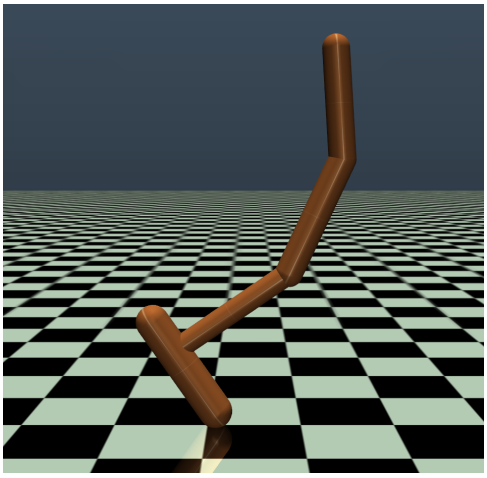

Fig. 1. The Hopper environment

I. Introduction

When we think about learning by interaction the idea that comes to mind is how, by interacting with the surrounding environment in many different ways -such as an infant touching by hands whatever surrounds him-, we form connections and 'rules' in our brain that allow us to reach a goal or effectively performing an action such as walking. Given this premise, it seems apparent that exploiting neural networks to replicate this process in a goal oriented fashion would be a possible approach to solve certain tasks such as robot control or autonomous driving. In this context, ***Reinforcement Learning*** *is learning what to do—how to map situations to actions-so as to maximize a numerical reward signal [1].*

What's different between Reinforcement Learning and other learning approaches -such as genetic algorithms- is that the learner is never told what actions to take, but rather he must discover by experiencing in first person the environment what actions allows him to maximize the obtained reward in a given task; what's fascinating is that the learner in complicated scenarios learns to give weight to actions that don't maximize the immediate rewards, but rather allow the learner to maximize the future reward, granting an overall better result.

Going back to how we described learning starting from the way humans learn, one of the main way we learn is by repetition, and this translates over to the way machines learn: one of the best way for an agent to learn a successful policy is by repeating it over and over again in order to better understand the possible future reward and the intricacies of the task at hand. This can be an issue though: the ultimate goal of applying a policy to the real world means that real world data is very valuable in the learning process, but at the same time learning and collecting data in the real world is extremely expensive and unpractical -imagine trying to train an autonomous driver on real cars-. The solution that the scientific community came up with was to train the policy as much as possible on simulation to exploit the inexpensiveness and availability of the data; with this another issue arose though, the so called *reality gap*, that is no matter how close our simulation environment is to the real world it will never be as detailed and varied as the real world itself.

To close the reality gap one of the best performing approaches that are available is *Domain Randomization* [3], which consists in training the agent on a multitude of randomized environments, so that when the policy is applied to the real world, the real world is nothing more than another simulation scenario to perform the task into. Most of the state-of-the-art approaches commonly randomize the simulator according to static handcrafted distribution. As an improvement over common domain randomization, in this paper we implemented Bayesian Domain Randomization [7], an approach that tries to close the reality gap by exploiting a

Gaussian process from which to sample the hyper-parameters that are than used to generate the distributions underlying the random domain parameters in a black box fashion; the Gaussian process is improved based on real world returns, meaning that the simulation gets closer and closer to reality as the training goes on, without sacrificing the randomness that allows for transferability between processes.

The reminder of the paper is organized as follows: first we go over the related works to get the required references for further explanations (Section II), then we'll go over the methods used to solve the reinforcement learning task (Section III), then over the proposed improvement over the common approach of domain randomization (Section IV) and to finish we'll compare results between approaches (Section V) and comment on them (Section VI).

II. Related Works

In this section we will talk about related works in the area of Domain Randomization with Adaptive Distributions of which BayRn is one.

A. Meta-algorithm which is based on a bi-level optimization problem

Ruiz et al. [4], proposed this meta-algorithm which is highly similar to BayRn. However there are two major differences: first, BayRn uses Bayesian optimization on the acquired real-wold data to adapt the domain parameter distribution, whereas *learning to simulate* updates the domain parameter distribution using REINFORCE. Second, the approach in [4] has been evaluated in simulation on synthetic data, except for a semantic segmentation task. Thus, there was no dynamics dependent interaction of the learned policy with the real world.

