공학석사학위논문

도메인 랜덤화 기법을 이용한 던지기 행동 전이

Zero-Shot Learning for Transfer of a Throwing Task via Domain Randomization

2021 년 2 월

서울대학교 대학원

기계항공공학부

박 성 용

도메인 랜덤화 기법을 이용한 던지기 행동 전이

Zero-Shot Learning for Transfer of a Throwing Task via Domain Randomization

지도교수 김 현 진

이 논문을 공학석사 학위논문으로 제출함

2021 년 1 월

서울대학교 대학원

기계항공공학부

박 성 용

박성용의 공학석사 학위논문을 인준함

2020 년 12 월

위 원 장 김 유 단

부위원장 김 현 진

위 원 백 찬 국

Zero-Shot Learning for Transfer of a Throwing Task via Domain Randomization

A Thesis

by

Sungyong Park

Presented to the Faculty of the Graduate School of
Seoul National University
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Mechanical & Aerospace Engineering

Seoul National University

Supervisor : Professor H. Jin Kim

February 2021

to my

FAMILY

with love

Abstract

Zero-Shot Learning for Transfer of a Throwing Task via Domain Randomization

Sungyong Park
Department of Mechanical & Aerospace Engineering
The Graduate School
Seoul National University

Deep reinforcement learning (DRL) on continuous robot control has received a wide range of interests over the last decade. Collecting data directly from real robots results in high sample complexities and can cause safety accidents, so simulators are widely used as efficient alternatives for real robots. Unfortunately, policies trained in the simulation cannot be directly transferred to real-world robots due to a mismatch between the simulation and the reality, which is referred to as 'reality gap'. To close this gap, domain randomization (DR) is commonly used. DR guarantees better transferability in the zero-shot setting, i.e. training agents in the source domain and testing them on the previously unseen target domain without fine-tuning. In this work, the positive influence of DR on zero-shot transfer in Sim2Sim setting with an object throwing task is presented.

Keyword : Robot manipulation, Reinforcement learning, Zero-shot learning, Domain randomization

Student Number : 2019-21027

Table of Contents

List of Figures

List of Tables

1 Introduction

With advances in deep learning, "deep reinforcement learning" (DRL) has become applicable to high-dimensional problems. DRL has recently yielded remarkable results such as achieving superhuman performances in games [1, 2] and training cooperative multiple agents [3]. Especially, DRL has succeeded in learning complex robotic control tasks which were previously regarded as unsolvable due to continuous state/action spaces [4, 5]. Over the last decade, a number of DRL algorithms have been suggested to learn continuous control policies. However, deploying DRL methods directly in the real-word robot is not appropriate; their inherent high sample complexity slows the entire training time, and exploratory actions of the early learning phase may cause safety accidents.

Training robot control tasks in the simulation is one of the promising approaches that avoid some problems in real-robot learning, and a lot of simulators are widely used as efficient alternatives for real robots. With physically well-modeled simulators, agents can acquire diverse and vast training data at a low cost. Additionally, since simulators can run multiple robot agents simultaneously, they can learn control policies drastically faster than real-world robots.

Unfortunately, policies trained in the simulation cannot be directly transferred to real-world robots due to unavoidable limitations of the simulator's precision; model discrepancies between the simulation and real-world dynamics always exist. This model mismatch is referred to as the

"reality gap".

To tackle this problem, a popular method is to diversify simulation environments in which an agent is trained. By choosing a different value of simulation parameters (e.g. mass, friction) randomly at each episode, an agent can be exposed to various learning environments, which is referred to as domain randomization (DR). With DR, an agent can be induced to learn a general policy that works in a wide range of environments, including real-world robots. Generally, DR is regarded as one of the reliable methods for the zero-shot domain transfer, where a policy optimized in the source domain (e.g. simulation) is tested on the target domain (e.g. real world) *without any fine-tuning* [6, 7].

If the source domain is the simulation and the target domain is the real-world robot, it is referred to as the sim-to-real (Sim2Real) transfer [8]. We can also set the sim-to-sim (Sim2Sim) transfer setting, both the source and target domains are the simulation but with different values of domain parameters. In this work, using the RL algorithm DDPG [9], the positive influence of DR on zero-shot transfer in Sim2Sim setting with an object throwing task is presented. The throwing task is rarely selected for evaluating DR because it includes prehensile manipulation involved with contact dynamics which hinder learning.

Further, I investigate the robustness of policies learned with DR against 'unmodeled' effects, i.e. physical attributes that are not modeled in the simulator. Inevitably, due to the inherent imprecision of the simulator, there always exist unmodeled effects and they decrease transfer performances to the real world. In Sim2Sim setting, unmodeled real-world effects can be considered as domain parameters that are not randomized in the source simulation during the training phase and vary only in the target simulation [10].

1.1 Literature review

In [6, 8], they use the uniform sampling to choose some parameter values to instantiate the source domain, such as the textures of objects or the height of the table. This DR method is called 'uniform domain randomization' (UDR). It is a simple idea and easy to implement, but it shows reasonable performances in real-world rollouts when compared to simulation results.

In EPOpt [10], they apply the adversarial training idea to DR and change the way training

data are selected. When the agent is trained, the agent interacts with environments instantiated from domain parameters sampled from the Gaussian distribution and outputs trajectories. At every iteration, EPOpt determines which trajectories are used to train the agent and selectively chooses trajectories that present low returns from total trajectory data. This technique achieves more generalized policy in simulation target domains and high zero-shot performances. UDR and EPOpt are similar in that they don't update their distributions over domain parameters; UDR maintains the uniform distribution in the whole training phase and EPOpt the Gaussian distribution. However, EPOpt introduces its own extra trajectory-sampling strategy and it leads to a more robust policy. [10] also investigates the influence of EPOpt on 'unmodeled' effects. Applied to Sim2Sim transfer setting, EPOpt also shows better zero-shot performances to unmodeled effect.

