How to Train a Robot and Still Sleep at Night

Francesco G. Gillio
Department of Control and Computer Engineering
Politecnico di Torino
Turin, Italy

Abstract

This study takes a closer look at how reinforcement learning (RL) agents can learn to hop like pros—on one leg. Using the Hopper environment from OpenAI Gym and the MuJoCo physics engine, we task a robotic monopod with learning to jump, stay upright, and move forward efficiently. It sounds simple, but getting robots to learn this kind of behavior in a simulated world—and then do it in the real one—is still one of the hardest nuts to crack in modern AI.

To make progress, we put a set of policy gradient algorithms under the microscope: REINFORCE, Actor-Critic, and the widely used Proximal Policy Optimization (PPO). We examine how well they train agents to handle the challenge, weighing their strengths and limits. But learning in a perfect simulation isn't enough. Real-world physics is messy, unpredictable, and full of surprises. That's why we also explore domain randomization—specifically, an advanced method called Domain Randomization Optimization IDentification (DROID). It shakes up the simulation conditions to better prepare agents for the real deal.

By combining policy gradients with robust domain randomization, this work takes a step toward narrowing the infamous "reality gap" between simulation and the physical world. The result? Smarter, more adaptable robots that aren't thrown off by a little chaos. Along the way, we lay the foundation for sharper RL strategies and smarter ways to bridge the virtual-to-real divide in robotics.

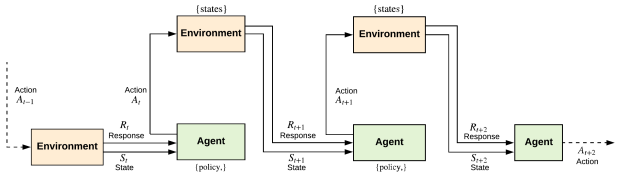
ACM Reference Format:
Francesco G. Gillio. 2025. How to Train a Robot and Still Sleep at Night. In . ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Reinforcement Learning (RL) is what happens when you throw an agent into an environment, tell it what's good and bad with a cryptic reward signal, and watch it fumble its way toward mastery. The agent doesn't know the rules. It doesn't know the map. It learns by bumping into the world, step after step. Each action it takes nudges the environment a bit—sometimes in the right direction, sometimes not. In return, it gets a pat on the back (or a slap on the wrist), encoded in a single number: the reward. Its goal? Maximize the total reward over time, a quantity we call the *return*.

That's it. That's the entire game. Reinforcement Learning methods are the bag of tricks we use to help the agent get better at playing that game.

Figure 1: The agent–environment interaction.



1.1 States and Observations

Let's start by separating what the agent *knows* from what the world *is*. A *state* s is a full, godlike description of the world—no secrets, no missing variables. But in practice, agents don't get that luxury. They see *observations* o, which are partial, filtered, noisy versions of the state. Sometimes, the two are the same. Usually, they're not.

In deep RL, both states and observations are usually encoded as real-valued vectors, matrices, or tensors. Imagine a robot. Its state might be a long list of joint angles and angular velocities. Messy, but rich with meaning.

1.2 Action Space

The agent's choices live in the *action space*. Discrete spaces are like video games—move left, jump, fire. Clean and countable. Continuous spaces are more like controlling a robot's joints: actions are real numbers, precise and subtle. Most real-world control problems fall into the continuous camp, where an agent must learn not just *what* to do, but *how much* and *how fast*.

1.3 Policy

A *policy* is the agent's brain. It takes in the state and spits out an action. If it always picks the same action for a given state, we call it *deterministic*:

$$a_t = \mu(s_t)$$

But more often, policies are *stochastic*: they roll the dice and sample from a distribution over actions:

$$a_t \sim \pi(\cdot|s_t)$$

In deep RL, these policies are parameterized—think neural networks with weights θ:

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$a_t = \mu_\theta(s_t) \quad a_t \sim \pi_\theta(\cdot|s_t)$$

When people say *"the agent does X"*, they really mean *"the policy outputs X"*. The two are basically synonymous.

1.4 Trajectories

An agent's life is a chain of states and actions, called a *trajectory*:

$$\tau = (s_0, a_0, s_1, a_1, ...)$$

The first state s_0 is sampled from some initial distribution ρ_0:

$$s_0 \sim \rho_0(\cdot)$$

Transitions between states depend on the current state and the action taken. In deterministic environments:

$$s_{t+1} = f(s_t, a_t)$$

In stochastic ones:

$$s_{t+1} \sim P(\cdot|s_t, a_t)$$

The policy tells the agent what to do. The environment decides what happens next.

