Sim-to-Real in Reinforcement Learning Across Multiple Domains

Tim Knudsen

FH Wedel University of Applied Sciences
tim04knudsen@gmail.com

May 12, 2025

Abstract

Reinforcement Learning is a modern and powerful approach to machine learning. It has achived remarkable success in simulated environments. But no matter how powerful your algorithm is, it is only as good as the deployment in the real world. Due to the complexity of the real world, the Sim-2-Real transfer is a major challenge in the field of Reinforcement Learning. In this paper, we will discuss the challenges of the Sim-2-Real transfer and how to overcome them. We will discuss different approaches to the Sim-2-Real transfer and how they can be used to improve the performance of Reinforcement Learning.

I. Introduction

Reinforcement Learning (RL) is a machine learning paradigm in which an agent learns to make decisions by interacting with an environment and receiving feedback in the form of rewards. Through trial-and-error, the agent improves its policy to maximize cumulative rewards. In recent years, deep RL has solved complex tasks in sumulation - from playing video games to controlling simulated robots - showcasing the protential of automomous learning. However, applying these learned policies on pysical systems presents a significant challenge known as the Sim-to-Real transfer or reality gap. Simulated training is appealing because collecting real-world data can be costly, slow, or unsafe, whereas simulations provide virtually unlimited, fast data in a safe setting. Yet differences between simulated and real physics, sensors, and visuals often cause a policy that works well in simulation to fail on the real hardware.Bridging this sim-to-real gap has become a crucial research focus in fileds like robotics and atuonomous vehicles.

To address the reality gap, researchers have developed a spectrum of transfer techniques. At one end, one can improve the simulators's fidelity through *system identification* - calibrating simulation parameters to better match reality. At the other end, one can make the learned policy robust to simulation imperfections via *domain randomization* - training on a wide variety of simulated conditions so that the real world appears as just another variation. In between, *domain adaptation* methods seek to transform or adjust simulated data (or learned models) to resembele real-world data, for example by using generative models to make simulated camera images look realistic. Each approach has advantages and limitations, and often a combination is needed for succcess. Indeed, closing the sim-to-real gap is not only about transferring a policy once, but also about ensuring the policy remains reliable under real-world variability und unforeeseen conditions.

This paper provides a detailed overview of sim-to-real transfer in reinforcement learning, with an emphasis on clarity and cross-domain coverage. **Our contributions are as follows**: (1) We introduce core concepts of RL and simulation, defining the reality gap and why it arises. (2) We describe key sim-to-real transfer techniques - domain randomization, do-

main adaptation, and system identification - and illustrate how they work. (3) We survey case studies of sim-to-real success in diverse domains: robotic object manipulation, drone navigation, and legged robot locomotion. (4) We discuss current challenges (e.g. simulator fidelity, safety, generalization) and future directions such as combining methods and improving sample efficiency.

The remainder of this paper is organized as follows. **Section II** provides background on reinforcement learning and simulation environments, and defines the sim-to-real gap. **Section III** presents the main sim-to-real transfer techniques (domain randomization, domain adaptation, system identification). **Section IV** showcases application case studies across different domains. **Section V** discusses open challenges and future research directions. Section VI concludes this paper.

II. Background

i. Reinforcement Learning Basics

In a standard reinforcement learning setup, an agent interacts with an environment over discrete time steps. At each time step t, the agent observes the state s_t and chooses an action a_t according to its policy π. The environment the transitions to a new state s_{t+1} and emits a reward r_t indicating the immediate outcome of the action. The agent's goal is to learn a policy that maximizes the expected cumulative reward over time. Formally, this is often modeled as a Markov Decision Process (MDP) defined by the tuple (S, A, P, R) - state space, action space, transition, dynamics, and reward function. Modern RL algorithms (e.g. DQN, PPO, DDPG) use deep neural networks as function approximators to represent policies or value functions, allowing agents to tackle high-dimensional state spaces (like images from cameras) and learn complex behaviors.