In ADR [5], they introduce a set of neural networks referred to as 'SVPG particles' [11, 12]. They output domain parameter values, in other words, they output randomized environments where agents are trained. SVPG particles are updated to output more difficult environments which make agents hard to achieve high performance. The difficulty is measured by introducing the reference environment and the discriminator neural network. The discriminator is updated to discriminates between trajectories from randomized environments and trajectories from reference environments. SVPG particles are more rewarded when they output the environment which causes the wrong prediction from the discriminator. This mechanism makes the RL agent be exposed to more difficult environments and induces it to learn a more robust policy. Similar to EPOpt, ADR also uses the adversarial training method. However, the different point from EPOpt is that ADR updates the way domain parameters are sampled; ADR selects environments where trajectory data are sampled, not trajectories directly.

In this paper, I use UDR to train robust policies and to compare them to policies trained without any DR methods.

1.2 Thesis contribution

In this paper, I present several results as follows:

1. The result shows that policies trained with DR show higher zero-shot performances (i.e. more generalized performances) over target domains.

2. I succeed in learning an object throwing task, which is rarely selected for evaluating DR because of its prehensile manipulation involved with contact dynamics which hinder learning.

3. Referring to [10], I build the experiment setting to realize 'unmodeled' effects in the simulation and the result shows that DR also outputs policies robust to unmodeled effects.

1.3 Thesis outline

The remainder of this paper is written as follows. In Chapter 2, I introduce the background knowledge related to RL in order to help readers to understand the rest chapters, including the explanation about notations. In Chapter 3, I elaborate the RL algorithm DDPG which I use to train the policy in this paper. In Chapter 4, I explain the concept of DR method and how it works. The entire setup for the robot simulation experiment is concretely described in Chapter 5 and simulation results are in Chapter 6. Chapter 7 contains the conclusion.

2

Background

In this chapter, I describe notations used in the remainder of this paper and introduce background concepts related to RL underlying through this paper.

2.1 Reinforcement learning (RL)

Reinforcement learning (RL) is one of the promising machine learning methods, training the agent to maximize total rewards which it obtains from the outside environemnt. RL assumes the environment to be Markov Decision Process (MDP) with the tuple $\mathcal{M} = (S, A, P, R, p_0, \gamma, T)$. S and A denote the continuous state space and the continuous action space, respectively. The dynamics of the environment is represented by the state-transition probability $P : S \times A \times S \to \mathbb{R}_+$. For every timestep, an agent gets a reward from the predefined reward function of the environment $R : S \times A \to \mathbb{R}$. A distribution of initial state is denoted as $p_0 : S \to \mathbb{R}_+$ and γ denotes the discount factor. I set every episode to terminate at a fixed horizon T. An action $a_t \in A$ is drawn from a policy $\pi(a_t|s_t)$ given a state $s_t \in S$, denoted as $a_t \sim \pi(a_t|s_t)$, and a state $s_{t+1} \in S$ follows the state-transition probability, denoted as $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$. The goal of an RL agent is to find the optimal policy parameterized by θ that maximizes the expected discounted sum of rewards

over trajectories $\tau = (s_0, a_0, ..., s_T, a_T)$, i.e. the policy maximizes the objective as follows:

$$\mathbb{E}_{\pi_\theta}[R(\tau)] = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t)\right] \tag{2.1}$$

where $R(\tau)$ is the entire trajectory reward and $R(s_t, a_t)$ the single-step reward.

2.2 Actor-critic structure

The actor-critic structure is the basic idea of a large number of RL algorithms. The actor neural network suggests the way the RL agent acts; it refers to the policy $\pi(a_t|s_t)$. The critic neural network evaluates the action the agent takes; in most cases, it refers to the action-value function $Q(s_t, a_t)$ which predicts the expected return from the visited state and the taken action at a certain timestep.

2.3 Deterministic policy

The deterministic policy, different from the stochastic policy which represents the probability distribution over actions given a state, represents the function that outputs a single action value from the input state. From the point of view of RL, the deterministic policy has several strengths compared to the stochastic policy. (i) When the gradient of policy weights is calculated (i.e. policy gradient), the stochastic policy requires the expectation calculation over the state space and the action space, but the deterministic policy only deals with the state space to calculate the expectation [13]. (ii) The deterministic policy can convert the on-policy algorithm into the off-policy algorithm. When the critic is trained, the TD(Temporal Difference) target is used to form the critic loss, and this TD target is induced by the Bellman equation (Eq. (2.2)).

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1}}\left[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1}}[Q^\pi(s_{t+1}, a_{t+1})]\right] \tag{2.2}$$

If the policy is deterministic, we can remove inner expectation over the action space as Eq. (2.3) and it indicates that we can update the policy with trajectories obtained from different policies [9].

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1}}\left[r(s_t, a_t) + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1}))\right] \tag{2.3}$$

3

Deep Deterministic Policy Gradient (DDPG)

In this work, I use an RL algorithm Deep Deterministic Policy Gradient (DDPG) to train policies. DDPG is a model-free, off-policy algorithm with actor and critic approximated by neural networks. DDPG can scale to high-dimensional, continuous control problems and it has been one of the widely popular value-based DRL algorithms.