1.5 Reward and Return

The *reward function* R is the only feedback the agent gets. It depends on the current state s_t, the action a_t, and the resulting state s_{t+1}:

$$r_t = R(s_t, a_t, s_{t+1})$$

We measure the agent's success by summing rewards over time—this is the *return*. For a finite horizon:

$$R(\tau) = \sum_{t=0}^{T} r_t$$

Or for infinite horizons, we introduce a *discount factor* $\gamma \in (0, 1)$ to make sure the sum doesn't explode:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

Why discount? Because in real life, a dollar today is worth more than a dollar tomorrow. And, well, math.

We can also define the return *from a particular time step* t:

$$R_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$$

Sometimes you'll see this written as G_t, but it's the same thing.

1.6 The RL Problem

So what's the grand objective? Find the policy that maximizes expected return. Formally, the probability of a trajectory under a policy is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$$

And the expected return becomes:

$$J(\pi) = \int_\tau P(\tau|\pi)R(\tau) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)]$$

The optimization problem is simple (on paper):

$$\pi^* = \arg\max_\pi J(\pi)$$

Now go find that π^*.

1.7 Value Functions

Let's say you're in state s. How good is it? That's what the *value function* tells you. If you follow policy π from there on out, your expected return is:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s]$$

But maybe you want to know how good an *action* is in a given state:

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]$$

The optimal versions assume you follow the best policy from the start:

$$V^*(s) = \max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s]$$

$$Q^*(s, a) = \max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]$$

1.8 Advantage Functions

Sometimes it's not enough to know *how good* an action is—you want to know *how much better* it is than the average. That's the *advantage*:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$

A big positive advantage? Great move. Negative? Maybe think twice.

1.9 The Optimal Q-Function and the Optimal Action

Once you've got $Q^*(s, a)$, you've basically got the world in your hands. The optimal policy just picks the action that maximizes it:

$$a^*(s) = \arg\max_a Q^*(s, a)$$

It's that simple. In theory.

1.10 Model-Free and Model-Based RL

Some agents learn by imagining. Others just try things until something works. If your agent has a *model* of the environment—meaning it can predict what happens next and what reward it gets—it can *plan*. AlphaZero does this, and it crushes humans.

But most agents don't have a model. They have to *learn* one from scratch, which is hard and often inaccurate. And if the agent starts gaming the model instead of the real world, you're in trouble.

That's why *model-free* methods are more common: no planning, just learn a good policy or value function straight from experience. Less elegant, more brute force. But they work.

1.11 What to Learn

Here's the menu: you can learn a *policy*, a *value function*, a *Q-function*, a *model of the environment*, or any combination thereof. Your choice depends on what tradeoffs you're willing to make.

1.12 What to Learn in Model-Free RL

In model-free RL, there are two main roads:

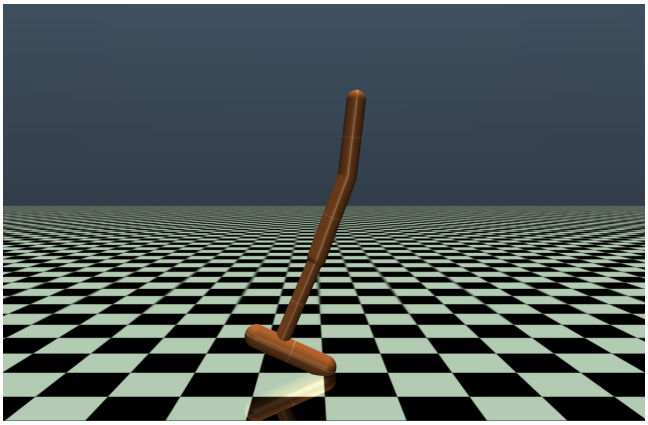
- *Policy Optimization*: You parameterize a policy $\pi_\theta(a|s)$ and tweak θ to maximize performance. Often done with gradient ascent on $J(\pi_\theta)$, and typically on-policy. You usually also learn a value function $V_\phi(s)$ to help guide the updates. Examples: A2C, PPO.
- *Q-Learning*: You learn a Q-function $Q_\theta(s, a)$ that estimates the optimal return for each state-action pair. It's usually trained off-policy, and you derive the policy from it:

$$a(s) = \arg\max_a Q_\theta(s, a)$$

2 Environment

Meet the Hopper: a quirky, one-legged robot hopping its way through a 2D world in the MuJoCo physics simulator.

Figure 2: The Hopper.