B. Active Domain Randomization (ADR)

This approach formulates the adaption of the domain parameter distribution as a RL problem where different simulation instances are sampled and compared against a reference environment based on the resulting trajectories. This comparison is done by a discriminator which yields rewards proportional to the difficulty of distinguishing the simulated and real environments, therefore providing an incentive to generate distinct domains. Using this reward signal, the domain parameters of the simulation instances are updated via Stein Variational Policy Gradient. Mehta et al. [5], evaluated their method in a sim-to-real experiment where a robotic arm had to reach a desired point. The strongest contrast between BayRn and ADR is they way in which new simulation environments are explored. While BayRn can rely on the well-studied Bayesian Optimization with an adjustable exploration-exploitation behavior, ADR can be fragile since it couples discriminator training and policy optimization, which results in a non-stationary process where distribution of the domains depends on the discriminator's performance.

C. Bayessim: Adaptive domain randomization via probabilistic inference for robotics simulators

In Ramos et al. [6], likelihood-free inference in combination with mixture density random Fourier networks is employed to perform a fully Bayesian treatment of the simulator's parameters. Analyzing the obtained posterior over domain parameters, Ramos et al. showed that BayesSim is, in a sim to sim setting, able to simultaneously infer different parameter configurations which can explain the observed trajectories. The key difference between BayRn and BayesSim is the objective for updating the domain parameters. While BayesSim maximizes the model's posterior likelihood, BayRn updates the domain parameters such that the policy's return on the physical system is maximized. The biggest advantage of BayRn over BayesSim is its ability to work with very sparse real-world data, i.e. only the scalar return values. return values.

III. Method

A. Markov Decision Processes [2]

The way we formalize a Reinforcement Learning problem is by using Markov Decision Processes. An MDP is a discrete-time stochastic control process that models decision making and describes it by using:

- $\mathbf{S}$, the sate space
- $\mathbf{A}$, the set of actions an agent can perform, called action space
- $P_a(s, s')$ which is the set of probabilities that a certain action a at time t will lead to state s' at time $t + 1$
- a reward function $Ra(s, s')$

A policy function π is a mapping from state space to action space. The goal of Reinforcement Learning is to train a policy such that it maximizes the expected (discounted) return, gained through the reward function, by adapting its policy parameters.

B. Introduction to Reinforcement Learning and Policy Gradient:

As previously said, the agent in reinforcement learning acts inside an environment. A model, which we might or might not know, determines how the environment responds to particular behaviors. The agent has the option of staying in one of the several environmental states or switching to a different state by executing one of the available actions. Transition probabilities between states determine which state the agent will enter. The environment provides a reward as feedback once an action is done. The agent's policy offers direction on the best course of action to pursue in a certain state in order to maximize overall rewards. Each state has can be valued via a value function that predicts the anticipated quantity of future rewards the agent may expect to obtain in that state by following the appropriate policy. The value function, in other terms, measures how good a state is. In reinforcement learning, we want to learn both policy and value functions.

Reinforcement Learning methods are categorized in many different ways one of them is splitting them into on-policy and off-policy categories.

- **On Policy** To train the algorithm, samples from the intended policy or deterministic results are used.
- **Off Policy** Instead of training on the distribution produced by the target policy, training is done on a distribution of transitions or episodes produced by a different behavior policy.

The following methods are all on-policy, **Policy Gradient** [1] methods: a subset of reinforcement learning techniques, policy gradient methods use gradient ascent to optimize parameterized policies in terms of expected return (long-term cumulative reward). They do not suffer from many of the issues that have marred conventional reinforcement learning methods, such as the absence of guarantees of a value function, the intractability issue brought on by ambiguous state information, and the complexity coming from continuous states and actions.

C. REINFORCE

Reinforce (Monte-Carlo policy gradient) relies on an estimated return by Monte-Carlo methods using episode samples to update the policy parameter θ. Reinforce works because the expectation of the sample gradient is equal to the actual gradient (Policy Gradient Theorem):

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)] \\ &= \mathbb{E}_\pi[G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)]\end{aligned} \tag{1}$$

As a result, we may gather data from the actual sample trajectory data and utilize it to update our policy gradient. Because it depends on a whole trajectory, it is a Monte-Carlo approach. The process is pretty straightforward:

1) Initialize the policy parameter θ at random.
2) Generate one trajectory on policy: $S_1, A_1, R_2, S_2, A_2, \ldots, S_T$
3) For t=1, 2, … , T:
   a) Estimate the the return;
   b) Update policy parameters: $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)$

D. Actor Critic

The policy model and the value function are the two primary elements of the policy gradient. Learning the value function in addition to the policy makes a lot of sense since doing so would help with updating the policy. For example, the Actor-Critic technique reduces gradient variance in vanilla policy gradients. Two models —two actor-critic— can potentially share parameters:

- **Critic:** updates the value function parameters w and depending on the algorithm it could be action-value or state-value.
- **Actor:** updates the policy parameters θ for $\pi_\Theta(a|s)$, in the direction suggested by the critic.

