Self-Paced Domain Randomization

Fabian Damken[*] Heiko Carrasco[*] Fabio Muratore Jan Peters

Abstract—**Deep Reinforcement Learning (DRL) has seen an uptick in publications due to its impressive performance in a variety of tasks. However, this comes at a cost since using a deep neural network policy requires a huge amount of data to learn from. Acquiring this data on a physical device is time and resource expensive. Thus, DRL often relies on simulations, since they provide vast amount of diverse training data faster than real time. A major problem in this research area is the *reality gap*, which describes the differences between the simulated and the real world, making the policy transfer from the virtual environment to a real robot brittle and difficult. In this paper we propose a novel application of curriculum learning to the area of domain randomization called Self-Paced Domain Randomization (SPDR), which puts the Reinforcement Learning (RL) policy "in the loop". By letting the policy influence the automatic generation of the curriculum of domain parameters based on its current performance we can show that this leads to performance increases and more stable policies when using DRL methods, both in the simulated environments and when applied to real-world platforms.**

Index Terms—**reinforcement learning, sim-to-real, deep learning**

I. Introduction

The usage of (physics) simulators in robot learning to train agents in a virtual environment has been proven quite beneficial, especially for DRL methods. They not only able to provide (almost) unlimited training data but can also easily be configured for different environments and tasks. Problems arise, however, when agents trained in such simulated environments are transferred to a real robotic platform. The so-called "reality gap", which denotes the subtle differences between the simulation and the real world, poses a well known challenge [1] for the trained agent to overcome. A number of strategies have been proposed in the recent years which attempt to address the reality gap in two main ways: Either by increasing the accuracy of the simulator (e.g. by incorporating real world data gathered from the robotic platform) [2] or by finding ways to increase robustness of the trained agent (for example by only providing noisy sensor data) [1]. While the former method has the downside of producing very specific policies, which do not generalize well (and might not even work on different instances of the same robotic platform), the latter method in the form of Domain Randomization (DR) has seen multiple successful applications in the past [1, 3, 4]. In DR certain domain parameters of the simulation such as for example the friction coefficients or the gravity constant are varied during training to encourage robustness. However, sampling the domain parameters using fixed training distributions, although showing promising results [5, 6, 7], might also be undesirable due to the increased learning complexity for the agent created by high-variance distributions. Dynamically updating the distribution could help with this but this leaves open the question on which basis the distribution is adapted during training.

One idea taken from supervised learning is the creation of a curriculum, which varies the examples presented to the model according to their estimated difficulty [8]. Applying this technique to the aforementioned problem of updating the domain parameter distribution allows for a gradually rising level of difficulty, which is bound by the agents performance. In this paper we present one possible attempt at implementing this solution based on the Self-Paced Reinforcement Learning framework proposed by Klink et al. [9]. We call this technique *Self-Paced Domain Randomization (SPDR)* and are able to show that this not only improves training time but also yields more stable agents which perform well on a wide range of domain parameters (not only the ones they were initially trained with). Compared to other distribution-varying domain randomization methods such as Uniform Domain Randomization (UDR) and Active Domain Randomization (ADR) we not only achieve higher performance in one-shot transfers to a real robotic platform but also are able to show that the trained agent is able to better generalize beyond the domain parameters used during training.

Contributions: We advance the state-of-the-art by introducing Self-Paced Domain Randomization (SPDR), a method to decrease the reality gap by iteratively adapting the domain randomization distributions according to the performance of the agent. Using curriculum learning approaches allows for an automated exploration of the search space, which is comparatively easy to implement yet still provides good results even when compared to other state-of-the-art algorithms such as ADR. Our method is independent from the used RL algorithm and can be used in a variety of tasks and environments. We validate our method by applying it both to real world and simulation tasks and compare it against other algorithms both in terms of overall performance and achieved robustness.

The remainder of the paper is structured as follows: First, we describe related work in addition to providing the theoretical background of our approach (Section II). We then continue with a description of SPDR in Section III before presenting our evaluation in Section IV. Finally, we conclude with a discussion our findings in Section V.

[*]Equal contribution.

II. Related Work

A. Curriculum Learning

Self-Paced Reinforcement Learning (SPRL) takes its basic principles from the observed human and animal learning behavior. By only gradually increasing the complexity of the shown samples, it can be shown that the learning performance and speed can be increased significantly [10]. This technique was formalized for supervised learning approaches by Bengio et al. under the term *curriculum learning* [8], in which the trained model is exposed to examples in an order defined by a curriculum. Ideally, this ordering is done by providing a training distribution from which examples can be sampled according to given configuration of weights. These weights are initially chosen to yield "easier" examples in the beginning and are updated according to the performance of the trained model.

The generation of such curricula has proven to be quite challenging, especially in DRL. Florensa et al. [11] propose "reversing the learning direction" by choosing the initial start states for the agent to be close to the goal state and gradually increase the distance between them based on the performance. Similarly, Riedmiller et al. [12] base their approach on the behavior of children during their playful exploration of the world by using basic rewards for small changes in the environment (such as the distance of certain movable objects between each other) in addition to higher but more complex task rewards. Both approaches, however, are based on heuristics and are not well understood, which inhibits further development.