Think of it as a stick figure with ambition — made of four parts: a top torso, a thigh, a shin, and a foot — all connected by three hinges. Those hinges are where the magic happens: torque gets applied, motion emerges, and the Hopper stumbles (hopefully hops) forward.

In this study, we toy with two flavors of the Gym Hopper environment: a source domain and a target domain. What separates the two? A seemingly trivial—but strategically chosen—detail: the torso's mass. In the source, it's set to precisely 2.53429174 kg. In the target, we dial it up to 3.53429174 kg. Just one kilogram heavier, and yet, that shift embodies the very soul of sim-to-real transfer: can an agent trained in a simulator handle a heavier, messier, real-world cousin?

This is not just about teaching a robot to jump; it's about preparing it for reality, with all its non-ideal quirks. If our algorithm survives this shift, we might just be onto something robust.

Now, let's talk incentives. The Hopper doesn't just jump for fun — it's guided by a reward function composed of three distinct components:

(1) Healthy Reward: Stay alive, stay rewarded. The agent earns +1 for every time step it manages not to fall flat.
(2) Forward Reward: Movement is progress. The agent gets rewarded for moving rightward, calculated as:

$$\text{reward} = \text{weight} \times \frac{x_{t+1} - x_t}{dt}$$

where dt is the time between actions, and x is the position along the horizontal axis. Go right, get paid.
(3) Effort isn't free. Excessive or erratic actions incur a penalty — a tax on chaos — encouraging smoother, more efficient motion.

This reward architecture subtly balances survival, progress, and control. It teaches the Hopper not just to hop — but to hop smart.

3 Policy Gradient Methods

Let's get straight to it: in reinforcement learning, we don't just want our agent to act — we want it to *learn to act better*. Policy gradient methods are the scalpel for this job. They optimize the agent's behavior — its *policy* — by tuning the parameters θ in the direction that increases expected return $J(\theta)$. The tool? A gradient, of course:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)]$$

This says: *"Take the gradient of the expected reward over all possible trajectories τ, sampled from the policy π_θ"*. But that's a bit abstract. So let's unpack it into something useful:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\phi_t\right]$$

Now we're talking. This is the *likelihood ratio trick* in action: push up the log-probability of good actions, pull down the bad ones. The ϕ_t term — that's where the subtlety lies. It acts as a weight, telling us how good (or bad) each action was. And depending on the flavor of your algorithm, this weight changes.

Let's lay it out cleanly:

Policy Gradient Method	ϕ_t
REINFORCE	G_t
REINFORCE with baseline	$G_t - \beta(s_t)$
Actor-Critic	$V^\phi(s_t)$ or $Q^\phi(s_t, a_t)$
Advantage-Actor-Critic	$A^\phi(s_t, a_t) = Q^\phi(s_t, a_t) - V^\phi(s_t)$

Each algorithm is telling the same story in a slightly different dialect: estimate how good the agent's decisions are, then nudge the policy toward better ones. But the devil's in the details — and ϕ_t is where that devil dances.

4 REINFORCE

Let's talk about REINFORCE — the granddaddy of policy gradient methods. It's as pure as it gets: you take what the environment gives you and push your policy in the direction of more reward. No tricks, no shortcuts. Just math and faith in the law of large numbers.

At the core, REINFORCE relies on a *stochastic policy network* π_θ, which spits out a Gaussian distribution over possible actions given the current state s_t:

$$s_t \xrightarrow{\pi_\theta} \begin{matrix} \mu_\theta(s_t) \\ \sigma_\theta(s_t) \end{matrix} \qquad a_t \sim \pi_\theta(\cdot|s_t) = \mathcal{N}(\mu_\theta(s_t), \sigma_\theta(s_t))$$

The objective is to maximize the *expected return* $J(\theta)$ using gradient ascent:

$$J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G(\tau)] \quad \nabla_\theta J(\theta) = \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G(\tau)]$$

Now here's where the mechanics kick in. A trajectory τ — a sequence of states and actions — gets sampled by rolling out the environment according to the dynamics model $P(s_{t+1}|s_t, a_t)$ and the policy $\pi_\theta(a_t|s_t)$. The expected return becomes:

$$\mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G(\tau)] = \int_\tau P(\tau|\pi_\theta)G(\tau)$$

Differentiating under the integral sign (thank you Leibniz!), we get:

$$\nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G(\tau)] = \int_\tau \nabla_\theta P(\tau|\pi_\theta)G(\tau)$$

Now, enter the chain rule:

$$\nabla_\theta P(\tau|\pi_\theta) = P(\tau|\pi_\theta)\nabla_\theta \log P(\tau|\pi_\theta)$$