DDPG adopts two major techniques from Deep Q Network (DQN) [14]. Similar to DQN, DDPG (i) updates target networks toward original networks slowly, i.e. "soft" target updates and (ii) uses a 'replay buffer' to train agents with temporally uncorrelated state transition samples. This leads to stable updates of actor and critic without divergence, and DDPG shows good performances on complex continuous action tasks.

In this chapter, I elaborate on how DDPG works and which techniques are used. Much of this chapter refers to [9, 13] a lot, including notations.

3.1 Policy update

DDPG adopts the policy update method from the DPG algorithm [13]. When dealing with continuous actions, calculating the optimized action value that makes the action-value function maximum at each timestep is computationally expensive. Instead, DPG updates the policy according to the gradient of Q as follows:

$$\theta^{\pi} \leftarrow \theta^{\pi} + \alpha \mathbb{E}_s \left[\nabla_{\theta^{\pi}} Q(s, \pi(s; \theta^{\pi}); \theta^{Q}) \right] \tag{3.1}$$

where α is the learning rate. With the chain rule, we can divide the gradient of Q into two parts as follows:

$$\theta^{\pi} \leftarrow \theta^{\pi} + \alpha \mathbb{E}_s \left[\nabla_{\theta^{\pi}} \pi(s; \theta^{\pi}) \nabla_a Q(s, a; \theta^{Q})|_{a=\pi(s;\theta^{\pi})} \right] \tag{3.2}$$

3.2 Training techniques

In this section, I elaborate on training techniques DDPG uses to achieve stable convergence and efficient learning.

3.2.1 Target networks

Similar to DQN, DDPG creates target networks for the actor and the critic to calculate the TD target y_i of the critic loss as follows:

$$L_{critic} = \mathbb{E}_{s_i, a_i, r_i, s_{i+1}} \left[(y_i - Q(s_i, a_i; \theta^{Q}))^2 \right] \tag{3.3}$$

where $y_i = r_i + \gamma Q'(s_{i+1}, \pi'(s_{i+1}; \theta^{\pi'}); \theta^{Q'})$. θ^{π} and θ^{Q} represent weights of the actor and the critic. The apostrophe on superscript letters represents the target network.

At the beginning of the training, target networks are copied from their own original actor and critic as initialization. At every timestep, target networks are updated as follows:

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau \theta^{Q} + (1 - \tau) \theta^{Q'} \\ \theta^{\pi'} &\leftarrow \tau \theta^{\pi} + (1 - \tau) \theta^{\pi'} \end{aligned} \tag{3.4}$$

where $\tau \ll 1$. This 'soft' target update changes target networks slowly and eventually slows down the learning process, but it is regarded as the inevitable trade-off in order to increase the stability of optimization.

3.2.2 Replay buffer

Since DDPG uses the deterministic policy, it can train agents off-policy. As mentioned in Section 2.3, the off-policy algorithm can use transition data generated from past policies to train the policy that the agent currently follows.

The repository where the past transitions are stored for later updates refers to the 'replay buffer'. At every timestep, the transition (s_i, a_i, r_i, s_{i+1}) is stored to the replay buffer and the minibatch of transitions is sampled to update networks. If the replay buffer is full, the oldest data are removed from the buffer. In short, DDPG employs the 'experience replay' method by using the replay buffer as in DQN.

Most optimization algorithms are based on 'independently and identically distributed' (i.i.d) samples; training data have to be sampled independently on other samples and from the unchanging distribution. However, neural networks used in RL are barely exposed to those i.i.d-sampled data since transitions in trajectories are sequential (i.e. temporally correlated). The experience replay method can relieve this problem by randomly choosing transitions from random timesteps as training data.

3.2.3 OU noise

Since DDPG uses the deterministic policy, the policy completely loses the exploration mechanism. To deal with this problem, authors in DDPG introduce the Ornstein-Uhlenbeck noise (i.e. OU noise) to the deterministic policy. At every timestep, OU noise is sampled and added to the deterministic policy. OU noise is well known for its exploration ability when it is applied to the environment with inertia.

4

Domain Randomization (DR)

In this chapter, I elaborate on the concept of DR with RL notations and the formula of RL. The objective of DR is to obtain policies robust to environments where policies have rarely or even never experienced but are supposed to be applied. If we refer to the environment where the RL agent is trained as 'the source domain' and where the trained RL agent is evaluated 'the target domain', the objective of DR can be rewritten: to obtain polices which are trained only in source domains, but show high performance over target domains without any additional training on target domains (or even any extra fine-tuning). We can instantiate various environments by changing the values of 'domain parameters'. If we apply trained policies to other domains, we refer to this as 'transfer'. Since DR methods transfer the policy to the target domain without any target domain training data, we refer to this as 'zero-shot domain transfer' [5].

4.1 Objective function

If we regard domain parameters ξ (e.g. mass, friction coefficient) as random variables, the environment is instantiated by randomized parameters and thus transition dynamics change; transition probability is additionally conditioned by ξ, which can be written in $s_{t+1} \sim P(s_{t+1}|s_t, a_t, \xi)$. ξ is sampled from the predefined distribution p parameterized by ϕ, which can be written as $\xi \sim p_\phi(\xi)$. Thus, the final objective of reinforcement learning with DR is to find the policy parameter θ that maximizes the expected discounted sum of rewards under $p_\phi(\xi)$:

$$\max_\theta \mathbb{E}_\xi[\mathbb{E}_{\pi_\theta}[R(\tau)]] \tag{4.1}$$

Eq. (4.1) is composed of two expectation parentheses. Inner parentheses represent the expected return over policies and this formula with the outermost maximization is exactly same with the objective function of RL (Eq. (2.1)). However, outer expectation wrapping the original RL objective changes the goal of the agent; the agent is updated to achieve high performance on overall 'source' domains (the agent can access only to the source domain when the agent is in the training phase). The policy trained with this modified objective is robust to the variation of the target domain and shows better zero-shot performances.