E. TRPO [8]

We should avoid parameter changes that drastically alter the policy in one step in order to increase training stability. By imposing a KL divergence limit on the size of the policy update at each iteration, Trust Region Policy Optimization (TRPO) implements this concept.

$$\begin{aligned}J(\theta) &= \sum_{s\in\mathcal{S}} \rho^{\pi_{\theta_{\text{old}}}} \sum_{a\in\mathcal{A}} \big(\pi_\theta(a|s)\hat{A}_{\theta_{\text{old}}}(s,a)\big) \\ &= \sum_{s\in\mathcal{S}} \rho^{\pi_{\theta_{\text{old}}}} \sum_{a\in\mathcal{A}} \big(\beta(a|s)\frac{\pi_\theta(a|s)}{\beta(a|s)}\hat{A}_{\theta_{\text{old}}}(s,a)\big) \\ &= \mathbb{E}_{s\sim\rho^{\pi_{\theta_{\text{old}}}},a\sim\beta}\big[\frac{\pi_\theta(a|s)}{\beta(a|s)}\hat{A}_{\theta_{\text{old}}}(s,a)\big]\end{aligned} \tag{2}$$

where Θ_{old} is the policy parameters before the update and thus known to us; $\rho^{\pi\Theta_{old}}$ is defined in the same way as above; $\beta(a|s)$ is the behavior policy for collecting trajectories. Noted that we use an estimated advantage $\widehat{A}(.)$ rather than the true advantage function a because the true rewards are usually unknown.

F. PPO [9]

PPO is a first-order optimization that simplifies its implementation. Similar to the TRPO objective function, it defines the probability ratio between a new policy and an old policy called r(θ).

$$r(\Theta) = \frac{\pi_\Theta(a|s)}{\pi_{\Theta old}(a|s)} \tag{3}$$

Also the objective function of TRPO will be updated as follow:

$$J(\Theta)^{TRPO} = E[r(\Theta)\widehat{A}_{\Theta old}(s,a)] \tag{4}$$

Without adding constraints, this objective function can lead to instability or slow convergence rate due to large and small step size update respectively. Instead of adding complicated KL constraint, PPO imposes policy ratio, r(Θ) to stay within a small interval around 1. That is the interval between 1-ϵ and 1+ϵ . ϵ is a hyperparameter and in the original PPO paper, it was set to 0.2. Now we can write the objective function of PPO.

$$\begin{aligned}J^{\text{CLIP}}(\theta) = \mathbb{E}[\min(&r(\theta)\hat{A}_{\theta_{\text{old}}}(s,a), \text{clip} \\ &(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_{\theta_{\text{old}}}(s,a))]\end{aligned} \tag{5}$$

G. UDR

Uniform Domain Randomization refers to the technique adopted in sim-to-real transfer learning where the training environment is randomized by sampling the domain parameters from handcrafted or arbitrary distributions (usually uniform). The process that is described in [3] consists in determining a set of parameters that we want to randomize ξ_i and to then uniformly sample each parameter within the range $\xi_i \in [\xi_i^{low}, \xi_i^{high}], i = 1, ..., N$. The result of this randomization is that the resulting policy is better suited to perform in the real world as during training it was exposed to a multitude

of different environments without *having time* to adapt to a specific one, thus giving as output a more robust policy.

The main issue within this approach is the choice of the bounds for each distribution parameter; the bounds $[\xi_i^{low}, \xi_i^{high}]$ are usually handpicked or determined after extensive testing on the real world, which for many scenarios is an unfeasible task. As such, the goal of recent years research was in optimizing the task of randomizing the environment in which to train the policy. In the next section we'll discuss one of these approaches whose main goal was to reduce the number of tests required on the real world.

IV. Improvement

Out of the many available methods present in the literature -as seen in section II- we chose to replicate the method applied in *Data-efficient Domain Randomization with Bayesian Optimization* [7] as the idea of using a black-box approach based in Bayesian statistics was appealing for its interoperability between different problems.