And what is $\log P(\tau|\pi_\theta)$, anyway? It's the sum of logs from all the building blocks:

$$P(\tau|\pi_\theta) = \rho_0(s_0)\prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

$$\log P(\tau|\pi_\theta) = w + \sum_{t=0}^{T}(\log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t))$$

Since θ only affects the policy, we ignore the transition terms during differentiation:

$$\nabla_\theta \log P(\tau|\pi_\theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Putting everything together:

$$\nabla_\theta J(\theta) = \int_\tau \nabla_\theta \log \pi_\theta(\tau)G(\tau)$$

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(\tau)G(\tau)]$$

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)G(\tau)\right]$$

Now here's the problem: this gradient treats *all actions equally*, rewarding them based on the total trajectory return $G(\tau)$ — even if the reward happened before the action! That's just silly.

So we fix it. We replace $G(\tau)$ with G_t, the *return from time t onward*:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)G_t\right] \tag{1}$$

In practice, this becomes an empirical average:

$$\nabla_\theta J(\theta) \approx \frac{1}{T}\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)G_t$$

Where:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$$

Then you do the classic weight update:

$$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$$

And that's it — you've got yourself a policy that improves by replaying what it saw and reinforcing what worked. But (and it's a big but) there's a catch: *REINFORCE waits until the end of an episode to update.* That means high variance, slow learning, and plenty of wasted effort in long or sparse-reward environments.

To tame this wild gradient, we subtract a *baseline*:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - \beta(s_t))\right] \tag{2}$$

This doesn't bias the gradient — it just reduces variance. Choosing the baseline $\beta(s_t)$ is an art, but a popular trick is to *whiten* the returns:

$$\beta(s_t) = \frac{G_t - \bar{G}(\tau)}{\sigma(G(\tau))}$$

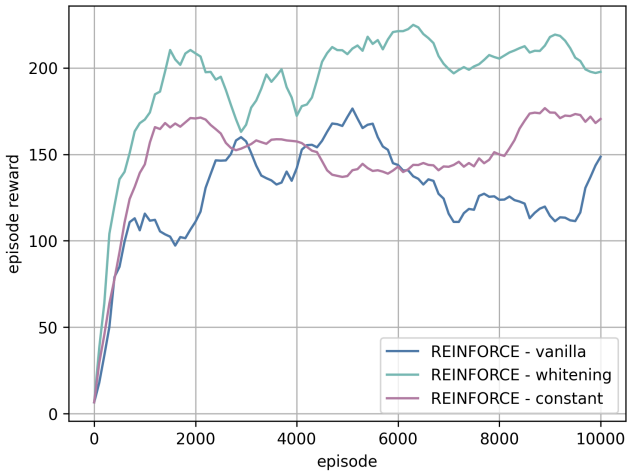
That means you're normalizing returns across the episode: zero mean, unit variance. Cleaner signal, faster learning.

So, what have we got? REINFORCE is simple, mathematically elegant, and effective — but it comes with rough edges. High variance and episodic updates make it a bit clunky for practical use. Still, it's a foundational method, and its lessons echo in every actor-critic that followed.

Algorithm 1 REINFORCE

Parameters: α, γ, $\beta(s)$
Initialize the policy $\pi_\theta(a|s)$ {with $\theta \sim \mathbb{R}^d$ at random}
for each episode **do**
  $s_0 \sim \rho_0(\cdot)$
  for each time-step $t \in T$ **do**
    $a_t \sim \pi_\theta(\cdot|s_t)$
    $f(s_t, a_t) \rightarrow r_t, s_{t+1}$
    $s_t \leftarrow s_{t+1}$
  end for
  $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
end for=0

Figure 3: Performance comparison of the REINFORCE model in different baseline configurations.



5 Actor-Critic

The Actor-Critic (A2C) algorithm is where reinforcement learning finally starts to get its act together. REINFORCE is a fine idea — take your gradients straight from the returns — but it's a noisy, jittery mess. What A2C does is simple: it brings in a second opinion. Instead of trusting the raw return G_t to guide learning, we introduce a critic, a network that learns to predict how good a state is. That's it. And it makes all the difference.