4.2 Uniform Domain Randomization (UDR)

For simplicity, I assume the parameter distribution p as the uniform distribution with predefined ranges for each parameter. The policy optimization algorithm is DDPG. The whole process of UDR is shown in Algorithm 1 as follows:

Algorithm 1: UDR

Initialize: policy π_θ, replay buffer R

```
for i ← 1 to N_iteration do
    ξ ∼uniform(ξ_low, ξ_high)
    build an environment E from ξ
    for t ← 1 to N_timesteps do
        generate the transition (s_t, a_t, r_t, s_{t+1}) ∼ E with π_θ
        R ← R ∪ (s_t, a_t, r_t, s_{t+1})
        sample the transition minibatch T from R
        θ ← PolicyOptimization(θ, T)
    end
end
```

5

Experimental setup

In this chapter, I describe the entire robot simulation experiment setup in detail. As mentioned earlier, I set the sim-to-sim(Sim2Sim) transfer setting, both the source domain and the target domain are the simulation but with different values of the domain parameter.

5.1 Robot and task setup

I use MuJoCo physics engine [15] to construct a simulator for the 6-DOF manipulator UR3 and make an object throwing task environment (Fig. 5.1). A simulation timestep is 0.002 s. I set the object as a cube with a side length of 3 cm and a container box with a dimension of 20 cm $\times$ 20 cm $\times$ 15 cm to which the cube is thrown. The position of the container box is fixed as 1.1 m in the x-coordinate. Initial joint angles for 6 joints of UR3 are [0, -45, -90, -180, -90, 90] degrees by the order of joint 1 $\sim$ joint 6.

For simplicity, I just operate joints 2, 3, 4 of UR3 and fix the other joints same as initial conditions. Rotation of joints 2, 3, 4 is limited within a range of [-90, 0] degrees, [-90, 90] degrees, [-270, -45] degrees, respectively. Additional saturation is applied when each joint torque exceeds its motor's input.

Each training episode starts with grippers gripping the object and grippers are set to release after 0.5 s.

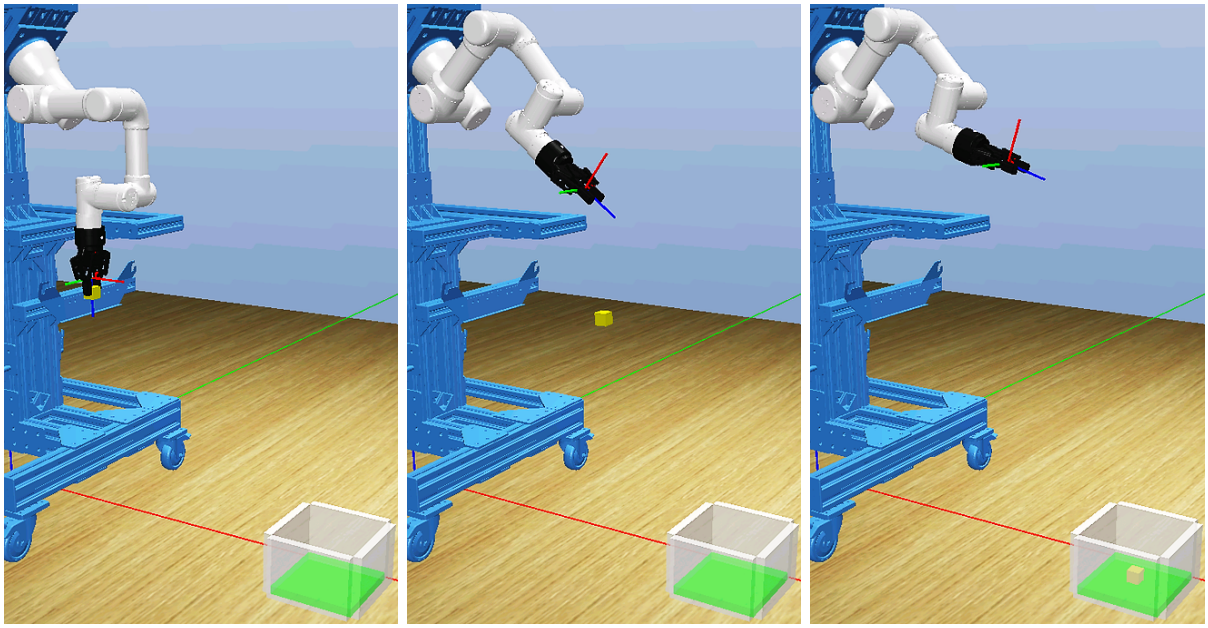
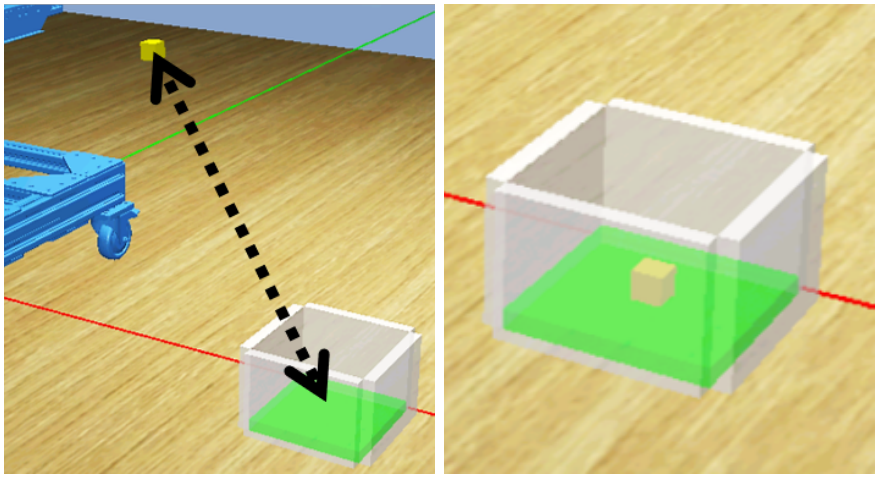