Based on this observation, Klink et al. [13] proposed a new RL algorithm called Self-Paced Reinforcement Learning built on top of the approximate inference of the latent variable model [14]. In their approach, certain parameters of the task at hand, for example the initial position for a ball thrown to a ball-catching robot, are encoded inside the distribution of a context variable in a Contextual Markov Decision Processs (CMDPs) [15]. A curriculum in this context would than consist of a series of parameter changes interpolating between an initial distribution with a low task complexity and the target distribution, which usually has a higher task complexity. To automatically generate such a curriculum, the authors reinterpret the generation as an inference problem, which they solve by weighing a change in the current context distribution against the expected performance of the agent on this changed task environment. SPRL has been evaluated on a variety of tasks: Two variations of a point mass environment, where the agent needed to steer a point mass through a gate, which size and position were used as context parameters, and a ball-catching environment, where an agent controlled a Barrett WAM Arm to catch a thrown ball; here, the initial position of the ball was parameterized by the context distribution.

B. Other Domain Randomization Techniques

Another approach for the improvement of sim-to-real transfer is the Bayesian Domain Randomization (BayRn) algorithm proposed by Muratore et al. [4], which has similarities to SPDR. Both algorithms locally optimize the used policy on a distribution of domain parameters in a first step and then update said distribution using the performance of the locally trained policy. The crucial difference appears in the second step: while SPDR uses an estimation of the performance to gradually reduce the distance between the current and a target distribution, BayRn executes the policy on the real platform and uses the estimated return to learn a Gaussian Process (GP). The GP is then used to propose a new set of distribution parameters for the domain parameter, thus resulting in an Bayesian optimization problem. This process is repeated until a performance bound is reached, upon which the algorithm terminates. Since this approach uses real-world data from the target platform it provides more accurate results for the sim-to-real transfer compared to SPDR. However, this comes at the (reduced) cost of training on a real platform, therefore requiring either an easily resettable platform and task or constant supervision. On the other hand, SPDR does not require such things but performs significantly worse when applied to sim-to-real.

In this work we try to apply SPRL to increase the robustness of our trained policy against certain variations in the distribution of the (unknown) transition distribution. This task is known in the literature as Distributionally Robust Optimization (DRO) and is one way of introducing uncertainty and probabilities to optimization problems [16]. Although DRO is considered a sub-discipline of optimization research in general, there are some papers applying the theory to the specifics of Markov Decision Processs (MDPs) [17, 18, 19]. However, these approaches usually require special training methods or limit the possible set of policies; for example the proposed approach by Derman and Mannor does not support non-linear function approximations [19].

A survey of different methods in sim-to-real can be found in [20]. Their work differs from ours in the respect that it is a survey and existing approaches are presented but not evaluated.

III. Methods

A. Preliminaries

Consider a time-discrete dynamical system

$$\begin{aligned} \boldsymbol{s}_{t+1} &\sim \mathcal{P}_{\boldsymbol{\xi}}(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t), & \boldsymbol{s}_0 &\sim \mu_{\boldsymbol{\xi}}(\boldsymbol{s}_0) \\ \boldsymbol{a}_t &\sim \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t \mid \boldsymbol{s}_t), & \boldsymbol{\xi} &\sim \nu_{\boldsymbol{\phi}}(\boldsymbol{\xi}) \end{aligned}$$

with the continuous state $\boldsymbol{s}_t \in \mathcal{S} \subseteq \mathbb{R}^{n_s}$ and continuous action $\boldsymbol{a}_t \in \mathcal{A} \subseteq \mathbb{R}^{n_a}$ at time step t. The environment, also called domain, is instantiated through its parameters $\boldsymbol{\xi} \in \mathbb{R}^{n_\xi}$ (e.g., masses, friction coefficients, or time delays), which are assumed to be random variables distributed according to the probability distribution $\nu : \mathbb{R}^{n_\xi} \to \mathbb{R}_+$ parameterized by $\boldsymbol{\phi}$. These parameters determine the transition probability density function $\mathcal{P}_{\boldsymbol{\xi}} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}_+$ that describes the system's stochastic dynamics. The initial state $\boldsymbol{s}_0$ is drawn from the initial state distribution $\mu_{\boldsymbol{\xi}} : \mathcal{S} \to \mathbb{R}_+$. Together with the reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and the temporal discount factor $\gamma \in [0, 1)$, the system forms a MDP described by the $\mathcal{M}_{\boldsymbol{\xi}} = (\mathcal{S}, \mathcal{A}, \mathcal{P}_{\boldsymbol{\xi}}, \mu_{\boldsymbol{\xi}}, r, \gamma)$.