A. Main idea and concepts

The main idea behind BayRn -short for Bayesian Domain Randomization- is to create an approach that aims at learning a policy in simulation such that it is transferable to a real world robot; this is performed by collecting data over a distribution over simulators and by modelling through a Gaussian Process a connection between the domain parameters distribution and the real world. The key concepts that take part in this are:

- **Bayesian Optimization** is a global optimization strategy which tries to optimize a black box problem by exploiting a probabilistic model -usually a Gaussian process- to decide where to evaluate the function next with the goal of maximizing the so called "acquisition function;
- **Gaussian Processes** are distributions over functions defined by a prior mean and a kernel that adapt to the input data and generate an average function from which the BO samples data to evaluate.

In practice, we define a number of hyper-parameters that describe the shape of the distributions that are used to sample domain distribution parameters, and we tune those parameters using a combination of a Gaussian process and Bayesian optimization.

B. Methodology

The way BayRn is carried on can be expressed in a bi-level formulation:

$$\phi^* = \underset{\phi \in \Phi}{argmax} J^{real}(\theta^*(\phi)) \tag{6}$$

$$\theta^*(\phi) = \underset{\theta \in \Theta}{argmax} \mathbb{E}_{\xi \sim v(\xi;\phi}[J(\theta, \xi)] \tag{7}$$

where the two equations, called upper level (6) and lower level (7) optimization problem, state the goal of finding the set of domain distribution parameters ϕ^* that maximizes the return on the real world target $J^{real}(\theta^*(\phi))$ when used to specify the distribution $v(\xi; \phi)$ during training in the source domain. Eq. (6) is our BO task, concerned with finding the optimal domain parameters distribution to use for training and gives as output a policy θ^* that maximizes real world return, eq. (7) is the RL algorithm, tested on the real world upon convergence-or whenever a threshold is reached-to collect the real world return that is then used to improve subsequent runs.

Given this, BayRn is carried on as a series of steps that repeat themselves preceded by an initialization phase that is required for the GP to work correctly, carried on using handpicked data or the previously mentioned UDR III-G for 5 to 10 steps; after that, the following cycle begins:

1) A RL algorithm in our case TRPO as it was the best performing- is employed to solve the lower level problem (7) by finding an optimal local policy for $\pi(\theta^*)$ for the current distribution used to generate ξ (the set of domain parameters);
2) The policy is evaluated on the real system for n_t rollouts providing an estimate of the return $\hat{J}^{real}(\theta^*)$ that is used to update the GP;
3) The upper level problem (6) is solved using BO, yielding a new distribution that is used to randomize the simulator.

The whole implementation was carried in Python using the *botorch* library for the BO part.

V. Experiments

As it was mentioned in the previous sections the primary purpose of our work was to train a policy in simulation such that it transfers to a different environment. The environment we used for this project was *Open Ai Hopper-V2*. The hopper is a one-legged robot model tasked with learning how to leap without falling while reaching the maximum horizontal speed achievable. 2 different environments were available: source and target. The source Hopper was generated by shifting the torso mass by 1kg with respect to the target domain, the only parameters we could then randomize in the simulation were the other weights of the hopper: thigh, leg and foot. Furthermore, we evaluated several approaches for this work to observe how different policies performed on this task. As mentioned in the methods section, we tested Reinforce, Actor-Critic, PPO and TRPO for on-policy approaches, and UDR and BayRn for domain randomization. Each policy is trained on the source domain and then evaluated on the source and target domains to compute the average return. Finally, each policy is trained directly on the target domain as to obtain an upper bound return.

A. On-Policy models training

The policies were trained in the custom source hopper environment with the hyperparameters seen in table I, the best performing was then tested on target. For REIFORCE and Actor Critic, the agent model is composed of 2 fully connected layers with 64 hidden layers. PPO and TRPO were trained using the *stablebaseline3* package. Due to the library requirements, the number of steps was used instead of the number of episodes. Checkpoints were saved every 1000 steps to pick the best performing policy.

