Gradient based adversarial domain randomization

by

Gijs Koning

Department of Intelligent Systems
Delft University of Technology
The Netherlands
June 04, 2024

Abstract

Recent advancements in differential simulators offer a promising approach to enhancing the sim2real transfer of reinforcement learning (RL) agents by enabling the computation of gradients of the simulator's dynamics with respect to its parameters. However, the application of these gradients is often limited to specific scenarios. In this thesis, we address these limitations by proposing methods to obtain accurate gradients through the use of a privileged value function. This approach provides valuable insights into the effectiveness of differential gradients and demonstrates that, in certain cases, it can significantly improve sim2real performance. To illustrate this, we develop an adversary that identifies the worst-case domain parameters for a given policy using local gradients. Our experiments are conducted on the Pendulum swing-up environment. This thesis forms the basis for the exploration of further possibilities of leveraging differential simulator gradients.

Contents

1 Introduction **1**
1.1 Main contribution . . . 2
1.2 Related Work . . . 3

2 Background **5**
2.1 Reinforcement Learning . . . 5
2.2 Deep reinforcement learning . . . 5
2.2.1 Proximal Policy Optimization . . . 5
2.3 Domain randomization . . . 6
2.4 Computing gradients through a differential simulator . . . 6
2.5 Privileged value function . . . 7

3 Method: Gradient based Adversarial Domain Randomization **8**
3.1 Introduction . . . 8
3.2 The adversary training pipeline . . . 9
3.3 Creating a non-smooth value distribution with RBF kernels . . . 10
3.4 Retrieving gradients . . . 11

4 Experimental Evaluation **13**
4.1 Experimental Setup . . . 13
4.2 The accuracy of the gradients depends on the rollout length . . . 14
4.3 A privileged value function improves the gradients . . . 15
4.4 Improved performance when training with the adversary . . . 16
4.4.1 Robust against changes in domain parameters . . . 16
4.4.2 Worst-case guided domain randomization improves robustness . . . 17
4.5 Visualizing the movement of the domain parameters . . . 19

5 Conclusion **22**

References **23**

A Hyperparameters **26**

1 Introduction

Reinforcement Learning (RL) is a method where an agent learns by interacting with an environment through trial and error. It receives feedback in the form of rewards or penalties for its actions and aims to maximize its cumulative reward over time. Reinforcement learning has been successfully applied to various complex tasks such as game playing, robotic control, and autonomous driving. One of the popular RL algorithms is the Proximal Policy Optimization (PPO) algorithm [1]. PPO is a policy gradient method that is popular because of its simplicity, stability and good performance on discrete and continous control tasks. The literature shows that it can be used to deploy RL agents in the real world by first training the agent in simulation [2, 3, 4, 5]. Training a Reinforcement Learning agent in simulation is a powerful tool to create agents to perform complex tasks while avoiding the costs and risks associated with training in the real world.

However, there is a gap in the performance of the agent in simulation and the real world [6]. Because it is impossible to simulate the real world perfectly. A lack of knowledge of how to simulate certain effects, unknown dynamics, and complexity also contribute to what is known as the sim2real gap. Another downside is that RL agents will overfit to the dynamics of the simulator including all events that aren't realistic. This means that we would like the simulator to be as accurate as possible and if possible reduce the overfitting of the RL agent to discrepancies of the simulator such that it can better generalize to the real world.

Domain randomization is a common approach to address the sim2real gap [6, 7, 8, 9, 10]. This technique randomizes the domain parameters of the environment during training such that the agent cannot overfit to a single configuration. Domain parameters like mass or friction define the dynamic behaviour of the robot. Sampling a set of these parameters with some distribution will make the agent more robust when the simulation doesn't match the actual real world dynamics precisely. Domain randomization can be seen as a regularization method that prevents the agent from overfitting to a single instance and generalize to a set of domain parameters. Here the intuition is that the real world is simply a sample in this set. The RL agents trained with domain randomization use the history of previous observations to learn to adapt to different domains without being told which ones explicitly. This is done by giving the agent a history of observations or using an RNN. For simplicity we will use a stack of the last observations in our method. It would be easier for the agent to know the domain parameters, but if this would be an input for the policy we cannot deploy it in real life since the exact parameters are not known.

The distribution that is used for domain randomization is an important and challenging hyperparameter to tune [11]. Often a multivariate gaussian or uniform distribution is chosen and the parameters of the distribution need to be optimized such that the set of training environments are diverse enough for the agent to generalize to the real-world scenarios without being so varied that the agent cannot develop a coherent and effective policy across these environments. The correlation between domain parameters should also be taken into account. And when the environment consists of many parameters this can be a difficult task to solve. Fortunately, one could find a decent distribution of the parameters by learning from data of a real robot. We will assume a fixed set of distributions and evaluate the performance of the agent on these distributions. Often a gaussian or uniform distribution is used to sample the domain parameters, however there will be certain domain parameter combinations that

simply require much more training time due to complex dynamics to get a succesful agent. Our method explores this irregularity by finding these difficult domain parameters and increases the sampling frequency to improve the robustness of an RL agent.

Existing literature shows that adversarial training can be another tool to improve the sim2real transfer [6, 12, 11, 13, 14, 15, 16, 17, 18, 19]. An adversary agent is trained to disturb the Rl agent by perturbing the actions or input of the agent or the environment itself [20] with the goal of reducing the reward of the agent. This can make the RL agent more robust to disturbances in the environment even if these are not directly modeled in the simulator. Adversarial training often presents challenges, notably involving the agent's training in a blackbox manner and the simultaneous training of two reinforcement learning agents. If one RL agent becomes too strong, it could degrade the other agent, which requires a sensitive hyperparameter to be added to the training loop. Our method can be seen as adding an adversarial agent to a regular domain randomization method that perturbs the environment by changing the domain parameters slowly over multiple episodes before resampling again.

A good example of an adversary that finds difficult dynamics for the current policy in the literature is RAMBO-RL [21]. RAMBO-RL is applied in the model-based Offline RL setting and updates its learned dynamical model within some bounds to reduce the performance of the policy. It does this only in regions where the model is more uncertain which alleviates the policy from overfitting on incorrect dynamics. We present a similar method, but instead of using the required learned dynamical model we will adapt the domain parameters of a differential simulator. Since the simulator is differential we can compute the domain parameter gradients over multiple simulation steps to get the parameters that result in a lower total reward for the current policy.

Many new differential simulators allow to compute gradients for many complex robots including contact forces. However, these gradients cannot be computed over too many simulation steps since that causes the gradients to explode or vanish [22]. We will reduce the amount of steps by bootstrapping the rollout with the value function from the PPO algorithm. This creates an important hyperparameter, since too short of a rollout will provide incorrect gradients as well. To further improve the accuracy of the gradients we will increase the accuracy of the value function with respect to the domain parameters. The literature shows that a privileged value function that incorporates the domain parameters as input improves RL with Domain Randomization training.