The goal of a Reinforcement Learning (RL) agent is to maximize the expected (discounted) return, a numeric scoring function which measures the policy's performance. The expected discounted return of a stochastic domain-independent policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t \mid \boldsymbol{s}_t)$, characterized by its parameters $\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^{n_\theta}$, is defined as

$$J(\boldsymbol{\theta}, \boldsymbol{\xi}, \boldsymbol{s}_0) = \mathbb{E}_{\boldsymbol{\tau} \sim p(\boldsymbol{\tau}|\boldsymbol{\xi})}\left[\sum_{t=0}^{T} \gamma^t r(\boldsymbol{s}_t, \boldsymbol{a}_t) \,\middle|\, \boldsymbol{\theta}, \boldsymbol{\xi}, \boldsymbol{s}_0 \right].$$

While learning from experience, the agent adapts its policy parameters. The resulting state-action-reward tuples are collected in trajectories, a.k.a. rollouts, $\boldsymbol{\tau} = \{\boldsymbol{s}_t, \boldsymbol{a}_t, r_t\}_{t=0}^{T}$, with $r_t = r(\boldsymbol{s}_t, \boldsymbol{a}_t)$. To keep the notation concise, we omit the dependency on $\boldsymbol{s}_0$. This parameterized version of the MDP can also be seen as a CMDP [15], where $\boldsymbol{\xi}$ is the context variable for the MDP.

B. Self-Paced Reinforcement Learning

To extend the formulation from Section III-A to curricula, we follow the notation by Klink et al. [13], except for using the domain parameter $\boldsymbol{\xi}$ and its corresponding distribution $\nu_{\boldsymbol{\phi}}$ as our context variable and its associated distribution. We augment the formulation of the expected reward from Section III-A to depend on the distribution $\nu_{\boldsymbol{\phi}}$ rather than on a single sampled parameter $\boldsymbol{\xi}$. This has no influence on our policy specification, however, it leads to a different expected return formulation

$$J[\boldsymbol{\theta}, \nu_{\boldsymbol{\phi}}] = \mathbb{E}_{\boldsymbol{\xi} \sim \nu_{\boldsymbol{\phi}}(\boldsymbol{\xi})}\big[J(\boldsymbol{\theta}, \boldsymbol{\xi})\big] \tag{1}$$

where the objective is a functional depending on the distribution $\nu_{\boldsymbol{\phi}}$. Compared to the previous formulation, a policy $\pi_{\boldsymbol{\theta}}$ which maximizes $J[\boldsymbol{\theta}, \nu_{\boldsymbol{\phi}}]$ can now be interpreted as being robust to perturbations of the domain parameters according to $\nu_{\boldsymbol{\phi}}$.

We can now introduce the concept of curricula in this setting by not optimizing (1) directly, but instead using an adaptive distribution $p_{\boldsymbol{\varphi}}(\boldsymbol{\xi})$ parameterized by $\boldsymbol{\varphi}$ during training. The distribution $p_{\boldsymbol{\varphi}}(\boldsymbol{\xi})$ is gradually changed over the course of training (e.g., after a fixed amount of steps) until it finally matches the desired target distribution $\nu_{\boldsymbol{\phi}}$. We call the tuple of all training distributions $p_{\boldsymbol{\varphi}_i}(\boldsymbol{\xi})$ used over the course of training the *curriculum*. As mentioned in Section II-A finding a good update rule and therefore a good curriculum for $p_{\boldsymbol{\varphi}}(\boldsymbol{\xi})$ is non-trivial.

To maximize (1), Klink et al. [9] propose a two-step method for finding $\boldsymbol{\theta}$ and $\boldsymbol{\varphi}$ separately:

1) employ any RL algorithm to optimize $J[\boldsymbol{\theta}, p_{\boldsymbol{\varphi}_i}]$ w.r.t. $\boldsymbol{\theta}$ to get $\boldsymbol{\theta}_{i+1}$ constituting the next policy π_{i+1}
2) update φ_{i+1} based on the performance of π_{i+1}

This is analogous to an Expectation-Maximization (EM) algorithm. While the first step is typically straightforward, the second step is more involved. Since a change in $\boldsymbol{\varphi}$ usually results in a performance drop of the current policy, it is necessary to strike a good balance between moving $\boldsymbol{\varphi}$ closer to the target distribution and not destroying all previously made training progress.

One proposal by Klink et al. [9] based on RL is allowing the agent itself to manipulate $p_{\boldsymbol{\varphi}}(\boldsymbol{\xi})$ by solving the optimization problem

$$\begin{aligned} \min_{\boldsymbol{\varphi}} \ & D_{\mathrm{KL}}\big(p_{\boldsymbol{\varphi}}(\boldsymbol{\xi}) \,\big\|\, \nu(\boldsymbol{\xi})\big) \\ \text{s.t.} \ & D_{\mathrm{KL}}\big(p_{\boldsymbol{\varphi}}(\boldsymbol{\xi}) \,\big\|\, p_{\boldsymbol{\varphi}_i}(\boldsymbol{\xi})\big) \leq \epsilon \\ & J[\boldsymbol{\theta}, p_{\boldsymbol{\varphi}}] \geq V_{\mathrm{LB}}, \end{aligned} \tag{2}$$

where V_{LB} is a lower bound on the performance and ϵ functions as a restriction for a proposed update by enforcing a maximum bound between the "old" distribution $p_{\boldsymbol{\varphi}_i}(\boldsymbol{\xi})$ and the "new" proposed distribution $p_{\boldsymbol{\varphi}}(\boldsymbol{\xi})$. Note that in this formulation we do not solve for the policy parameter $\boldsymbol{\theta}$, since it already incorporates the decoupled optimization of policy and training distribution as outlined in the two-step process before.