A key requirement for successful RL is a large number of interactions with the environment to explore and learn an effective policy. Training directly on physical robots or real systems is often impractival due to time, cost, and safety constraints. For example, an autonomous drone learning via trial-and-error might crash many times before mastering flight, which is costly and dangerous. This is where *simulation environments* become extremely useful. A simulator can emulate the real environment's dynamics and sensors, enabling the agent to gather experience rapidly and safely in a virtual world. The simulator can run faster than the real time and can be reset or configured arbitrarily, providing a rich supply of training data that would be infeasible collect on real hardware.

Many popular RL environments are indeed simulations – from game engines to physics simulators. In robotics, physics engines such as **MuJoCo**, **PyBullet**, and **Gazebo** are widely used to simulate robots and their surroundings. For instance, MuJoCo (Multi-Joint dynamics with Contact) provides efficient rigid-body physics and is used for simulating robotic locomotion and manipulation. PyBullet (built on the Bullet physics engine) is an open-source simulator often used for robotics research, and Gazebo is commonly paired with ROS (Robot Operating System) for realistic 3D robot simulations, including sensors and environments. More recently, Isaac Gym (NVIDIA) introduced GPU-accelerated simulation, enabling thousands of simulation instances to run in parallel on a single GPU. This massively parallel simulation can speed up RL training by 2–3 orders of magnitude compared to traditional CPU simulators. With such simulation tools, an RL agent can effectively experience years' worth of interactions in a matter of hours or days in wall-clock time.

RL in Simulation: When training in a simulator, we typically assume we have a reasonably accurate model of the environment's dynamics. The agent is optimized on this source domain (the simulation) under the expectation that the learned policy will also perform well when deployed in the target domain (the real world). In an ideal scenario with a perfect simulator, an optimal policy in simulation would also

be optimal in reality. In practice, however, no simulator is a perfect replica of the real world. Physics engines use simplifications: for example, simulating contact dynamics, friction, or sensor noise exactly as in reality is extremely difficult. Moreover, certain real-world factors like wear-and-tear, temperature effects, or unexpected disturbances are hard to model exhaustively. As a result, a policy may overfit to the simulator's quirks and fail when those assumptions are violated in the real environment. The discrepancy between the simulator and reality is what we call the sim-to-real gap.

ii. The Sim-to-Real Gap

The *sim-to-real gap* (or reality gap) refers to the differences in dynamics, sensory inputs, and other factors between a simulated environment and the real world which can cause an RL agent's performance to drop when moving from sim to real. These differences can arise in various forms:

- **Modeling errors**: The simulator may have incorrect physical parameters (mass, friction coefficients, motor torque curves, aerodynamics, etc.) or simplified dynamics. For example, a simulator might treat a robot's joints as perfectly rigid, whereas the real robot's joints have flexibilities or backlash. Similarly, simulated cameras might produce ideal images while real cameras suffer from motion blur or lens distortion. Even high-fidelity simulators cannot capture every nuance of reality (such as minor manufacturing differences between units of the same robot, or the way cables and batteries affect a robot's weight distribution).
- **Sensor discrepancies**: In simulation, one can directly access the true state of the system (like precise object positions) or render synthetic images. In reality, sensors provide observations with noise, delay, and limited accuracy. Lighting conditions, reflections, or backgrounds in real images differ from the neatly rendered simulated visuals. An agent trained on pristine simulation images might be confused by the clutter and variability of real-world visual scenes if not properly accounted for.
- **Domain boundaries**: The real world may contain scenarios that were never present in the simulator. An RL policy trained in a simulated room might encounter novel textures, object shapes, or dynamics when applied in a real room, if the simulation did not encompass those variations. In other words, the real environment could lie outside the training distribution of the simulation. This is especially problematic if the simulator was tuned to be as "clean" and deterministic as possible for faster learning – the policy might exploit simulator-specific artifacts that do not hold in reality.