1.1. Main contribution

The main contribution of this thesis is an analysis of the accuracy of gradients computed through a differential simulator, a method to show how these gradients can be improved and how the gradients can be used to improve the robustness of an RL agent when using it during training. In detail, we show the effect of the gradient rollout length on the accuracy of the gradients and the improved accuracy of a privileged value function that incorporates the domain parameters. We show that a differential simulator with the right rollout length and privileged value function provides accurate gradients that find the worst-case domain parameters with respect to the current policy. Training with these worst-case parameters creates a robust RL agent with higher performance for the difficult parameters. We will show this improves a gaussian and uniform domain distribution and highly improves a less regular distribution created with RBF kernels.

We evaluate our approach on the Pendulum environment, a single action continuous control problem and compare the results to the baselines of regular PPO training without any randomization and training with domain randomization.

In summary, we make three key claims:

I We verify that rollout truncation with a value function reduces computational overhead while still obtaining accurate gradients.

II Adding privileged information to the value function leads to more accurate gradients during training.

III Training with worst-case domain parameters by moving the parameters in the direction of these gradients leads to a better worst-case performing agent than regular domain randomization. And this is particularly shown when the value landscape is non-smooth.

The experiments that backup these claims are discussed in chapter 4. The algorithm developed for claim **III** is described in chapter 3.

1.2. Related Work

One of the most common RL methods for continuous robotic systems is PPO [1], which is why it is used in our experiments.

Training an RL agent however requires a vast amount of training data, which is hard and expensive to acquire on a physical system [2]. Collecting all data in simulation is therefore appealing. However, the simulation does not capture the environment or the robot accurately in every detail. Even if we calibrate the simulation to be as close as possible to the real system, this is often not enough [23]. These differences cause a "reality gap" and make it unlikely for a policy trained in a simulation with these inaccuracies to transfer well.

To overcome the reality gap we can train on a distribution of many simulations where the parameters of each simulation is randomized. With so called Domain Randomization (DR) the trained policy doesn't overfit to a single simulation and is more likely to have success on a physical robot [23, 6, 9].

DR can lead to suboptimal, high-variance successful policies when trained under uniform distribution [7]. There is an issue in DR where the policy will optimize on a certain region of domain parameters and ignore the worst case parts, while still achieving a good average performance. Prior work, EPOpt [20], has addressed this issue by replacing the uniform average across distributions with the conditional value at risk (CVaR).

Another issue that arises with DR is the decision on the magnitude of randomization. Too large distribution can create a very conservative policy and too small doesn't transfer well. A solution is to use Automatic Domain Randomization [2]. Which in short, increases the uniform bounds of a single domain parameter at random when the success rate has achieved a minimal level.

A history of observations and actions as input to the policy allows the agent to identify the system dynamics and learn a policy that can adapt and have different behaviours for different domain parameters [24, 6]. Alternatively, system identification can be implicitly embedded into a policy by using a recurrent model [2, 9]. We use a history of past observations and actions stacked together as input instead of just the latest observation and action in our experiments due to its simplicity compared to processing the latest observation with an recurrent network which requires more tuning to work well.

An asymmetric Actor-Critic [25, 23] provides additional information aside from the environment observations to the critic compared to the actor at training time. It exploits the fact that the value network can have access to information from the simulator that is not available on the real robot system. This so called Privileged value function potentially simplifies the problem of learning good value estimates since less information needs to be inferred. Similar ideas in the literature provide additional information like friction to be accessed by the value function [5] or add noiseless observations and poses of all fingers [2, 23]. Our method also benefits of the additional information to the value function.

Adversarial training proposes a minmax RL problem. A second agent is added to reduce the reward of the main agent or protagonist. This second agent is often learned using RL as well. By learning the adversary to perturbate the environment it can make the main agent more robust to fixed domain changes. [12, 26, 27]

There is a trade-off between improving the worst-case reward of a policy and its natural mean reward. A policy will become more conservative when it is forced to increase its worst-case reward [13]

One downside noted in the literature is that an RL agent can overfit to the counteractions of the adversary if the adversary is a deterministic function of the state. A solution is to limit the amplitude of the actions [27] or use multiple adversaries and sample one at random [11]

There are alternatives to training a second RL agent. A random adversary network that applies state- and action based noise to observation and simulation dynamics is apparently better and easier. [2]

Our method is an alternative to training a second agent and reduces overfitting to constant adversary actions because the adversary actions is based on the latest policies trajectory and reward of the value function.

New differential simulators allow us to compute gradients in complex robotic environments [28, 29, 30, 31]. Not all simulators provide the same features. For example, some only provide gradients over actions and others are difficult to create randomized environments.

Gradients with robotics systems or rigid body systems are notoriously difficult to treat differentiably [32]. The SHAC algorithm [33] avoids chaotic gradients by subdividing trajectories in short optimization windows that are bootstrapped with the value function. For a smoother reward function and a reduction of memory footprint. Our method is inspired by SHAC since we compute gradients over a short horizon by bootstrapping the trajectory with the value function, such that we get smoother gradients and reduce

computation needs. One could also replace a differential simulator with a learned model such that it is differentiable. But in this thesis, we focus on differential simulators only.

2
Background

This chapter starts by describing relevant notation for Reinforcement learning with domain randomization and gradients from differential simulators. It follows by introducing RL, including specific concepts in RL that are relevant to our work, such as value functions, deep RL and PPO. Finally, it adds detail to domain randomization in RL, how to compute gradients through a simulator and the use of a privileged value function.

2.1. Reinforcement Learning

Reinforcement Learning helps agents solve decision problems such as Markov Decision Processes (MDPs) and Partially Observable Markov Decision Processes (POMDPs) [34]. An agent learns to act using trial-and-error without knowledge of the underlying transition and reward models. At time t, the agent interacts with the environment by taking actions often according to a stochastic policy $\pi(a_t|s_t)$, based on the current state s_t. By continuously interacting with the environment and receiving new states and rewards, the agent learns an optimal policy to maximize the expected future return.

In RL, it is common to learn a state-value function V(s) and/or an action-value function Q(s, a) alongside the policy π. For MDP, the state-value function $V_\pi(s)$ represents the expected return starting from s and following the states generated by expected actions from the policy π thereafter. Similarly, the action-value function Q_π(s, a) represents the expected return starting from s, taking a, and following π thereafter. For POMDPs, since the state s is not observable, the observation o is used in learning these value functions $V_\pi(o)$ or policy Q_π(o, a). Relying solely on the last observation is suboptimal; therefore, most algorithms use a sequence of recent observations or employ a Recurrent Neural Network as an encoder to capture and memorize useful features from past observations.

Traditional RL methods use tabular representation or simple function approximation with hand-crafted feature construction to represent the value function. In contrast, Deep Reinforcement Learning (DRL) techniques such as DQN [35] and DDPG [36], learn the value function and policy in an end-to-end manner without feature engineering. DRL leverages deep neural networks to automatically learn feature representations, making it easier to generalize to new scenarios.

2.2. Deep reinforcement learning

Deep learning, within machine learning (ML), involves using deep neural networks (DNNs) [37]. In deep reinforcement learning (DRL), agents utilize DNNs as function approximators. An early successful example of this integration is the deep Q-networks (DQN) algorithm [38], where a DNN is employed to learn optimal policy Q values. Various other approaches have emerged, encompassing functions such as value and policy functions [39], as well as a dynamic model of the environment [40].