We've now got two networks. The *actor*, π_θ, which is your policy — it tells you what to do. The *critic*, V_ϕ, tells you how good the state is. The actor still outputs a Gaussian over actions:

$$s_t \xrightarrow{\pi_\theta} \begin{matrix}\mu_\theta(s_t)\\ \sigma_\theta(s_t)\end{matrix} \qquad a_t \sim \pi_\theta(\cdot|s_t) = \mathcal{N}(\mu_\theta(s_t), \sigma_\theta(s_t))$$

But now, at every time step, the critic gets to weigh in with an estimate of the value of the state:

$$V_\phi(s_t) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G(\tau)|s_0 = s_t]$$

That expectation? It's just the total return starting from state s_t, assuming we follow π_θ from there on. Now here comes the clever part. Instead of optimizing the policy using the return G_t directly, we define an advantage function:

$$A(s_t, a_t) = Q_\phi(s_t, a_t) - V_\phi(s_t)$$

This function tells us how much better (or worse) an action is compared to the average behavior in that state. You can think of it as *"extra reward"*. If it's positive, the action was a good call. If it's negative, we screwed up. And Q here is just the expected return if we start with that specific (s_t, a_t) pair:

$$Q_\phi(s_t, a_t) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G(\tau)|s_0 = s_t, a_0 = a_t]$$

In practice, we estimate Q with the empirical return G_t, so the advantage becomes:

$$A(s_t, a_t) = G_t - V_\phi(s_t)$$

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$$

Now we get to the meat of it. We want to update the policy so that it increases the log-probabilities of actions with positive advantage. That's what this gradient is doing:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)\right] \tag{3}$$

$$\nabla_\theta J(\theta) \approx \frac{1}{T}\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)$$

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J(\theta)$$

At the same time, the critic has its own job to do: approximate G_t as best it can. It does this by minimizing the mean squared error between G_t and its own guess, $V_\phi(s_t)$:

$$\nabla_\phi L(\phi) = \mathop{\mathbb{E}}_{\tau\sim\pi_\phi}\left[\nabla_\phi (G_t - V_\phi(s_t))^2\right]$$

$$\nabla_\phi L(\phi) \approx \frac{1}{T}\sum_{t=0}^{T} \nabla_\phi (G_t - V_\phi(s_t))^2$$

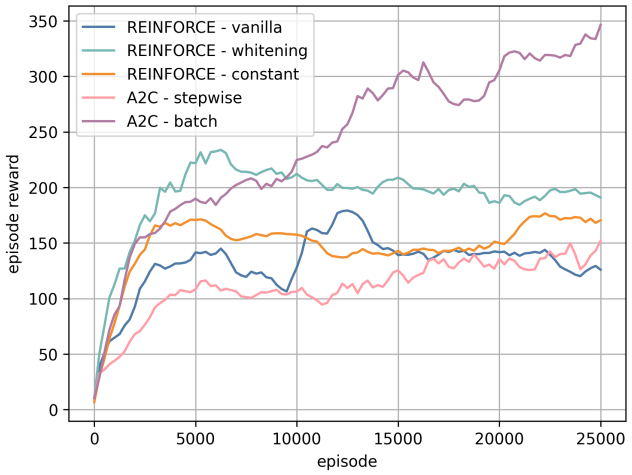
$$\phi \leftarrow \phi + \alpha_\phi \nabla_\phi L(\phi)$$

Now the training loop becomes a two-step dance: the actor pushes toward better decisions, while the critic tries to keep up and give sharper value estimates. That interaction is the core of A2C.

Algorithm 2 ACTOR-CRITIC

Parameters: $\alpha_\theta, \alpha_\phi, \gamma$
Initialize the policy $\pi_\theta(a|s)$ {with $\theta \sim \mathbb{R}^d$ at random}
Initialize the state-value function $\hat{V}_\phi(s)$ {with $\phi \sim \mathbb{R}^d$ at random}
for each episode **do**
  $s_0 \sim \rho_0(\cdot)$
  for each time-step $t \in T$ **do**
    $a_t \sim \pi_\theta(\cdot|s_t)$
    $V_\phi(s_t)$
    $f(s_t, a_t) \rightarrow r_t, s_{t+1}$
    $s_t \leftarrow s_{t+1}$
  end for
  $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J(\theta)$
  $\phi \leftarrow \phi + \alpha_\phi \nabla_\phi L(\phi)$
end for=0

Figure 4: Performance comparison of the A2C model and the REINFORCE model in different configurations.



6 Proximal Policy Optimization

Now that we've met Actor-Critic, let's take it up a notch. Enter *Proximal Policy Optimization (PPO)*, the workhorse of modern reinforcement learning. It's like A2C, but with one major upgrade: it's not reckless. PPO takes the idea of improving a policy — but says, *"Hold on, don't change too much at once"*. It uses a clever trick: clip the update so you never step too far away from where you started. Why? Because giant updates can break everything.