As the second constraint of (2) requires the expected performance given a new context distribution, a method for gauging this performance given a new $\boldsymbol{\varphi}$ is necessary.

If the upstream RL algorithm provides a value function $V^{\pi_{i+1}}(\boldsymbol{s}_0, \boldsymbol{\xi})$ or an estimated of the values (which is assumed from now on), estimating the new performance can be done in an importance sampling fashion. By sampling a set of K domain parameters $\boldsymbol{\xi}^{1:K} \sim p_{\boldsymbol{\varphi}_i}(\boldsymbol{\xi})$ and executing them, obtaining the trajectories $\boldsymbol{\tau}^{1:K}$, the respective values $V^{\pi_{i+1}}(\boldsymbol{s}_0, \boldsymbol{\xi}_k)$ can be approximated. Treating the distribution $p_{\boldsymbol{\varphi}_i}(\boldsymbol{\xi})$ as the proposal distribution, the expected return for a new domain parameter distribution $p_{\boldsymbol{\varphi}}(\boldsymbol{\xi})$ is

$$\hat{J}\big(\boldsymbol{\theta}, p_{\boldsymbol{\varphi}}(\boldsymbol{\xi})\big) = \frac{1}{K} \sum_{k=1}^{K} \frac{p_{\boldsymbol{\varphi}}(\boldsymbol{\xi}_k)}{p_{\boldsymbol{\varphi}_i}(\boldsymbol{\xi}_k)} V^{\pi_{i+1}}(\boldsymbol{s}_0, \boldsymbol{\xi}). \tag{3}$$

It should be noted that sampling is only done once after getting a new $\boldsymbol{\theta}$ and not in every iteration of the optimizer used for solving (2) (e.g., Sequential Quadratic Programming methods).

A few changes were necessary to adapt the technique behind SPRL [9] to the self-paced learning of domain parameters such as the friction or the motor constant. First, in the experiments done by Klink et al., often both mean and variance of the initial and target distributions vary. Although we tried this approach in the initial stages, it turned out to be more effective to use a fixed mean and start with a small initial variance and use a large target variance, such that the agent gradually learns to adapt to a wide range of possible domain parameter values. This can be interpreted as a maximum-entropy approach to increase the distributional robustness of the policy in regards to its transition matrix.

Second, Klink et al. used a different optimization objectives whenever the agent did not reach the performance lower bound in the last iteration. In our experiments we were able to show that this second objective, which tries to maximize

Figure 1: The Quanser Qube Platform

the performance of the agent by changing the distribution to be more accommodating to the agent, does not work in environments where the performance prediction using (3) is not very precise. Since this is specifically the case in our test environments with more than one domain parameter, we switched to a simpler approach, where the agent was trained until convergence was reached before the domain parameter distributions were updated.

IV. Experiments

We evaluated SPDR in sim-to-sim and sim-to-real settings. For sim-to-real we used a Furuta pendulum and a cart-pole swing up, simulated using a numerical solver as described in [4, app. A]. To also evaluate SPDR in sim-to-sim settings, we used again the platforms already used in sim-to-real and in addition several OpenAI Gym [21] environments. The environments we use are `HalfCheetah-v2`, and `Hopper-v3`. Both of these environments are powered by MuJoCo [22]. For implementing the method we used SimuRLacra [23], a framework for randomized physics simulations. For sim-to-real, we utilized the Quanser Qube (see Figure 1).

Algorithm 1: Self-Paced Domain Randomization

```
  Input: initial context distribution $p(\boldsymbol{\theta}; \boldsymbol{\varphi}_0)$, initial
         policy parameter $\boldsymbol{\theta}_0$, target context
         distribution $\nu(\boldsymbol{\xi})$, performance lower bound
         $V_{\mathrm{LB}}$, number of iterations $N$, rollouts per
         policy $K$
  Output: policy parameter $\boldsymbol{\theta}_N$
1 for $i \leftarrow 1$ to $N$ do
      ▷ Train the policy:
2     Sample $K$ domain parameters: $\boldsymbol{\xi}^{1:K} \sim p(\boldsymbol{\xi}; \boldsymbol{\varphi}_i)$
3     Generate trajectories for all; domain parameters:
        $\boldsymbol{\tau}^{1:K} \sim \mathrm{ROLLOUT}(\boldsymbol{\theta}, \boldsymbol{\xi}^{1:K})$
4     Train policy on $\boldsymbol{\tau}^{1:K}$ to get $\boldsymbol{\theta}_{i+1}$.
      ▷ Update the context distribution:
5     Obtain $p_{\boldsymbol{\varphi}_{i+1}}(\boldsymbol{\xi})$ by optimizing objective (2)
6 end
```

A. Baselines

To assess the quality of our approach, SPDR, we chose three different baselines: Uniform, Active, and Iterative Domain Randomization (UDR, IDR, and ADR). UDR applies domains randomization to all domain parameters available in the respective environment, e.g., the Furuta pendulum. The randomization is applied on the start of each rollout and the distributions over the parameters are never changed during training. IDR is similar to UDR but starts with a small variance on the domain parameters and iteratively increases it once a set performance bound is reached. This performance bound is different for each of the environments and coincides with the value of the return we associate with solving an environment. ADR [24] takes its core ideas from UDR but tries to maximize the "utility" of the chosen domain parameter distribution for the agent by training multiple Stein Variational Policy Gradient (SVPG) particles from which the domain parameters are subsequently sampled during training.