2.2.1. Proximal Policy Optimization

Proximal Policy Optimization (PPO) [1] is a prominent reinforcement learning algorithm known for its stability and efficiency. Which is the reason its used in this work. PPO operates in the policy gradient framework, aiming to maximize the expected cumulative reward by adjusting policy parameters while

ensuring controlled updates within a trust region.

The algorithm in simplified form of PPO is as follows: (source: pseudocode ppo [41]):

Algorithm 1 PPO-clip

1: **Input:** Initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment
4: Compute rewards-to-go $\hat{R}_t = \sum_{k=t}^{H} \gamma^{k-t} r_k$
5: Compute advantage estimates, $\hat{A}$ based on the current value function V_{ϕ_k}
6: Update the policy by maximizing the PPO-Clip objective:
7: $\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \min\left(A_t \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, A_t \text{clip}(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, 1+\epsilon, 1-\epsilon)\right)$,
typically via stochastic gradient ascent with Adam.
8: Fit value function by regression on mean-squared error:
$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2$, typically via some gradient descent algorithm.
9: **end for**

With A_t being the advantage function: $A^{\pi_\theta}(s_t, a_t)$.

2.3. Domain randomization

Domain randomization is a technique to improve sim2real performance for agents that are trained in a simulator [6]. In domain randomization, a set of parameters denoted as c is subject to randomization. These parameters, like lighting intensity, mass or other object dynamics, are sampled from a distribution, $c \sim p(C)$, covering a wide range of variations. In our experiments with the pendulum environment $c = \{m, l\}$ with m and l being some positive decimal value when we use a Gaussian or uniform distribution. For the RBF distribution $c = \{m1, m2, m3, l1, l2, l3\}$ (further details in Section 3.3).

Mathematically, the goal is to maximize the expected return of an agent's policy, π_θ, across randomized environments:

$$J(\pi_\theta) = E_{c\sim p(C)}[R(\pi_\theta, c)] \tag{2.1}$$

Here, $R(\pi_\theta, c)$ represents the cumulative reward when the agent follows policy π_θ in an environment with parameters c. The agent is trained to handle real-world conditions by optimizing π_θ using gradient-based methods like PPO.

2.4. Computing gradients through a differential simulator

Similar to SHAC [33] which uses the differential simulator gradients to improve the PPO algorithm, our method computes the future reward gradients over multiple simulation steps and this is only possible when using a differential simulator. Their policy loss function when minimized improves the policies parameters such that its actions will obtain a higher reward. It is defined as [33, eq. (5)]:

$$\mathcal{L}_\theta = -\frac{1}{Nh} \sum_{i=1}^{N} \left[\left(\sum_{t=t_0}^{t_0+h-1} \gamma^{t-t_0} \mathcal{R}(\mathbf{s}_t^i, \mathbf{a}_t^i) \right) + \gamma^h V_\phi(\mathbf{s}_{t_0+h}^i) \right] \tag{2.2}$$

We use a similar loss function but instead minimize the return with respect to the domain parameters and calculate the loss for a single part in a trajectory:

$$\mathcal{L}_c = \left(\sum_{t=t_0}^{t_0+h-1} \gamma^{t-t_0} \mathcal{R}(\mathbf{s}_t, \mathbf{a}_t) \right) + \gamma^h V_\phi(\mathbf{s}_{t_0+h}) \tag{2.3}$$

$\mathbf{s}_t^i$ and $\mathbf{a}_t^i$ are the state and action at time step t, γ is the discount factor, $\mathcal{R}$ is a differential reward function (a requirement for our method) and V_ϕ is the value function with parameters that remain unchanged during updates. The actions are sampled from the current policy $a_t \sim \pi_\theta(a_t|s_t)$. The next state is obtained using the dynamics function of our differential simulator $s_{t+1} \sim P(s_{t+1}|s_t, a_t, c)$. Which

shows that the simulator depends on the domain parameters c (mass and length of the pendulum in our experiments), without that input the gradients on c would not exist. To compute the gradient of the domain parameter loss $\frac{\partial \mathcal{L}_c}{\partial c}$, we treat the simulator as a differentiable layer and perform regular backpropagation.

2.5. Privileged value function

A privileged value function is privileged because it is trained with more available information than the policy function. This is possible since the value function can have access to information from the simulator that is not available on the real robot system [2]. We provide the value function with the current mass and length parameters of the simulator to enhance the prediction accuracy of the value function. This changes the value function in Eq (2.3) to:

$$\mathcal{L}_c = \left(\sum_{t=t_0}^{t_0+h-1} \gamma^{t-t_0} \mathcal{R}(\mathbf{s}_t, \mathbf{a}_t) \right) + \gamma^h V_\phi(\mathbf{s}_{t_0+h}, c) \tag{2.4}$$

This different way of training an actor-critic algorithm is also coined to be Asymmetric Actor Critic [25].

3

Method: Gradient based Adversarial Domain Randomization

In this chapter, we present our approach that will support the three claims made in the introduction. We first describe the algorithm that finds the worst-case domain parameters (or contexts) with respect to the current policy. Then, we describe how we can use this adversary to improve the robustness and sample-efficiency of an RL agent and show that it effect is most noticable when the distribution is non-smooth with RBF kernels. Finally, we detail how we obtain the value gradients with respect to the domain parameters of a differential simulator using a truncated rollout by bootstrapping with the value function.

3.1. Introduction

The idea of our method is to find and use the worst-case domain parameters under the current policy. By running the environment with the worst-case parameters more, the minibatch will have a higher percentage of difficult experiences and this forces the PPO algorithm to improve in these regions.

Our method only modifies the rollout of an RL algorithm combined with domain randomization. The domain parameters are sampled from a distribution every few episodes. During rollout of these episodes we compute the gradient of the current domain parameters with respect to the future reward and update the domain parameters towards the worst-case parameters. This move towards the worst-case parameters is done multiple times during a rollout and causes the rollout buffer to contain more difficult experiences. The benefit of only adapting the rollout is that our method could be applied to any RL algorithm that has a value function.

Our method is based on the idea that we can use the gradients of the future reward and the value function with respect to the domain parameters to find the worst-case domain parameters under the current policy. Since the environment is a differential simulator we can compute the gradients over multiple steps of the simulator.