We still have our two main characters: the *actor*, $\pi_\theta(a|s)$, and the *critic*, $V_\phi(s)$. The actor samples actions from the policy:

$$a_t \sim \pi_\theta(\cdot|s_t)$$

The critic estimates the value of a state under the current policy:

$$V_\phi(s_t) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G(\tau)|s_0 = s_t]$$

As before, $G(\tau)$ is just the total discounted return. Nothing fancy there. And once again, we define an advantage function:

$$A(s_t, a_t) = Q_\phi(s_t, a_t) - V_\phi(s_t)$$

Same idea: if this action was better than average, boost its probability. But here's where PPO changes the game.

Instead of pushing the policy wherever the gradient points, PPO says: *"Let's be careful"*. It introduces a clipped surrogate objective. This is the core of PPO:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{s_t\sim\pi_\theta}[\min(r_t(\theta)A(s_t, a_t), \eta)]$$

$$\eta = \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A(s_t, a_t)$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

Here's what's going on: $r_t(\theta)$ is the ratio between the new and old policy. If it's greater than $1+\epsilon$ or less than $1-\epsilon$, we clip it. That way, we don't let the new policy wander too far. This avoids wild swings and keeps learning steady.

The critic still has its usual job: minimize the gap between the return G_t and its guess $V_\phi(s_t)$:

$$\nabla_\phi L(\phi) = \mathop{\mathbb{E}}_{s_t\sim\pi_\phi}\left[(G_t - V_\phi(s_t))^2\right]$$

The big picture: PPO alternates between updating the actor (carefully!) and the critic (honestly!). Both updates use stochastic gradient descent, and each iteration nudges the agent a little closer to optimal behavior. The actor gets sharper, the critic gets smarter, and the whole thing converges more reliably than older methods.

Algorithm 3 PPO

Parameters: α_θ, α_ϕ, γ, ϵ
Initialize the policy $\pi_\theta(a|s)$ {with $\theta \sim \mathbb{R}^d$ at random}
Initialize the state-value function $\hat{V}_\phi(s)$ {with $\phi \sim \mathbb{R}^d$ at random}
for each episode **do**
  $s_0 \sim \rho_0(\cdot)$
  for each time-step $t \in T$ **do**
    $a_t \sim \pi_\theta(\cdot|s_t)$
    $V_\phi(s_t)$
    $f(s_t, a_t) \rightarrow r_t, s_{t+1}$
    $s_t \leftarrow s_{t+1}$
  end for
  $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta J(\theta)$
  $\phi \leftarrow \phi + \alpha_\phi \nabla_\phi L(\phi)$
end for=0

7 Domain Randomization

Let's talk about reality. Or more precisely — how to get out of simulation and into reality without everything falling apart.

Domain Randomization (DR) is a trick. A good one. You train your agent in a simulator — sure, that's cheap, fast, and safe. But the real world isn't a simulator. It's messy, unpredictable, and full of nasty surprises. So how do you prepare your agent for that?

Easy: make your simulator messy too.

That's domain randomization in a nutshell. You deliberately inject noise, change lighting, tweak physics, vary textures — whatever you can think of. You bombard the agent with so much variety during training that, when you finally drop it into the real world, nothing surprises it. The agent thinks, *"Oh, I've seen worse"*.

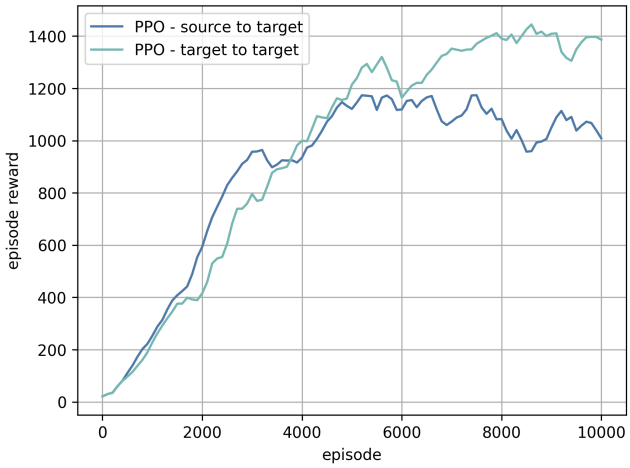
So what's the goal here? *Generalization*. By training on a swarm of randomized environments, the agent learns not just to solve a problem, but to solve the problem — regardless of small environmental quirks. That's how we shrink the dreaded sim-to-real gap.