Figure 5.1: MuJoCo simulator of UR3 for an object throwing task.

5.2 State and action

Raw observation dimension is 45 comprised of the joint angles, the joint velocities, and the position and velocity of the object. Observation is converted into 7-dimensional state. The state is comprised of observation elements corresponding to joints 2, 3, 4 and the gripper translation to check if it has released the object.

Action is comprised of desired joint velocities of 3 dimension. It is converted into torque by the internal PID controller. Grippers are opened at a predefined time so the input torques to grippers are not included in action space.



(a) (b)

Figure 5.2: Reward shaping : (a) Agents get negative rewards when the cube is too far from the container box (b) Agents get positive rewards when the cube touches the floor of the container box.

5.3 Reward function shaping

I construct the reward function for the throwing task as follows:

$$r_t = -C_1 r_{dist} + C_2 r_{touch} \tag{5.1}$$

In Eq. (5.1), r_t represents single-step reward at timestep t, $r_t = R(s_t, a_t)$, and it consists of two sub-rewards (Fig. 5.2). C_1 and C_2 are the positive hyperparameters to be shaped. r_{dist} represents the L2 distance between the cube and the center of the container floor. This sub-reward penalizes farther distance. r_{touch} represents the sensor reading which outputs 1 for every timestep when the cube touches the container floor. This sub-reward encourages the cube to arrive in the container.

r_{touch} is a bonus reward that is necessary because, without this sub-reward, the agent can get lower rewards even though the cube gets in the container. For example, a cube thrown directly before the outer wall of the container in a short time can get higher total rewards than a cube flying in an arc and falling into the container with a longer time because the latter was penalized by r_{dist} for its overall trajectories.

5.4 Domain parameters

I choose three simulation parameters to randomize: the damping coefficient of the gripper joints, the damping coefficient of the object and the density of the object.

Even though UR3 grippers have just 2 joint motors to operate, the total grippers are comprised of several sub-elements and there are 8 other joints so the number of the total joints is 10. They have the same damping coefficient value.

In the simulator, an object is regarded as a 6-DOF free joint and it also has parameters of joint: friction loss, damping coefficient, stiffness. I choose the damping coefficient to randomize due to its dominance against the others; the damping coefficient has a greater effect on the trajectory of the object than the others.

The parameters above are randomized in the source domain. I also choose the density of the object as the domain parameter which varies only in the target domain to realize unmodeled real-world effects. Since I fixed the object shape as a cube with the same side length, the same density also means the same mass.

5.5 Policy training scheme

I trained two kinds of policies: the baseline policy ("Baseline") and the policy with the uniform domain randomization ("UDR"). The baseline was trained on a single simulator instance, in other words, default environment dynamics parameters. UDR was trained on randomized simulated environment instances which are instantiated by sampling parameters uniformly in predefined ranges at the beginning of every training episode. Randomized domain parameters for training are defined in Section 5.4. Default parameter setting and predefined ranges are provided in Table 5.1.

Table 5.1: Training parameter setting.

Parameters	Default	Range
Gripper damping [N·m·s]	1	[0.7,1.3]
Object damping [N·s/m]	5e-4	[1e-4,1e-3]

At the beginning of every rollout, uniform noise $u \sim uniform(0, 0.01)$ was added to the initial joint velocities of UR3 to provide an initial state distribution and to avoid the same results when the same policy is applied. I trained agents with 5 random seeds for each policy (Baseline, UDR). Each agent was trained for 500 episodes and each episode consists of 1500 timesteps. For the reward function in Eq. (5.1), I set $C_1 = 0.1$ and $C_2 = 0.1$. If the agent succeeds in throwing the cube in the container box within the last 10 episodes, the task is considered to be solved and the training process is finished.

5.6 Policy evaluation scheme

To test the transfer performance of the baseline and UDR, I set up target domains with 81 different sets of parameters; 9 gripper damping coefficients are uniformly sampled as [0.2, 0.4, $\cdots$, 1.6, 1.8] and 9 object damping coefficients are log-uniformly sampled as [5e-6, 5e-5.5, $\cdots$, 5e-2.5, 5e-2] which are combined with each other to instantiate simulation environments. Note that large parts of target domains are unseen during the training phase. I rollout 5 trajectories on each target domain and evaluate the total rewards for each policy.

I design another experiment setup to investigate the robustness of DR against unmodeled effects and choose the 'object density' as the unmodeled parameter. I rollout the baseline and UDR on target domains with varying object densities while gripper damping and object damping are fixed. Since the object is set to have a fixed cube shape, the object density is linearly proportional to the 'object mass'. The baseline and UDR are trained in source domains with the constant object density of 1000 kg/m^3 and I assess the zero-shot performance of the baseline and UDR on target domains where the object density varies from 150 kg/m^3 to 15000 kg/m^3 while other parameters are the same with the default setting in Table 5.1.