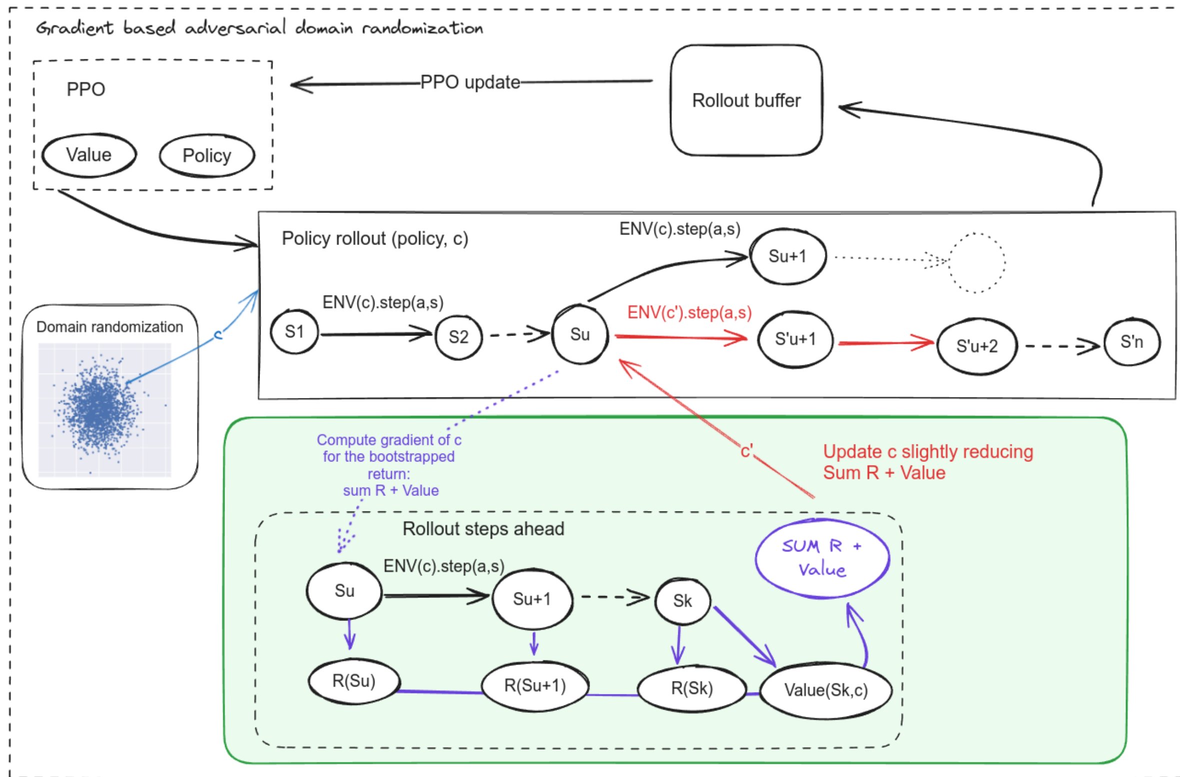


Figure 3.1: Our modified PPO training pipeline. We modify the rollout of trajectories by updating the context parameter of the environment multiple times in the trajectory with the Adversary update (green). The latest policy and privileged value network and the differentiable simulator are used to calculate the derivative of the bootstrapped return for K steps. We use the derivative to update the context parameter C to make the future return smaller. Providing the context parameters to the value function results in a better prediction of future rewards.

3.2. The adversary training pipeline

This section introduces the adversarial training loop which trains an RL agent with PPO and makes it robust by changing the domain parameters during training such that they become more difficult for the current policy to perform well. In short, the method adds an adversarial step to a regular PPO + Domain Randomization implementation by changing the sampled domain parameters towards worst-case domain parameters. We modified the PPO implementation of Brax to work with Domain Randomization and added the adversary update step.

The full algorithm is described in Algorithm 3. With Algorithms 2 and 4 describing the adversary context update and Compute bootstrapped return functions respectively. Figure 3.1 provides the general overview of the algorithm. In green the adversary update, our addition that modifies the context of the environment during trajectory generation.

Our method only adapts the rollout of an RL algorithm and requires access to a value or Q function. This means that it can be applied to most common RL algorithms that have a value function. For all our experiments we use PPO since it is one of the most common methods.

The adversary update The adversary update that is called multiple times during episode rollout uses the gradients calculated as in Algorithm 4 to update the current domain parameters. This means the domain parameters are updated multiple times while running the episode. We explain the frequency of these updates in the next section. We use ADAM [42] as optimizer that updates the domain parameters. There are two hyperparameters G and a that are used to define precision of adversary update. We use G to define the amount of optimizer updates that are done when are doing a adversary update and learning rate a defines the step size of a single gradient update. Multiplying G while dividing a with the same factor can be seen as taking multiple steps with a smaller step size. This allows the optimizer to update the domain parameters more precisely. However, the downside is the increase in computation requirements. That is why we choose G to be 2 and a 0.005 in all experiments. The idea of taking two steps is that it allows the optimizer find a better minimum instead of the alternative of taking a bigger

single step by doubling the learning rate.

Big spikes of gradients can happen irregularly, we avoid this by clipping the gradients with a maximum gradient size G_{max}.

Training pipeline See Algorithm 3 for the pseudocode of the full training algorithm. The algorithm is similar to PPO and the only adaption to the algorithm happens during the rollout of new trajectories.

Every U steps the adversary context update is executed which modifies the current context c. Note that when calling this function the domain parameters are updated G times. This context will be used for the next U steps. We set U to be 5 in our experiments. It's a balance between a lower number causing too much computation overhead or too high reducing the amount of times the adversary is updated.

Algorithm 2 Adversary context update: adversary_update()

Require: c, s, adversary learning rate α, gradient updates G, maximum gradient size G_{max}
1: Assume compute_bootstrapped_return has access to π_θ, V_θ, env, H
2: **for** $g = 0$ to $G - 1$ **do**
3: $\quad grads_c \leftarrow$ jax.forward_grad(compute_bootstrapped_return)(c, s) ▷ Alg 4
4: $\quad c \leftarrow c - \alpha * \text{clip}(grads_c, -G_{max}, G_{max})$
5: **end for**
6: **Return** c

Algorithm 3 Full training algorithm

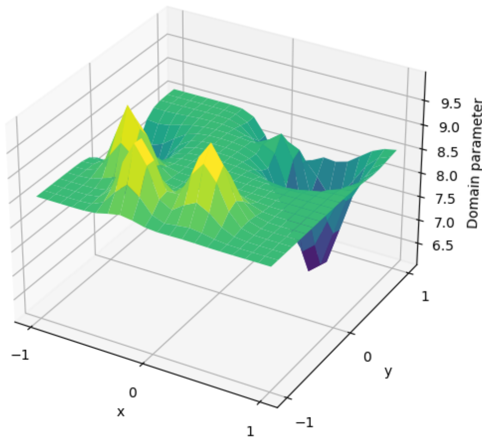
Require: steps per adversary update U, episode length E, DR distribution X, new context frequency F
1: Initialize V_θ , π_θ for PPO
2: $c \leftarrow c \sim X$ ▷ Sample new domain parameters
3: **while** not converged **do**
4: $\quad s_0 \leftarrow env$.reset()
5: $\quad$**for** $i = 0$ to $E - 1$ **do**
6: $\quad\quad$**if** rand(0, 1) $< F$ **then**
7: $\quad\quad\quad c \leftarrow c \sim X$
8: $\quad\quad$**end if**
9: $\quad\quad$**if** i divisible by U **then**
10: $\quad\quad\quad c \leftarrow$ adversary_update(s_i, c, V_θ , π_θ) ▷ Alg 2
11: $\quad\quad$**end if**
12: $\quad\quad s_{i+1} \leftarrow env$.step($c$, s_i, $\pi_\theta(s_i)$)
13: $\quad$**end for**
14: $\quad$Collect trajectories $\tau(s_0, s_{i+1} + \ldots s_E)$
15: $\quad V_\theta, \pi_\theta \leftarrow$ Update θ using PPO update and τ
16: **end while**