B. MuJoCo Gym Environments

We used different domain parameters for the environments: In the `HalfCheetah-v2` environment the total mass initially sampled from a distribution of $\mathcal{N}(m_c \,|\, 14, 0.1)$, where the variance was increased until a target distribution of $\mathcal{N}(m_c \,|\, 14, 20)$ was reached. For the `Hopper-v3` environment we used the friction coefficient of the foot as a parameter for SPDR and ADR.

For the policy we used a Multi-Layer Perceptron (MLP) with two hidden layers with 64 hidden units each and hyperbolic tangent activation functions. The subsequent training was performed with Proximal Policy Optimization (PPO) using Generalized Advantage Estimation (GAE) as the critic. Results can be found in Figure 2 and Figure 3.

Although SPDR performs better around the nominal value of the domain parameter in the `HalfCheeath-v2` case, it results in much more noisy returns compared to the other baselines. It stands to reason however, whether such divergences in the domain parameter actually occur in the real world and whether the agent should be robust against it (and possibly trading peak performance against overall robustness). For the `Hopper-v3` environment we observed no learning in the region around the nominal parameter for the friction coefficient for any of the tested baselines or our approach. We attribute this to the used policies hyperparameters and want to further investigate this phenomenon, which does not occur when trained with minimal or no domain randomization.

C. Furuta Pendulum (Sim-to-Sim and Sim-to-Real)

We modified the reward function of the Furuta environment to penalize derivations from the center position of the rotary arm more. To do this we modified the weights in the error term to be $\mathbf{Q} = \mathrm{diag}(1, 1, 2 \cdot 10^{-2}, 5 \cdot 10^{-3})$ and $\mathbf{R} = 4 \cdot 10^{-3}$. This prevents strong movements of the rotary arms and improves performance on the real environment by preventing the arms to crash into the workspace boundaries. The actions

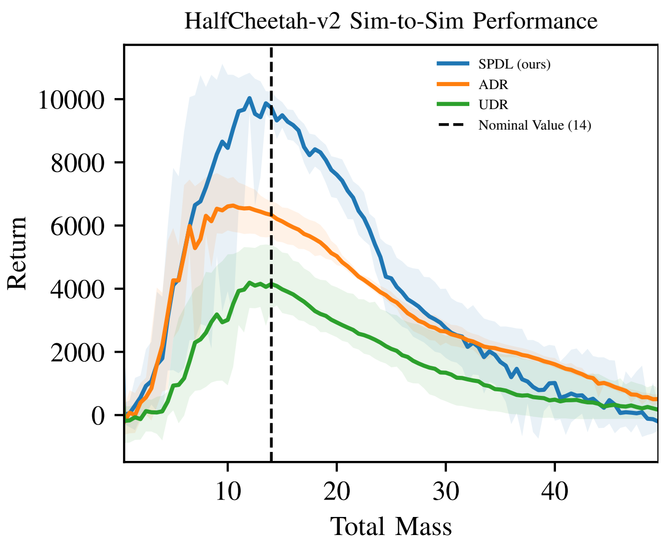

Figure 2: Comparison between UDR, ADR and SPDL on the `HalfCheetah-v2` gym environment. Shown are the average values with a shaded area depicting the 5% and 95% quantile.

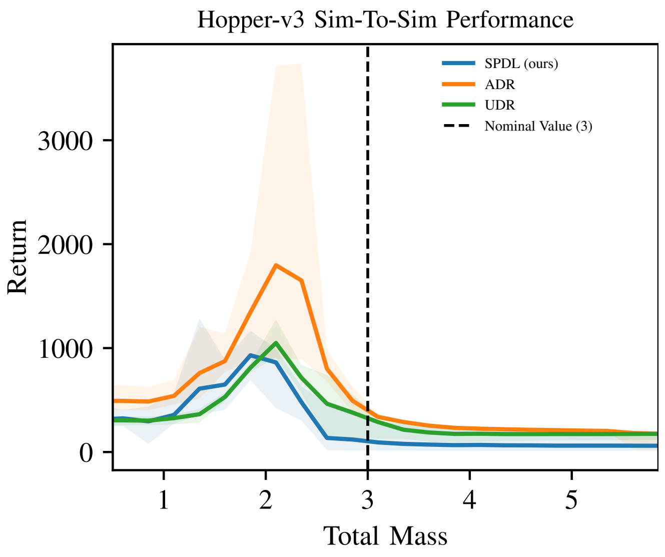

Figure 3: Comparison between UDR, ADR and SPDL on the `Hopper-v3` gym environment. Shown are the average values with a shaded area depicting the 5% and 95% quantile.

a are given by the voltage applied to the rotary arm motor which causes an acceleration in the horizontal pole.