6 Results

In this chapter, I enumerate the results of our Sim2Sim experiment setting. Each section in this chapter includes the discussion.

6.1 Learning curves

I obtain the learning curves comparing the performance of the baseline and UDR (Fig. 6.1). The results are averaged across 5 random seeds and shaded areas represent one standard deviation. If one of the agents solve the throwing task (based on Section 5.5) before the entire 500 episodes are done, I freeze that agent; I end the training of that agent and assume that it maintains its last return consistently until the entire episodes are done.

It is inappropriate to compare learning curves of the baseline and the UDR directly since (i) the number of episodes that agents experience is not constant for all agents because of freezing and (ii) environments change for every episode in the UDR setting. Nevertheless, those learning curves represent the whole training process clearly; we can see that UDR agents outperform in terms of overall training time and learn faster than the baseline agents. Learning curves demonstrate that DR boosts the learning process.

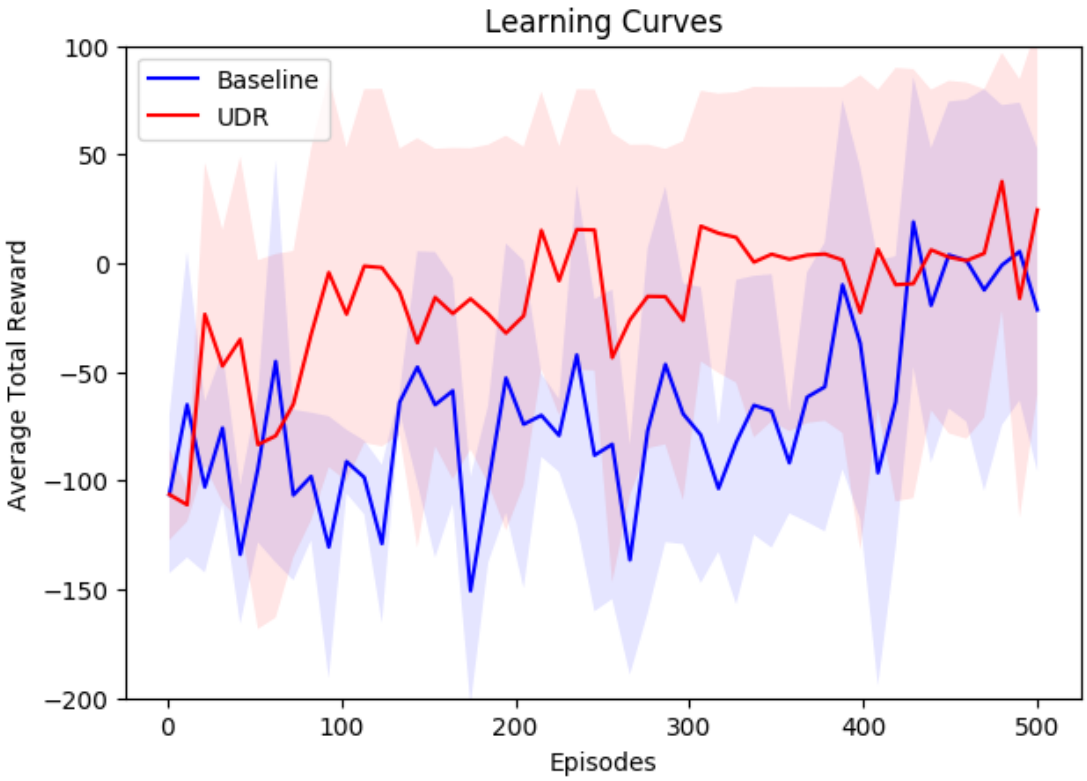


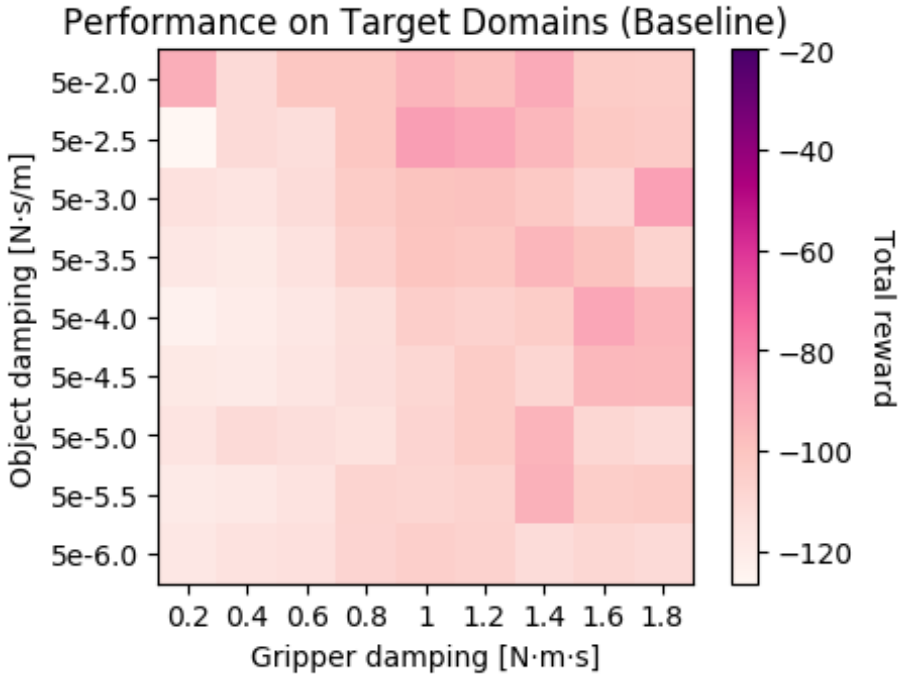
Figure 6.1: Learning curves of Baseline and UDR.