The use of a privileged value function Algorithm 3 already uses a privileged value function. It's not clear directly. The context parameters c are used in the adversary_update function (Algorithm 2), which uses the gradients obtained from the compute_bootstrapped_return function to update the context parameters with multiple gradient steps. Internally this function uses the agent's value function to predict future reward. By adding the context parameters as inputs to the value function $V_\theta(s)$ next to the observation we obtain a privileged value function: $V_\theta(c, s)$. Which has a higher accuracy in predicting future rewards as is shown in our experiments 4.3

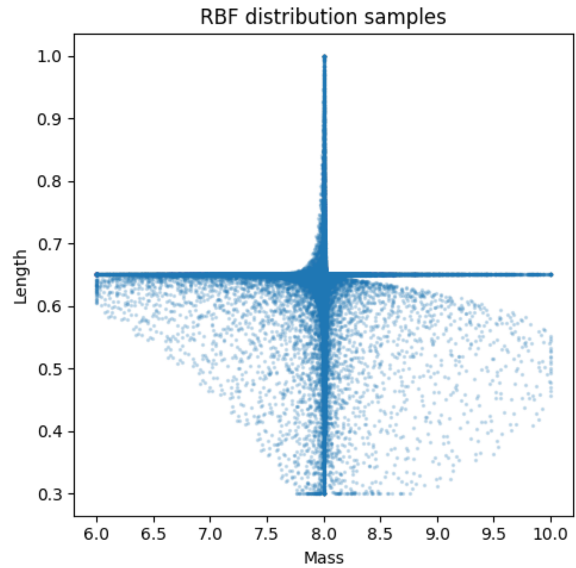
3.3. Creating a non-smooth value distribution with RBF kernels

We use the pendulum environment as the problem for our solution and its domain parameters that the RL agent should adapt to are the mass and length of the rod. When sampling the mass and length using a gaussian or uniform the distribution the value distribution is actually very smooth. Only the outer region towards a high mass and length is difficult to learn because of the amount of swings needed. To improve this region one can simply increase the width of the distribution towards this region.

To better show the added benefit of our method which automatically finds the difficult spots, we want an environment with a domain parameters where the difficulty is not evenly distributed and



(a) XY -> domain parameter function. Example with two uniform parameters x and y that are sampled between -1 and 1. The z axis is resulting domain parameter that is returned by the function and its output in this example is bounded between 6 and 10. The function is a combination of 8 kernels that are centered around random points in the x,y plane with random widths between 0.1 and 0.2



(b) The new sampling distribution of the mass and length parameters that is used during training by transforming two sets of 3 uniform parameters to the original domain parameters.

thus is difficult to improve by manual selecting a distribution for domain randomization. This is why we use radial basis function (RBF) kernels to create a non-linear distribution by transforming a set of new sampled parameters called c_{rbf} to the original domain parameters while being differentiable. In the Background section 2.4 we showed that we are differentiating through the simulator function $P(s_{t+1}|s_t, a_t, c)$ with c as the original parameters. With the RBF function being differentiable we can do the same, but replace c with $RBF(c_{rbf})$ resulting in: $P(s_{t+1}|s_t, a_t, RBF(c_{rbf}))$.

We define the RBF function for every original domain parameter. Each function has multiple random rbf kernels that transform three uniform $(-1, 1)$ parameters to a single value. The output is scaled such that all samples are within a reasonable defined bound of the original domain parameters. This bound is the same as the size of the grid we use for evaluating the agent.

A single RBF kernel is a gaussian kernel that is centered around a point and has a width σ. The RBF kernel is defined as:

$$K(x, x') = \exp\left(-\frac{||x - x'||^2}{2\sigma^2}\right)$$

An example of a transformation function that only transforms 2 uniform parameters is shown in Figure 3.2a. The new distribution that is used during training for our experiments is shown in Figure 3.2b.

How does this affect the gradients of a rollout and the inputs of the value function? Instead of sampling two parameters from a gaussian distribution, we have a new set of 3x2 parameters that are sampled uniformly between $(-1, 1)$. Since the transformation functions use rbf kernels that are differentiable we can use the same method as before with the only exception that we need to transform the parameters before we execute the environment step. This way we use the same domain parameters for the simulator but the sampling distribution is different. This change with new and more parameters also affects the input to the privileged value function. But since the value function is a neural network we can simply add more inputs to the network and train it to use these inputs.

3.4. Retrieving gradients

With the differential simulator Brax [28] and the Autograd [43] of Jax [44] library we can compute gradients through steps of the simulator. Brax as default doesn't provide gradients over the domain parameters. But it can be modified by changing the reset and step functions to include the variable domain parameters as inputs and use these to update the internals of Brax at every step with these parameters. This allows us to update the parameters at any step, changing the internal dynamics of the simulator.

We then obtain the gradients of our domain parameters over a rollout of steps of the policy. To change the future return of the policy we take the gradients with respect to the sum of future reward of the policy rollout. We will only compute gradients over a short horizon of steps so the remaining future reward needs to be bootstrapped with the value function. Jax and its Autograd features makes this relatively simple to compute the gradients. We use the forward jacobian function provided by Jax instead of the more commonly reverse mode version, since they provide better support for the Brax internal loop functions and are efficient when the number of parameters are small [45]. See Algorithm 4 for the pseudocode. Forward mode starts from the input variables and works its way forward through the computation graph, computing both the function value and the derivative with respect to each input variable at the same time.

Algorithm 4 Example of computing gradients of the future return with respect to the domain parameters

Require: Contexts c, policy π_θ, value V_θ, environment env, prediction horizon H, discount γ

1: $s \leftarrow env$.reset()
2: **function** COMPUTE_BOOTSTRAPPED_RETURN(c, s)
3: $return \leftarrow 0$
4: **for** $h = 0$ to $H - 1$ **do**
5: $s, r \leftarrow env$.step(c, s, $\pi_\theta(s)$)
6: $return \leftarrow return + \gamma^h r$
7: **end for**
8: **Return** $return + \gamma^H V_\theta(c, s)$
9: **end function**
10: $grads_c \leftarrow$ jax.forward_grad(compute_bootstrapped_return)(c, s)
11: **Return** $grads_c$

4

Experimental Evaluation

The main focus of this work is to show that gradients of a differentiable simulator can be used to create an adversary that improves robustness of an RL agent.

We present our experiments to show the capabilities of our method and to support our key claims, which are:

I We verify that rollout truncation with a value function reduces computational overhead while still obtaining accurate gradients.

II Adding privileged information to the value function leads to more accurate gradients during training.

III Training with worst-case domain parameters by moving the parameters in the direction of these gradients leads to a better worst-case performing agent than regular domain randomization. And this is particularly shown when the value landscape is non-smooth.