During training, we optimize the distribution over the gravity g and the motor resistance R_m (in ADR, IDR, and SPDR). The initial parameter distributions are set to $\mathcal{N}(g \,|\, 9.81, 0.01)$ and $\mathcal{N}(R_m \,|\, 8.4, 0.01)$. For the target distribution, the variance is increased while the mean is kept identical as described in Section III-B: $\mathcal{N}(g \,|\, 9.81, 1)$ and $\mathcal{N}(R_m \,|\, 8.4, 1)$. It should be noted that, although the joint distribution of g and R_m is diagonal (and also isotropic), it is possible that the covariance matrix is non-diagonal during optimization. For the policy we used a MLP with two hidden layers with 64 hidden units each and hyperbolic tangent activation functions. The subsequent training was performed with PPO using GAE at a sample frequency of 500 Hz with a rollout length of 3000 steps (corresponding to 6 s of interaction time). We consider the problem to be solved for a return of over 1500, corresponding to a swing-up and perfect stabilization after 3 s. For training, we used 50 SPDR iterations with 150 PPO iterations. For SPDR a Kullback-Leibler (KL) constraint of $\epsilon = 10$ and a performance lower bound of $V_{\mathrm{LB}} = 1000$ was used, corresponding to a policy that be just about able to swing the pole up. Note that we used the same hyper-parameters for PPO for all methods.

1) Sim-to-Sim: Our hypothesis is that even though we solely randomized the gravity and motor resistance parameters, the resulting policies will be robust towards perturbations of other domain parameters, too. We test this on the damping coefficients D_r and D_p (for the rotary arm and pendulum, respectively) by varying the parameters on a grid $(D_r, D_p) \in [0.0, 0.0006] \times [0.0, 0.0001]$. The nominal values are $D_r = 5 \times 10^{-6}\,\mathrm{N\,m\,s\,rad^{-1}}$ and $D_p = 1 \times 10^{-6}\,\mathrm{N\,m\,s\,rad^{-1}}$. A comparison of the baseline and SPDR results is given in Figure 4. The heatmap shows the undiscounted cumulative return averaged over 20 policies trained with different seeds.

While our hypothesis is supported by the empirical results (the policy is robust even to perturbations in other parameters than the randomized ones), all methods are outperformed by UDR and exhibit similar performance. It should be noted, however, that the domain randomization in UDR is performed across all domain parameters, including the damping coefficients. An interesting result is that the boundary between a high and low reward is quite sharp. This can be explained by the policy needing to follow a fundamentally different strategy for tackling higher friction. One example for this is that the pole has to be swung back and forth a different number of times, making the difficulty of the problem non-continuous w.r.t. the domain parameters.

2) Sim-to-Real: We evaluated all (20) policies on the Quanser Qube 5 times each, corresponding to a total of 100 evaluations per algorithm. Again we set the maximum number of steps to 3000 and used a sampling frequency of 500 Hz. The results are shown in Figure 5. It is clear that ADR and UDR outperform out method, SPDR. Only IDR performs worse on the real platform that most policies trained with SPDR. It should be noted again that UDR was executed by sampling from all domain parameters while ADR, IDR, and SPDR only sampled on the gravity and motor resistance.

V. Discussion

For the MuJoCo environments we see that on the HalfCheetah-v2 environment, the performance of SPDR is significantly better than both ADR and UDR around the nominal value. As a downside, the return is much more noisy and reduces steeply for a larger total mass. On the Hopper-v3 environment we noticed some strange behavior: while ADR