In the next sections, we'll zoom in on two flavors of DR that take different routes to the same goal:

- *Uniform Domain Randomization (UDR)*: just randomize everything, uniformly.
- *DROID*: more refined — a method to optimize how you randomize.

But the big idea is simple: don't train in a bubble. Shake the bubble. Smash it. Make your agent robust by never letting it get too comfortable.

Figure 5: The sim-to-real gap (PPO).



8 Uniform Domain Randomization

Let's say you want your agent to face the world — but the world's a little unpredictable. What do you do? You randomize. Uniformly.

Uniform Domain Randomization (UDR) is the blunt-force version of domain randomization. You don't try to be clever. You don't try to match the real world precisely. You just add noise — across the board. And if your agent can survive that chaos, it can probably survive anything.

In this implementation, the target is the *Hopper robot*. Specifically, we shake up the masses of its moving parts: thigh, leg, and foot. The torso stays fixed — but not faithful. It's intentionally set to be off by a full kilogram compared to the real-world system. That mismatch is deliberate — a gap the agent must learn to bridge.

Here's how it works. For every training episode, we draw new mass values for each link from a uniform distribution centered around the original value $\theta_i^{(0)}$, scaled by a variation factor ϕ. Like this:

$$\theta_i \sim \mathcal{U}_\phi((1-\phi)\cdot\theta_i^{(0)}, (1+\phi)\cdot\theta_i^{(0)})$$

That means: every time the agent wakes up in training, the world feels slightly different. Heavier foot. Lighter leg. Strange thigh. It doesn't know what's coming — and that's the point.

The result? A policy that must learn to adapt. It can't overfit to one perfect world. It has to generalize across a spectrum of possibilities. And that's what makes it robust.

The only thing left to tune is ϕ — the amount of randomness. We use grid search to sweep across values and find the sweet spot.

Table 1: Grid Search for UDR

Hyperparameters	*UDR*
Variation Factor (ϕ)	{0.25, *0.5*, 0.75}

UDR is simple. Brutal. Effective. But as we'll see next, we can do better — by making the randomness smart.

9 Domain Randomization Optimization IDentification

DROID is UDR's smarter sibling. Where UDR floods the agent with uniform chaos, DROID takes a scalpel and shapes the noise — carefully, iteratively, and with purpose.

Here's the idea: rather than guessing which kinds of variations might help, DROID tries to learn the best randomization scheme by aligning the simulated world to the real one. Not blindly, but mathematically — through trajectory distributions.

9.1 Objective

At its core, DROID aims to minimize the *sim-to-real gap*. It adjusts the physical parameters of the simulator — like masses and dynamics — so that the agent's behavior in simulation looks like its behavior in the real world.

And how do we measure similarity? Enter the *Wasserstein distance* — a metric that doesn't just look at averages or variances but compares the full structure of trajectory distributions:

$$W(\mathcal{D}_{\text{real}}, \mathcal{D}_{\text{sim}}(\theta)) = \inf_{\gamma\in\Gamma(\mathcal{D}_{\text{real}},\mathcal{D}_{\text{sim}}(\theta))} \mathop{\mathbb{E}}_{(s,a)\sim\gamma}[c(s,a)]$$

Where:

- $\mathcal{D}_{\text{real}}$ is the real-world trajectory distribution,
- $\mathcal{D}_{\text{sim}}$ is the one generated by simulation parameters θ,
- $c(s,a)$ is a cost function over state-action pairs.

This isn't hand-wavy. DROID isn't just guessing the right noise — it's optimizing for it.

9.2 Method

DROID starts with some initial parameters, $\theta^{(0)}$, and proceeds through M iterations, updating θ to minimize W. In each iteration, it tweaks the parameters, runs the simulation, measures the Wasserstein distance, and updates accordingly.

Here's the loop in action:

Algorithm 4 DROID

Initialize $\theta = \theta^{(0)}$
Collect $\mathcal{D}_{\text{real}} = \{(s_i, a_i)\}_{i=1}^N$
Collect $\mathcal{D}_{\text{sim}}(\theta) = \{(s'_j, a'_j)\}_{j=1,\theta}^N$
for $m = 0, 1, 2, \ldots, M$ **do**
  $b \leftarrow W(\mathcal{D}_{\text{real}}, \mathcal{D}_{\text{sim}}(\theta))$
  for each $\theta_i \in \theta$ **do**
    $\theta_i \leftarrow \theta_i + \eta$
    $\mathcal{D}_{\text{sim}}(\theta_i) = \{(s'_j, a'_j)\}_{j=1,\theta_i}^N$
    $\mathcal{L}_i = W(\mathcal{D}_{\text{real}}, \mathcal{D}_{\text{sim}}(\theta_i))$
    $\nabla \leftarrow \nabla_{\theta_i}\mathcal{L}_i$
    $\theta_i \leftarrow \text{clip}(\theta_i, 0.01, 10.0)$
  end for
  $\mathcal{D}_{\text{sim}}(\theta) \leftarrow \{(s'_j, a'_j)\}_{j=1,\theta}^N$
end for=0

The result is a simulator that learns how to fool the agent in just the right way — not by being wild, but by being accurately wrong.