The three sections following the experimental setup in this chapter prove the claims respectively:
Section 4.2, provides data to show the affect of the rollout length to the accuracy and computation requirements. It shows how increasing the rollout length benefits the accuracy of the gradients since it reduces the dependency on the value function. However, a balance needs to be found since the value function reduces the reward variance coming from the non-linear simulation dynamics, since its an approximation of the expected reward.
Section 4.3, shows that a privileged value function has a better accuracy in predicting the future reward and therefore creates better domain parameter gradients for the adversary.
Section 4.4, presents the final results of our method. First, it demonstrates the increased robustness to any domain parameter changes when training with the adversary. Then, it shows that our method, which identifies the worst-case domain parameters during training, significantly enhances the RL agent's performance in challenging regions.

4.1. Experimental Setup

We will evaluate our method on the Pendulum Swing-up environment implemented in the Brax differentiable simulator. We choose Brax since it uses the simple Python language, can be optimized with Jax to run many environments in parallel and with modifications it allows to differentiate over domain parameters. Our baseline is PPO with Domain Randomization and a privileged value function. Our method extends the baseline by adding an adversary in the episodes generation that changes the domain parameters in the direction of the worst-case gradients every so many steps. Both the baseline and our method will be evaluated on the same distributions of mass and length and for each experiment we show the success rate for these domain distributions: Gaussian, Uniform and RBF. We will train each method for 9 agents with different initial seeds such that we can show the variance of the performance and make a better claim of the significance in performance difference. For all barplots we visualize the 95% confidence interval.

The hyperparameters for PPO, the distribution details and other training properties are laid out in the tabel in the appendix A.1.

Success metric An episode is successful when the Pendulum is upright and stable at the last 30 steps (out of 320) of the episode. We base the success metric on the reward penalty of the environment which is 0 when the pendulum is upright with no velocity. An average of a too high velocity or angle for the last 30 steps can cause a failure. We choose to use a success metric over the reward since the return over the whole episode is highly dependent on the domain parameters. For example, a higher mass for the Pendulum needs more swings which results in a lower total reward. This is why a minimal reward for the last 30 steps much better reflects the performance of the agent.

4.2. The accuracy of the gradients depends on the rollout length

We want the most accurate gradients with respect to the domain parameters such that our adversary moves the domain parameters in the most difficult direction. We can increase the accuracy with a longer rollout length for calculating the gradients. A longer rollout length will reduce the dependency on the value function and therefore the gradients will be more accurate, especially in the early stage of training when the value function is not learned well.

The idea is that the predicted return with a longer rollout will be more accurate since it uses the value function less by means of discounting. Figure (4.1, left) shows the increased accuracy when using a longer rollout when predicting the value with some rollout steps and the value function. The value is predicted just after we change the domain parameters a small step in some random direction for each parameter. The plot shows the relative error of the predicted value compared to the true return.

Interestingly this is a contradiction compared to [33] where they show that a longer rollout length reduces the accuracy of the gradients. A possibly explanation is that the dynamics of our environment are simple enough to not see the downsides of complex gradients with a longer rollout.

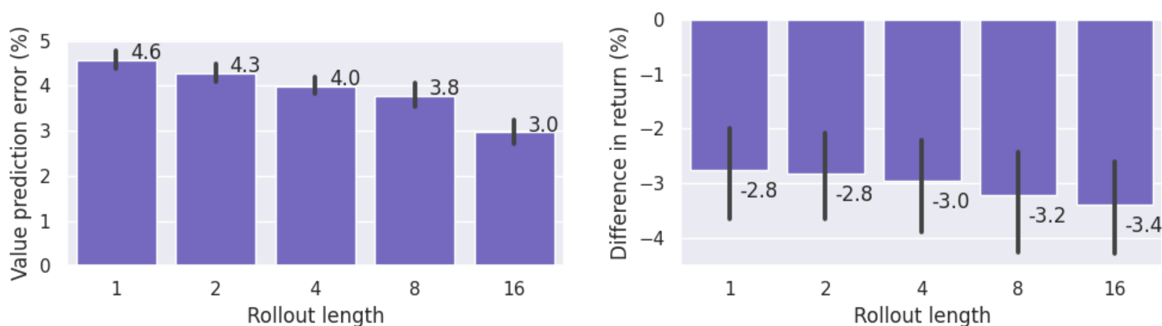


Figure 4.1: Visualizing the effect of the rollout length when predicting the return. We show the results for 3 different initialized trained agents. The rollout length only has an effect during plotting. Furthermore, we average the measurement over a trajectory from 20 initial states and for every 5 steps in the trajectory. The y-axis shows the percentage relative to the actual return (the sum of reward of the full trajectory) Left: The prediction error between the true return and the predicted return using the discounted sum of the reward of N rollout steps and the return predicted with the value function. We want to measure the accuracy of the gradients of changing parameters, therefore we change the parameters a small step in every direction and measure if the prediction is still accurate. Specifically, all vectors obtained from the product of -1, 0, 1 for every parameter (mass and length). This update is done for every trained agent, over 20 different episodes and for every 10th step. Right: The difference in return after a single attack from the adversary using gradients to step towards more difficult domain parameters (Lower is a stronger attack).

This result is the first step in showing why it is useful to calculate the gradients through the simulator. The increased accuracy is also reflected when we actually use the gradients to step towards the worse performing domain parameters. This is shown in Figure (4.1, right), the increase in accuracy of the gradients correlates with a bigger difference in return after a gradient step. A stronger attack is preferred since this means the adversary moves the domain parameters in the direction of the worst-case parameters.

Because we use PPO with a non-deterministic policy it is good to mention that when predicting the future return by rolling out the trajectory for a few steps, we take the same actions as the original trajectory would do. This is possible with Jax because we call the policy with the same sequence of keys used for sampling the actions resulting in the same action values and trajectory.

4.3. A privileged value function improves the gradients

We improve the accuracy of the value function by adding the domain parameters as input to the value function. The parameters are only known when training in simulation, so they can be given as additional information to value function but not to the policy which is used during real life deployment. The privileged value function can predict the future reward more accurately (Figure 4.2), the error is already low but one can see that it reduces the error for the non adversarial case slightly. But halves the error when training with the adversary.

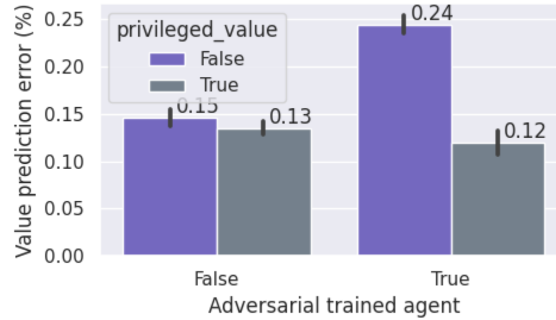


Figure 4.2: Absolute value prediction error computed when using 8 rollout steps, the default in our experiments

Next, we plot Figure 4.1 again but now comparing the effect of the privileged value function and see an slight improvement when a domain parameter was changed (Figure 4.3, left), and similar improvement when attack with the improved agent (Figure 4.3, right).