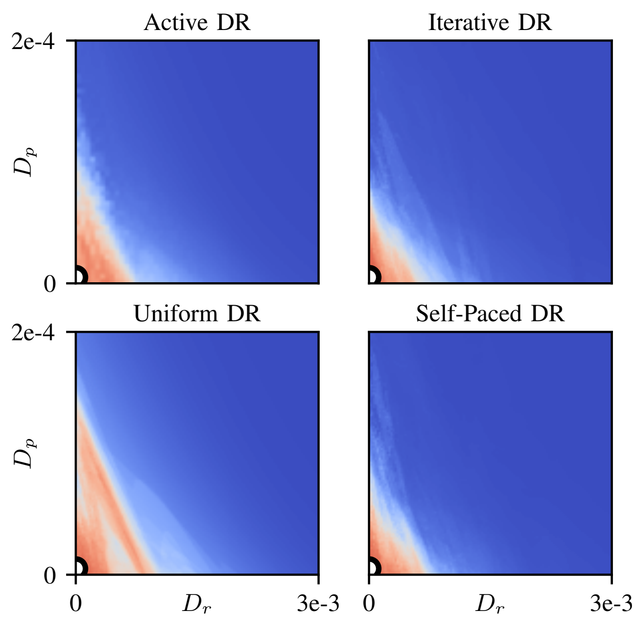

Figure 4: Sim-to-sim evaluation of the Furuta swing-up. The panels show, from the top left to the bottom right, the ADR, IDR, and UDR baselines along with results from SPDR (ours). Shown is the cumulative undiscounted return under perturbations of the damping coefficients D_r and D_p (on the x- and y-axis, respectively). The marker in the bottom left corners shows the nominal values. All baselines as well as SPDR have been evaluated on 20 or 20 or 20 different policies, training with different seeds. For all (meta-) algorithms, PPO was uses as the subroutine with equivalent hyper-parameters. The plots show the average total return over all policies generated by a single swing-up with the initial state $(0, 0, 0, 0)$ and no randomization. The dark red regions correspond to a total reward of 3000 whereas the dark blue regions correspond to a total reward of 0.

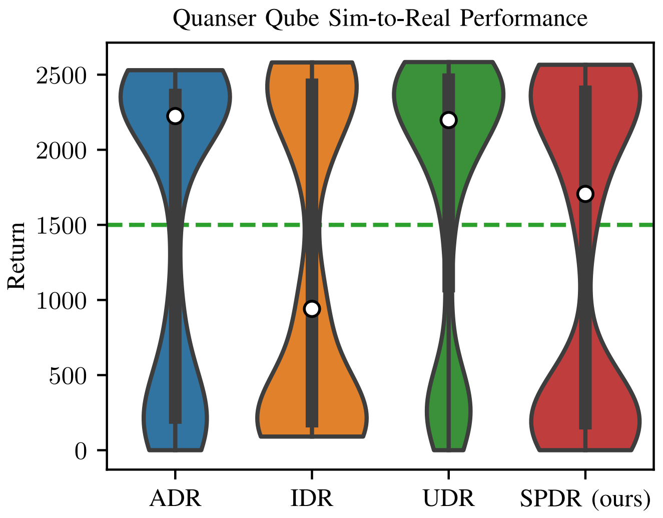

Figure 5: Sim-to-real evaluation of the Furuta swing-up on the Quanser Qube platform. Shown is a violin plot of 20 trained policies per method (the same policies used during sim-to-sim evaluation) with 5 evaluations on the real platform each. The upper and lower bounds of the black box in the center show the first and third quartile, the white dot shoes the second quartile (the median), and the needles show the minimum and maximum return. The colored area corresponds to a kernel density estimation of the return distribution for the respective method. Note that the widths of the distributions are scaled to match each other, so the areas will differ. The dashed green line shows the boundary after which we consider the problem to be solved.

outperforms SPDR and UDR and SPDR are roughly equal, all of them fail around the nominal mass.

From our sim-to-sim and sim-to-real experiment on the Furuta pendulum we conclude that while ADR and UDR ourperform our method, IDR performs significantly worse on the real system while having similar performance in simulation. This is a rather promising result as we used the same initial and target distribution for IDR, but the performance of SPDR is clearly superior. This indicates that the more complicated update rule yields more robust policies compared to the rather simple update done in IDR.

During the evaluation of SPDR we found that it has problems dealing with domain parameters with are small in magnitude. This could be caused by the variance being tiny in those cases, possibly leading to numerical instabilities. We evaluated using a log-transformation to increase the parameter size in magnitude. However, this leads to both problems with parameters close to one (for which the transformed value is, again, small in magnitude). Additionally the inverse transformation (i.e., an exponential function) squishes the parameters values in a small space around zero for largely negative values, which also causes problems with values falling in this range. Hence, tackling environments with domain parameters small in magnitude remains an open question.

We also found that the performance estimation needed in the first constraint of (2) and carried out by the importance sampling (3) has serious issues. By empirical measures it can be shown that the estimated performance is not close to the actual performance under the new domain parameter distribution. This has a high impact on the optimizer, which currently has to be compensated by adapting the upper bound ϵ on the KL divergence. For future work we suggest focusing on the performance estimation and to test different estimation methods such as computing a few rollouts for the new distribution to better estimate the performance. Another approach would be to model the dependency of the value on the domain parameters using a GP and to use the expectation of the marginal w.r.t. the domain parameters as the performance estimation. The joint distribution is comprised by the GP and the distribution over the domain parameters as the prior.

Acknowledgment

Calculations for this research were conducted on the Lichtenberg high performance cluster of the TU Darmstadt.

References

[1] Nick Jakobi, Phil Husbands, and Inman Harvey. "Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics." In: *Advances in Artificial Life*. Ed. by Federico Morán et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1995, pp. 704–720. ISBN: 978-3-540-49286-3. DOI: 10.1007/3-540-59496-5_337.

[2] Florian Golemo et al. "Sim-to-Real Transfer with Neural-Augmented Robot Simulation." In: *Conference on Robot Learning*. Conference on Robot Learning. PMLR, Oct. 23, 2018, pp. 817–828. URL: https://proceedings.mlr.press/v87/golemo18a.html (visited on 09/03/2021).