The consequence of the sim-to-real gap is often a dramatic drop in performance when a naive simulated-trained policy is tested on the real system. For instance, without special measures, a robot arm policy trained to grasp objects in a simulator might consistently miss the object in real trials, or a drone's flight controller might become unstable in actual flight due to aerodynamic effects not present in simulation. Researchers have documented that directly transferring deep RL policies from simulation to reality *without adaptation* rarely succeeds except in very constrained cases. Thus, a major area of research focuses on techniques to **bridge the sim-to-real gap** so that the benefits of simulation (abundant data, safety, speed) can be leveraged without sacrificing real-world performance.

Several approaches have emerged to tackle this transfer problem. Broadly, these approaches either (1) make the simulator more like the real world, or (2) make the learned policy robust enough to handle differences, or (3) adapt the policy (or data) when moving to real. In the next section, we will delve into the three primary techniques – *domain randomization*, *domain adaptation*, and *system identification* – that exemplify these strategies. Before that, it's worth noting that these are not mutually exclusive; in fact, state-of-the-art sim-to-real results often combine multiple techniques. We will

later see examples where improving simulator fidelity (system ID) is used together with training for robustness (randomization) to achieve successful transfer.

III. Sim-to-Real Transfer Techniques

Researchers have developed a variety of techniques to enable policies learned in simulation to work in the real world. Figure 1 illustrates three common strategies: system identification (making the sim more like reality), domain adaptation (making the data or model translate between sim and real), and domain randomization (exposing the policy to diverse simulated experiences so it generalizes to real). We will discuss each in detail, including how they are implemented and examples of their use in RL. It is important to note that these techniques are often complementary – for example, one might first apply system identification to get a reasonably accurate simulator, then use domain randomization during training, and perhaps a form of adaptation at deployment. The optimal approach can depend on the task and what aspects of the sim-to-real gap are most significant (e.g. visual differences vs. physics differences).

i. Domain Randomization

Concept: Domain randomization (DR) is a strategy where, instead of trying to make the simulator perfectly realistic, we intentionally randomize various aspects of the simulation during training. The idea, introduced by Sadeghi and Levine (2016) and Tobin et al. (2017), is that by exposing the RL agent to a wide range of environment variations, the learned policy will focus on task-relevant features and become robust to changes. If the randomization is sufficiently broad, the real world – with its particular configuration – will appear to the agent as just another variation it has seen in training. In effect, domain randomization enlarges the training domain to encompass the real domain.

What to Randomize: Practically any parameter of the simulation that could differ in reality is a candidate for randomization. Common randomizations include: object colors and textures, lighting conditions, background imagery, positions and shapes of objects, sensor noise, and physics properties like masses, friction coefficients, and joint damping. For example, in a simulated robotic vision task, one might randomize the colors and textures of walls and floors each episode, add random noise or blur to the camera images, and randomize lighting direction and intensity. In a dynamics-centric task like robot locomotion, one might randomize gravity slightly, vary the robot's motor torque scalars, or add random forces (perturbations) during the episode to simulate bumps or wind. The randomization ranges should be chosen to cover the plausible real-world variations – often initially set wide when unsure. Notably, domain randomization does not attempt to perfectly mimic reality; it instead says "throw everything at the agent, and if it can handle all of it, it will handle reality."

Effect on Learning: During training, each episode or training iteration uses a differently randomized environment. The RL algorithm, say Proximal Policy Optimization (PPO), optimizes the policy to maximize expected reward over this distribution of environments (rather than one fixed environment). Mathematically, if ξ denotes a vector of simulation parameters to randomize (both visual and physical), domain randomization seeks a policy π_θ that performs well under the expectation over $\xi \sim \Xi$, where Ξ is the space of randomization. In other words, the objective becomes:

$$\max_{\theta} \mathbb{E}\xi \sim \Xi \left[\mathbb{E}[R\tau] \right], (\xi)$$

where $R(\tau)$ is the cumulative reward of a trajectory τ sampled from the simulator with randomization ξ. By optimizing this, the agent learns to handle all the randomized conditions, rather than overfitting to one setting.

In pseudocode, a domain randomization training loop might look like this:

References