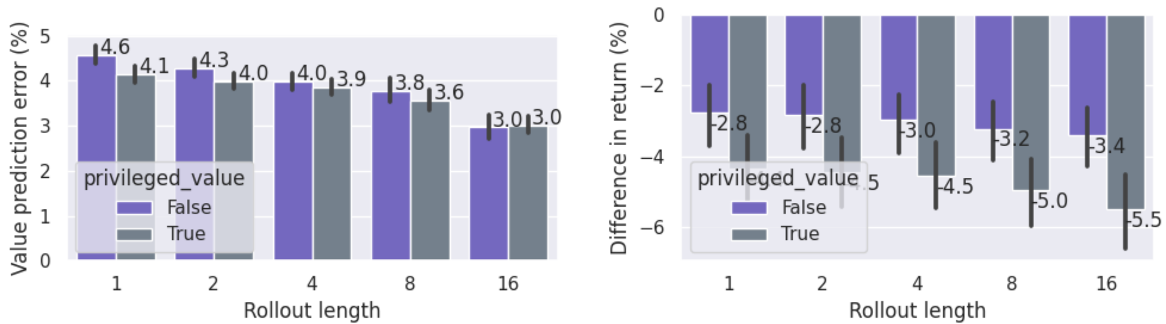


Figure 4.3: Same configuration as Figure 4.1 but now comparing the effect of the privileged value function. Left: The value prediction accuracy is better for the smaller rollout steps. Right: The increase in attack strength of the adversary with a privileged value function is visible for every rollout step setting.

Note that there are hidden effects that we cannot separate: The agent with a privileged value function also has a higher success rate (see Figure 4.11), so its return is generally higher and this leaves more room to reduce the reward by an attack. Figure 4.4 shows this is the case: The agent trained with a privileged value function has a bigger difference in return when changing the domain parameters slightly. This shows that we cannot just state that the gradient attack is actually stronger, since changing the domain parameters in any direction on average is already stronger. However, we can still conclude that a privileged value function is beneficial for the final success rate of the agent (Figure 4.11). And this effect is much stronger when training with the adversary step. One possible explanation is that training with the adversary makes it more difficult for the agent to learn a good policy and value function when only looking at the last 6 states and actions. With the additional information of the current domain parameters the value function has much directer form of feedback. Figure 4.5 shows the much better lower value prediction error when we use a privileged value function when training with the adversary.

Figure 4.4: The difference in return after a domain parameter change. The average is calculated by taking a small step in every direction (similar step size as the adversary takes). Specifically, all vectors obtained from the product of -1, 0, 1 for every parameter (mass and length). This update is done for every trained agent, over 20 different episodes and for every 10th step.

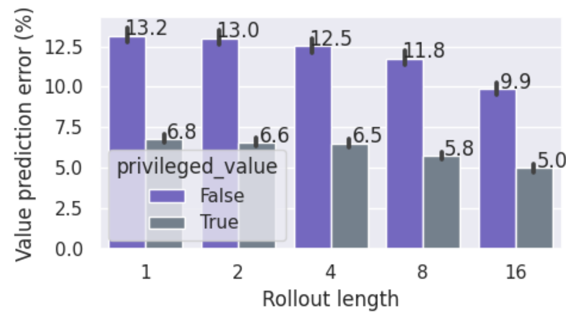


Figure 4.5: A downside of training with the adversary is that it increases the absolute value error. See the non-adversary plot with a values close to 4 in Figure 4.3 (left). Fortunately, this can largely be mitigated when using the privileged value function. We show the value prediction after a small change in domain parameters

4.4. Improved performance when training with the adversary

We have showed the effects of the rollout length and privileged value function to the accuracy of the gradients and the effect when applying them. The final results of our method are shown in this section. We first show the improved resistance to change of domain parameters and finally the improved success rate when training with the adversary and a privileged value function.

4.4.1. Robust against changes in domain parameters

The previous section showed that a privileged value function is more vulnerable to changing domain parameters since it reaches a higher performance. However this result was for an agent that was trained without an adversary. Figure 4.6 shows that when training with an adversary that changes the domain parameters during training, the return difference is much less and actually the privileged value function enhances the resistance to changing domain parameters.

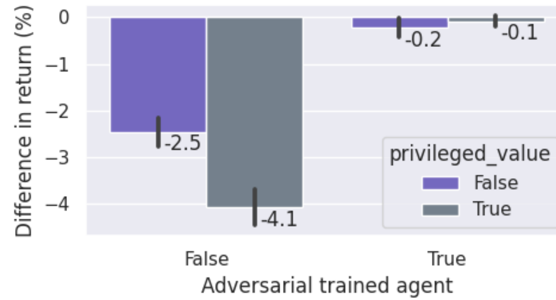


Figure 4.6: The difference in return when a domain parameter changes in any direction during an episode (The same evaluation as done in Figure (4.3, left)). It shows that training with an adversary improves the resistance to changing domain parameters. This effect is enhanced when training with a privileged value function.

In the next section we show that this robustness doesn't come at the cost of performance.

4.4.2. Worst-case guided domain randomization improves robustness

We show that our method, Gradient based adversarial domain randomization, improves the overall performance of all the evaluation distributions and the effect is most positive for the RBF distribution where certain areas have a low chance of being sampled randomly. Note that the baseline is an agent trained with domain randomization and a privileged value function. Our method also those two methods so the difference is only in adding the adversary step.

We evaluate the performance of the agent on a 16x16 grid of domain parameters and show the average success rate over all grid points. We see that the Pendulum is most difficult to control with a high length and mass because of the many swings needed to reach the top. The agent will need to perfect the actions for the full episode. Another difficult region is when the mass is high and the length is small, in this scenario its difficult to swing the pendulum to the top and brake. Braking too late and the pendulum will fall at the other side. With a lower mass the agent can easier compensate for any mistakes.

The choice of distribution is important for the final evaluation score. The gaussian distribution covers only the middle section of the grid and therefore the agent will perform well in the middle but not in the corners. This directly shows the difficulty of manual selecting a good distribution. One could move the distribution more to the difficult top right corner, however this makes it almost impossible for the policy to learn anything since it will never find the good trajectory that reaches zero reward at the top. Similar the RBF distribution is very narrow and mostly samples on two lines: one horizontal and one vertical line at the mean of that axis. The uniform distribution evenly samples the same grid as the evaluation grid, however this might be ineffective since some regions are more difficult than others.

Figure 4.7 shows the difference in performance for the pendulum swing up environment between the baseline and our method when training with the adversary and shows that it improves on average for all three cases: Gaussian, uniform and RBF domain parameter distributions. This average is calculated by testing each agent on a evaluation grid bigger than the training distributions. The heatmap showing the individual success rates per grid point is visualized for the Gaussian distribution in Figure 4.8. The other distributions use the same evaluation grid. It also details the difference for each individual grid point between the baseline method and our method with adversarial training.

The RBF distribution shows the biggest improvement. Difficult domain regions should be sampled more often such that the policy can perfect its actions for those regions. To simulate an environment with difficult regions that are sampled rarely, we change the domain space by artificially creating more parameters using RBF kernels as transformation to the real parameters of the environment. The new distribution space has a couple of spikes towards low and high regions of Mass or Length and the remaining output maps to a similar value. This allows us to show that the method works well and is able to find regions where the agent doesn't perform well yet, while avoiding the complications and time of training a different environment.