TABLE I
HYPER PARAMETERS TESTED

model	hyper parameters
Reinforce	n-episodes:[10k,50k,100k], lr:[0.01,0.001,0.0001] gamma:[0.998,0.999]
Actor Critic	n-episodes:[10k,50k,100k], lr:[0.01,0.001,0.0001] gamma:[0.998,0.999]
PPO	n-steps:[100k,1M,10M], lr:[0.01,0.001,0.0001] gamma:[0.998,0.999]
TRPO	n-steps:[100k,1M,10M], lr:[0.01,0.001,0.0001] gamma:[0.998,0.999]

B. UDR

Uniform Domain randomization was carried by directly interacting with the environment builder *.py* script: the script was adapted to have a *udr* environment which, every time the environment is initialized, generates values for the thigh, leg and foot by sampling a uniform distribution in $[a, b]$. The size of the uniform distributions interval was handpicked starting from the original body part weight, but multiple configurations were tested to optimize the bound size:

TABLE II
TRPO+UDR BOUNDS TUNING

UDR Distributions	S-S return	S-T return
[3.4,4.5] [2.5,3.5] [4.5,5.5]	1436	1380
[3,5] [2,4] [4,6]	1421	1375
[1,7] [0.1,6] [2,8]	945	780

In table II we can see that going too far away from the original value (of 3.926, 2.714, 5.089) led to worse results overall, while if we stayed too close we wouldn't be exploiting the advantages of UDR; a slight deviation of ± 0.5 gave the best results.

C. BayRn

First of all, we tested different configurations for our BayRn implementation, changing how many steps to train for before testing on the target domain -but leaving the number of tests on the target domain unchanged-, the learning rate and the number of initialization steps; from the results -which were obtained by training a model for each configuration and then testing it over 100 iterations on the target domain- we can see in figure 2 that the hyperparameters did not influence much the return on the target environment, the number of initialization steps is not reported but it had the least influence.

Most of the best performing policies were trained with a higher learning rate and 10000 training steps between tests on the real world, so we decided to keep as parameters for further tests 5 initialization steps, 10000 steps -for 50 iterations for a total of 500000 steps- between tests and a learning rate of 0.01.

The last variable that was left to be tuned were the upper and lower bound for the values which we tried to optimize

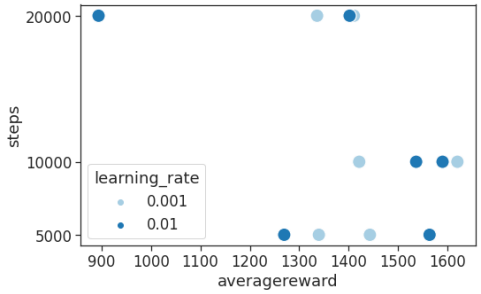

Fig. 2. Hyperparameter tuning for BayRn

using BayRn, that were the upper and lower bound of the uniform distributions used to generate the domain parameters. Given that we were looking for the range $[a, b]$ for each of the three domain parameters, b was fixed for each value in the lower bound while a for the upper bound, the remaining values were multiplied by a factor and then tested:

TABLE III
BAYRN BOUNDS TUNING

lower_bounds	[3.4,4.4,2.2,3.2,4.5,5.5]
upper_bounds	[3.9,4.4,2.7,3.2,5,5.5]

multiplier	average real world return	return variance
0.9/1.1	1041	28
0.7/1.3	1468	14
0.5/1.5	1053	35
0.2/1.8	1596	4

From the table III we can deduce that having a larger range of values from which to generate the domain distribution parameters lead to better performing policies, further trials not reported in the table highlighted the need to keep the bounds in range of the original value to a certain extent; having the lower bound b's and the upper bound a's fixed and close to the original value lead to better results consistently.

VI. RESULTS

TABLE IV
TEST RESULTS

model	Best hyper parameter	S-S	S-T	T-T
Reinforce	n-episodes: 50k, lr: 0.001, gamma: 0.998	55	48	11
Actor Critic	n-episodes: 100k, lr: 0.0001, gamma: 0.998	238	215	220
PPO	n-steps: 1M, lr: 0.001, gamma: 0.998	1552	1030	1420
TRPO	n-steps: 1M, lr: 0.001, gamma: 0.998	1629	1473	1722

Table IV represents the parameters among the ones we've tested in I which achieved the best results considering number of episodes and leaning rate.

For Reinforce, using more than 50k episodes or a smaller learning rate lead to worse results overall as it could not traverse well. On the other hand, for larger learning rates, it was exploring in a rapid way. For Actor Critic, going over 100k episodes or using learning rates above and below 0.001 proved to be worse overall.

The results of PPO and TRPO lead to better results overall when compared to off-policy, 1M steps and 10^{-3} proved to be the best parameters.

TABLE V
TRPO VS TRPO+UDR

Agent	S-S Return	S-T Return
TRPO	1338	1265
TRPO+UDR	1436	1380

In figure 3 we can see that REINFORCE does not manage to improve in performance over the episodes, meanwhile Actor Critic showed a slow improvement over episodes.