6.2 Performance on target domains

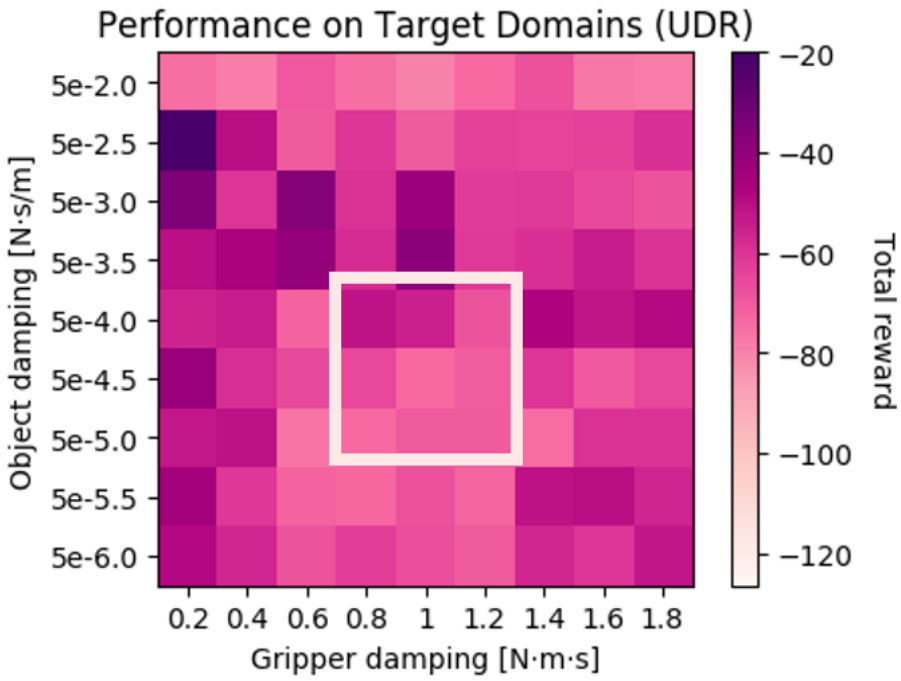
As stated in Section 5.6, target domain parameters are sampled from the wider ranges. They include the original training source domain but also include simulation environments that are previously unseen. The plots in Fig. 6.2 show the performance of policies on each target environment. A small square cell comprising the whole heatmap indicates a single instantiated target domain. The results are averaged across 5 random seeds. Thicker color means that the policy outputs higher total rewards. The white square in the middle of Fig. 6.2.(b) indicates the original source domain of UDR (Table 5.1). The areas outside of the marked square are unseen environments during training.

Both randomization dimensions (Gripper damping, Object damping) affect whether or not the cube is thrown according to the intended action. Changing the values of these parameters beyond their original ranges creates challenging environments. For example, lowering or raising the gripper damping affects the sensitivity to input torques of grippers, which makes grippers release the cube faster or slower than intended. Lowering or raising the object damping affects the momentum of the cube and the throwing trajectory of the cube, which makes the cube fly farther or shorter than intended. These effects are visualized in Fig. 6.2.

As shown, UDR outperforms the baseline for all the target domains and exhibits competitive zero-shot transfer performance to the unseen test domains.



(a)



(b)

Figure 6.2: Performance heatmap of (a) Baseline and (b) UDR. Thicker color represents higher total rewards. The object damping axis is displayed in log-scale.

6.3 Performance to unmodeled effects

Parameters for unmodeled effects that changed in the target domain are far different from the unseen domain since they are not only unseen during training but are not even randomized. As stated in Section 5.6, the parameter of the object density is set as an unmodeled parameter; it is changed only in the target domain, not in the source domain. Fig. 6.3 demonstrates that UDR shows better zero-shot performance on the environment with the unmodeled object density and outperforms the baseline over the entire range.

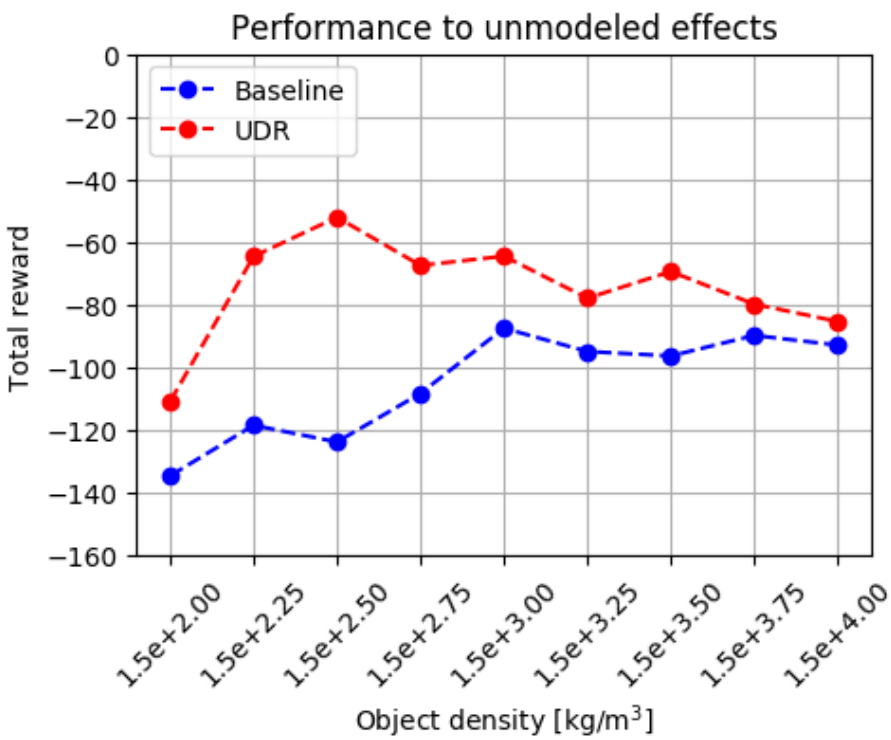


Figure 6.3: Performance to unmodeled effects of the object density. The object density axis is displayed in log-scale.