[3] Yevgen Chebotar et al. "Closing the Sim-to-Real Loop: Adapting Simulation Randomization with Real World Experience." In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019 International Conference on Robotics and Automation (ICRA). May 2019, pp. 8973–8979. DOI: 10.1109/ICRA.2019.8793789.

[4] Fabio Muratore et al. "Data-Efficient Domain Randomization with Bayesian Optimization." In: *IEEE Robotics and Automation Letters* 6.2 (Apr. 2021), pp. 911–918. ISSN: 2377-3766, 2377-3774. DOI: 10.1109/LRA.2021.3052391. arXiv: 2003.02471.

[5] Xue Bin Peng et al. "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization." In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018 IEEE International Conference on Robotics and Automation (ICRA). May 2018, pp. 3803–3810. DOI: 10.1109/ICRA.2018.8460528.

[6] Josh Tobin et al. "Domain randomization for transferring deep neural networks from simulation to the real world." In: *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2017, pp. 23–30.

[7] Xue Bin Peng et al. "Sim-to-real transfer of robotic control with dynamics randomization." In: *2018 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2018, pp. 3803–3810.

[8] Yoshua Bengio et al. "Curriculum Learning." In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. New York, NY, USA: Association for Computing Machinery, June 2009, pp. 41–48. ISBN: 978-1-60558-516-1. DOI: 10.1145/1553374.1553380.

[9] Pascal Klink et al. "A Probabilistic Interpretation of Self-Paced Learning with Applications to Reinforcement Learning." In: *arXiv:2102.13176 [cs]* (Feb. 2021). arXiv: 2102.13176 [cs].

[10] B. F. Skinner. "Reinforcement Today." In: *American Psychologist* 13.3 (1958), pp. 94–99. ISSN: 1935-990X(Electronic),0003-066X(Print). DOI: 10.1037/h0049039.

[11] Carlos Florensa et al. "Reverse Curriculum Generation for Reinforcement Learning." In: *arXiv:1707.05300 [cs]* (July 2018). arXiv: 1707.05300 [cs].

[12] Martin Riedmiller et al. "Learning by Playing - Solving Sparse Reward Tasks from Scratch." In: *arXiv:1802.10567 [cs, stat]* (Feb. 2018). arXiv: 1802.10567 [cs, stat].

[13] Pascal Klink et al. "Self-Paced Contextual Reinforcement Learning." en. In: *Conference on Robot Learning*. PMLR, May 2020, pp. 513–529.

[14] Marc Toussaint and Amos Storkey. "Probabilistic Inference for Solving Discrete and Continuous State Markov Decision Processes." In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML '06. New York, NY, USA: Association for Computing Machinery, June 2006, pp. 945–952. ISBN: 978-1-59593-383-6. DOI: 10.1145/1143844.1143963.

[15] Assaf Hallak, Dotan Di Castro, and Shie Mannor. "Contextual Markov Decision Processes." In: *arXiv:1502.02259 [cs, stat]* (Feb. 2015). arXiv: 1502.02259 [cs, stat].

[16] Hamed Rahimian and Sanjay Mehrotra. *Distributionally Robust Optimization: A Review*. Aug. 12, 2019. arXiv: 1908.05659 [cs, math, stat]. URL: http://arxiv.org/abs/1908.05659 (visited on 09/03/2021).

[17] Arnab Nilim and Laurent El Ghaoui. "Robust Control of Markov Decision Processes with Uncertain Transition Matrices." In: *Operations Research* 53.5 (Oct. 1, 2005), pp. 780–798. ISSN: 0030-364X. DOI: 10.1287/opre.1050.0216. URL: https://pubsonline.informs.org/doi/abs/10.1287/opre.1050.0216 (visited on 09/03/2021).

[18] Wolfram Wiesemann, Daniel Kuhn, and Berç Rustem. "Robust Markov Decision Processes." In: *Mathematics of Operations Research* 38.1 (Feb. 1, 2013), pp. 153–183. ISSN: 0364-765X. DOI: 10.1287/moor.1120.0566. URL: https://pubsonline.informs.org/doi/abs/10.1287/moor.1120.0566 (visited on 09/03/2021).

[19] Esther Derman and Shie Mannor. *Distributional Robustness and Regularization in Reinforcement Learning*. July 14, 2020. arXiv: 2003.02894 [cs, math, stat]. URL: http://arxiv.org/abs/2003.02894 (visited on 09/03/2021).

[20] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. *Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: A Survey*. Sept. 24, 2020. arXiv: 2009.13303 [cs]. URL: http://arxiv.org/abs/2009.13303 (visited on 11/25/2020).

[21] Greg Brockman et al. "OpenAI Gym." In: *arXiv:1606.01540 [cs]* (June 2016). arXiv: 1606.01540 [cs].

[22] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A Physics Engine for Model-Based Control." In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[23] Fabio Muratore. *SimuRLacra – A Framework for Reinforcement Learning from Randomized Simulations*. Honda Research Institute. 2021.

[24] Bhairav Mehta et al. "Active Domain Randomization." en. In: *Conference on Robot Learning*. PMLR, May 2020, pp. 1162–1176.