And as with any learning process, tuning matters. The learning rate η plays a key role. Too high? Unstable. Too low? Stuck. We let grid search figure it out.

Table 2: Grid Search for DROID

Hyperparameters	*DROID*
Learning Rate (η)	{1e-3, 5e-3, 5e-4, *1e-4*}

9.3 Results: Randomness Done Right

Now let's see what all this buys us.

Figure 6: Comparison of PPO model performance trends in the target environment across different training configurations.

Figure 6 compares PPO performance across different training regimes:

- The naïve baseline (just train in the source),
- UDR (uniform randomness),
- DROID (optimized randomness),
- And an ideal upper bound (trained directly in the target environment).

Here's the story:

- UDR improves generalization. It helps — but not enough.
- DROID really helps. It comes much closer to the upper bound.

Why? Because it doesn't waste its randomness. It targets it.

DROID shows us what happens when we stop guessing how to fool our agents — and start learning how to simulate the world that really matters.

10 Conclusion

Let's get one thing straight: making robots learn is hard. Getting them to behave in simulation is already a miracle. But making sure they behave the same way in the real world? That's where the real trouble starts. That's the notorious Sim-to-Real gap—and this work has tried to throw a rope across it.

We began by unleashing a few classic RL algorithms—REINFORCE, Actor-Critic, PPO—on the Hopper robot. Some were slower, some smarter, but all had the same task: hop forward and don't embarrass yourself. That alone required tuning, patience, and watching a surprising number of robots fall on their faces.

But the real lesson lies elsewhere. This study dove headfirst into Domain Randomization. Not just because it's trendy, but because the real world doesn't care about your perfect simulator. So we tried Uniform Domain Randomization (UDR): shake up the robot's parameters, let it train under chaos, and hope it becomes wise. It worked—sort of. The robot got tougher, learned to deal with variance, and didn't freak out when things changed.

Then came DROID. No more guessing. Instead of tossing randomness and hoping for the best, DROID fine-tunes the simulator itself. Adjusts parameters until simulation feels like reality. It's like replacing a blindfolded boxer with one that can see—suddenly, performance improves, and your agent doesn't just survive the real world—it thrives.

So what's the takeaway? Domain Randomization matters. Done carelessly, it helps. Done scientifically, it transforms. DROID shows us that we don't need to make agents stronger by toughening them blindly—we can make them smarter by shaping the world they train in.

There's more work ahead. Smarter algorithms, better simulators, finer-tuned randomness. But one thing's clear: the path to real-world robotics doesn't lie in perfect models—it lies in training for imperfection.

References

[1] OpenAI, *Introduction to Reinforcement Learning*. Available at: https://spinningup.openai.com/en/latest/index.html, 2018.

[2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan and D. Hassabis, *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, arXiv preprint arXiv:1712.01815, 2019.

[3] E. Todorov, T. Erez and Y. Tassa, *MuJoCo: A Physics Engine for Model-Based Control*, in *Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vilamoura-Algarve, Portugal, 2012, pp. 5026–5033, doi: 10.1109/IROS.2012.6386109.

[4] R. J. Williams, *Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning*, *Machine Learning*, vol. 8, no. 3, pp. 229–256, 1992. GitHub: https://github.com/DLR-RM/stable-baselines3.

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver and K. Kavukcuoglu, *Asynchronous Methods for Deep Reinforcement Learning*, arXiv preprint arXiv:1602.01783, 2016. GitHub: https://stable-baselines3.readthedocs.io/en/master/modules/a2c.html.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, *Proximal Policy Optimization Algorithms*, arXiv preprint arXiv:1707.06347, 2017. GitHub: https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html.

[7] 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*, arXiv preprint arXiv:1703.06907, 2018.

[8] Y.-Y. Tsai, H. Xu, Z. Ding, C. Zhang, E. Johns and B. Huang, *DROID: Minimizing the Reality Gap Using Single-Shot Human Demonstration*, arXiv preprint arXiv:2102.11003, 2019.