6.4 Goal-in rate

Apart from the total rewards, I also assess 'goal-in rate', i.e. whether the cube arrives inside the container box at the end of each episode, which is another major objective of the object throwing task. The results are tabulated in Table 6.1 and show that UDR succeeds in throwing the cube and putting it in the container box more while the baseline fails at almost all attempts.

Table 6.1: Goal-in rate of UDR and Baseline. 'Unseen' represents the target domain in Section 6.2 and 'Unmodeled' represents the target domain in Section 6.3.

Target domain	UDR(%)	Baseline(%)
Unseen	22.9	3.0
Unmodeled	11.6	1.8

7 Conclusion

In this work, I presented analyses about the sim-to-sim zero-shot transfer performance of an RL algorithm with DR. The source domain and the target domain were constructed by MuJoCo simulation of robot manipulator UR3. Policies were trained on the source domain with or without DR on cube throwing task and tested on two kinds of target domains: environments sampled from parameters unseen during training and environments sampled from parameters unseen and even unmodeled during training. The results show that policies trained with DR outperform policies trained without DR and can generalize better for the target domain without fine-tuning domain parameters.

For future works, I plan to set a sim-to-real (Sim2Real) transfer experiment where the source domain is the simulation and the target domain is the dynamics of real-world UR3. Modifying the method to instantiate the environment where the agent is trained is also considered.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[3] J. Jiang, C. Dun, T. Huang, and Z. Lu, "Graph convolutional reinforcement learning," *arXiv preprint arXiv:1810.09202*, 2018.

[4] F. Ramos, R. C. Possas, and D. Fox, "Bayessim: adaptive domain randomization via probabilistic inference for robotics simulators," *arXiv preprint arXiv:1906.01728*, 2019.

[5] B. Mehta, M. Diaz, F. Golemo, C. J. Pal, and L. Paull, "Active domain randomization," in *Conference on Robot Learning*. PMLR, 2020, pp. 1162–1176.

[6] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 23–30.

[7] F. Sadeghi and S. Levine, "Cad2rl: Real single-image flight without a single real image," *arXiv preprint arXiv:1611.04201*, 2016.

[8] 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)*. IEEE, 2018, pp. 1–8.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[10] A. Rajeswaran, S. Ghotra, B. Ravindran, and S. Levine, "Epopt: Learning robust neural network policies using model ensembles," *arXiv preprint arXiv:1610.01283*, 2016.

[11] Q. Liu and D. Wang, "Stein variational gradient descent: A general purpose bayesian inference algorithm," *Advances in neural information processing systems*, vol. 29, pp. 2378–2386, 2016.

[12] Y. Liu, P. Ramachandran, Q. Liu, and J. Peng, "Stein variational policy gradient," *arXiv preprint arXiv:1704.02399*, 2017.

[13] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," 2014.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[15] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

국 문 초 록

심층강화학습(DRL)을 연속 공간 상의 로봇 제어에 적용하는 문제는 지난 십수 년간 많은 관심을 받아 왔다. 실제의 로봇을 작동시켜 학습 데이터를 얻는 방식은 샘플 복잡도(sample complexity)를 높이고 안전 사고를 초래할 수 있어, 많은 경우에 로봇의 학습은 시뮬레이터로 효율적으로 대체되고 있다. 그러나, 시뮬레이션 상에서 학습된 정책(policy)은 보통 실제의 로봇의 운용에 바로 적용하기가 힘들다. 시뮬레이션의 근본적인 한계로 인해 불가피하게 생기는 시뮬레이션과 실세계 사이의 불일치 때문으로, 이 간극을 '리얼리티 갭'(reality gap) 또는 '심투리얼 갭'(Sim2Real gap)으로 부른다. 이 간극을 줄이는 방법으로 도메인 랜덤화(domain randomization) 기법이 주로 이용된다. 도메인 랜덤화 기법을 이용하여 강화학습 에이전트를 학습시키면 제로샷(zero-shot) 설정에서의 전이 능력(transferability)의 개선이 보장된다. 이는 즉 학습이 이루어지는 소스 도메인(source domain)의 범위에 포함되지 않는 환경을 타겟 도메인(target domain)으로 정하여 테스트를 진행하더라도, 추가적인 미세 조정(fine-tuning) 및 학습 없이 비교적 적정한 성능이 도출됨을 의미한다. 본 논문에서는 던지기 동작을 임무(task)로 하며, 도메인 랜덤화 기법을 학습에 이용하는 것이 서로 다른 파라미터 값을 가지는 시뮬레이션 간의 제로샷 전이에 어떤 영향을 미치는지 조사한다.

주요어 : 로봇 매니퓰레이션, 강화학습, 제로샷 학습, 도메인 랜덤화.

학번 : 2019-21027