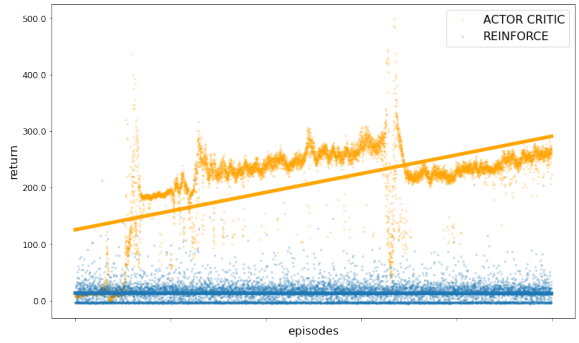

Fig. 3. Return over episodes for REINFORCE and Actor Critic

In table V are reported the results of our best performing TRPO and TRPO+UDR policies. As we can see, while training on the same environment we want to test on lead so better result to that specific environment, it is undeniable how adding Uniform Domain Randomization lead to a better performing policy on target, thus attesting to the efficacy of the method.

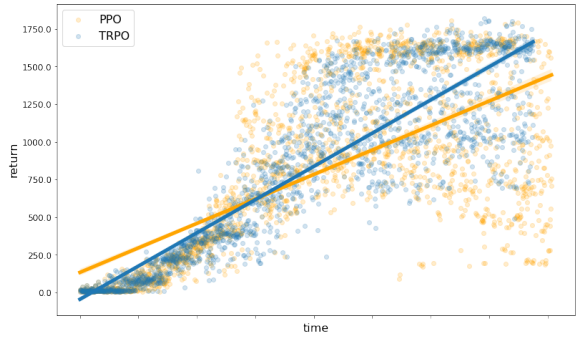

Fig. 4. Return over training time for PPO and TRPO

In figure 5 we can see a comparison between the reward we get using the different models we obtained, each of them

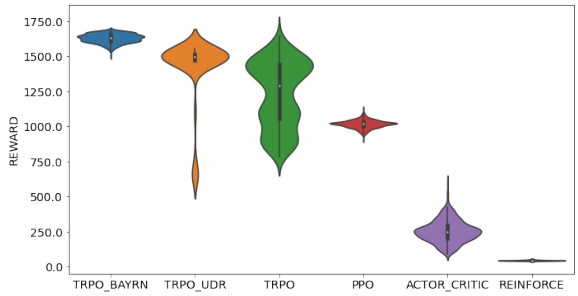

Fig. 5. Comparison of the reward on target between the trained models

was tested over 500 episodes on the target environment. As we can see, BayRn not only performs better than every other approach, but does so consistently; comparing BayRn with its direct competitor UDR, we can see that the theoretical best performance is similar, but the consistency in reaching said performance is what differentiates the two.

Speaking about UDR, we can see the effects of domain randomization by looking at the figure as well: when using UDR during training the resulting policy becomes more consistent overall despite the change in environment.

VII. Conclusions

In this paper we provided an overview of the reinforcement learning task by going through the progress that the task went through over the years, from on-policy methods such as REINFORCE up to complex domain randomization techniques such as BayRN. The results we observed while applying this technique show only one of the many advantages of using a domain randomization technique to bridge the reality gap, and give a hint of the future that lies ahead; as domain randomization becomes more and more advanced, more and more complex policies able to perform real world tasks will inevitably emerge, making precision tasks that were once believed to be only doable by humans something that a robot could automate.

References

[1] Richard S. Sutton & Andrew G. Barto: Reinforcement Learning: An introduction

[2] Wrobel, A. (1984). "On Markovian Decision Models with a Finite Skeleton". Mathematical Methods of Operations Research.

[3] Josh Tobin, Rachel Fong et al.: Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World

[4] N. Ruiz, S. Schulter, and M. Chandraker, "Learning to simulate," ArXiv e-prints, vol. 1810.02513, 2018

[5] B. Mehta, M. Diaz, F. Golemo, C. J. Pal, and L. Paull, "Active domain randomization," in CoRL, Osaka, Japan, October 30

[6] F. Ramos, R. Possas, and D. Fox, "Bayessim: Adaptive domain randomization via probabilistic inference for robotics simulators," in RSS, University of Freiburg, Freiburg im Breisgau, Germany, June 22-26, 2019.

[7] Fabio Muratore et al.: Data-efficient Domain Randomization with Bayesian Optimization

[8] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, Pieter Abbeel: Trust Region Policy Optimization

[9] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov: Proximal Policy Optimization Algorithms