Figure 4.7 shows the improvement in performance for the pendulum swing up environment when training on a distribution created by RBF kernels. The adversary is able to improve the performance for

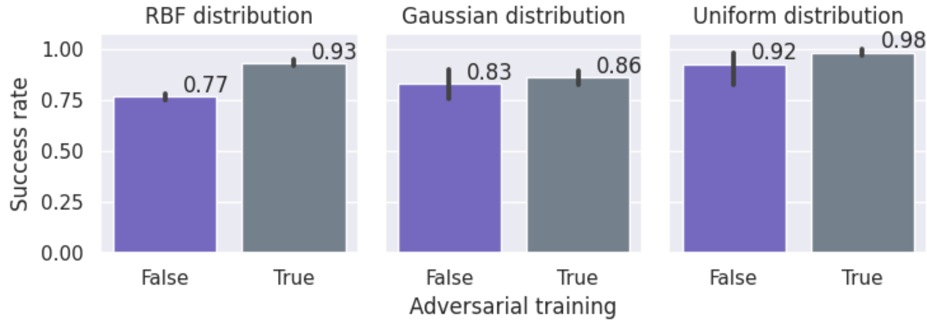


Figure 4.7: Final performance measured in success rate when training with or without the adversary and trained with different parameter distributions. Evaluated on 9 agents trained with different initial seeds, on a 16x16 context grid with each grid point evaluated over 16 initial states.

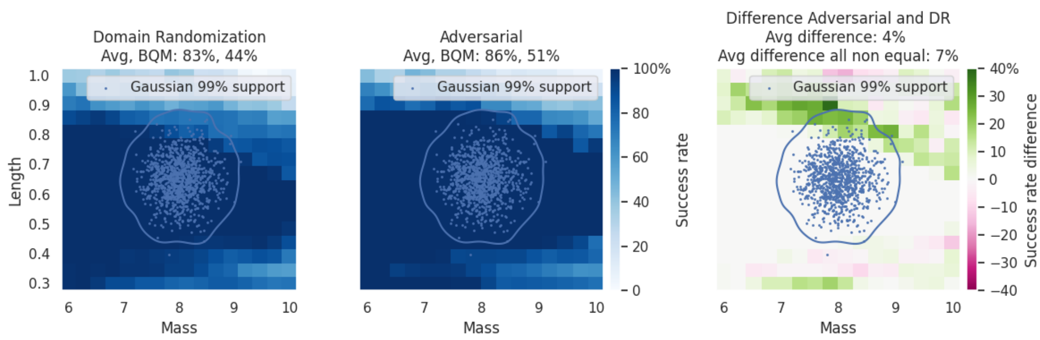


Figure 4.8: Success rate Heatmap for Adversarial and Domain Randomization baseline both trained on a Gaussian distribution with small variance relative to the evaluation grid. Evaluated on 9 agents trained with different initial seeds, on a 16x16 context grid with each grid point evaluated over 16 initial states. BQM: Bottom quartile mean 0-25% (worst-case mean).

the evaluation grid. The heatmap in Figure 4.9 shows the performance difference in detail and Figure 4.10 the success rate over time.

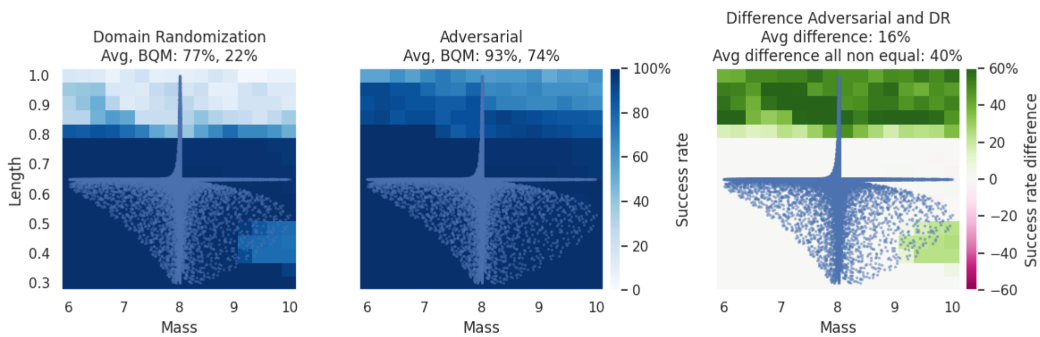


Figure 4.9: Success rate Heatmap for Adversarial and Domain Randomization baseline both trained on a distribution created by RBF kernels. Right: The difference in success rate between the adversarial trained agents and domain randomization agents. Green is a better performing Adversarial agent. Both grids are evaluated on 9 agents trained with different initial seeds, on a 16x16 context grid with each grid point evaluated over 16 initial states.

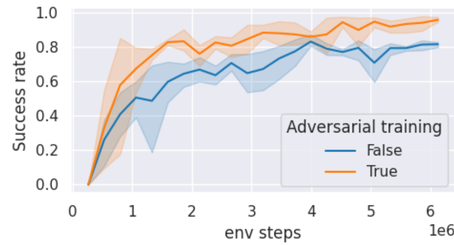


Figure 4.10: RBF distribution evaluation success rate during training. Evaluated on 9 agents trained with different initial seeds, on a 16x16 context grid with each grid point evaluated over 16 initial states

Privileged value function ablation The beneficial effect in success rate when using the privileged value function and the slight gain of the adversary for the gaussian distribution is provided in Figure 4.11. For both cases using a privileged value function improves the performance. The combination of both has the highest average however its not a significant difference due to the high variance.

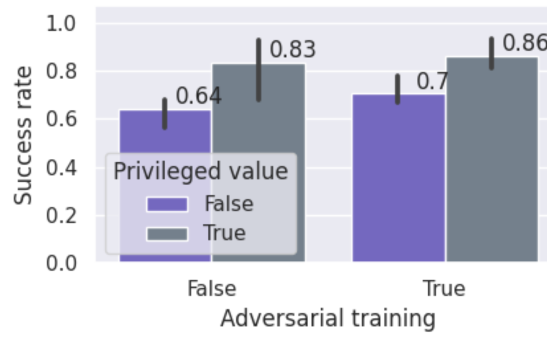


Figure 4.11: Gaussian distribution success rate ablation with privileged value function and adversarial training. Evaluated on 9 agents trained with different initial seeds, on a 16x16 context grid with each grid point evaluated over 16 initial states

4.5. Visualizing the movement of the domain parameters

To show how the domain parameters are moved by the adversary during training we visualize the movement of the domain parameters in Figure 4.12 for the RBF kernel distribution and Figure 4.13 for the Gaussian distribution. Figure 4.13 shows how the domain parameters are bounded since they aren't allowed to move further than the 99% support of the distribution. This is done to ensure we can compare the method fairly when evaluating it on a bigger grid. Training with the adversary provides the most effect during the first part of the training when comparing to the baseline. This is probably because the policy sees the harder parts of the grid earlier and more than regular domain randomization and so learns this faster. The idea that the adversary moves to the bottom part in the first checkpoints even though that should be easier than the top right is that the value function is not converged yet and the policy could be failing in the bottom part. In the RBF case 4.12 at checkpoint 3.2M we see that the adversary moves all domain parameters to the top right, which is to be expected since the agent is very successful in the bottom part of the grid.

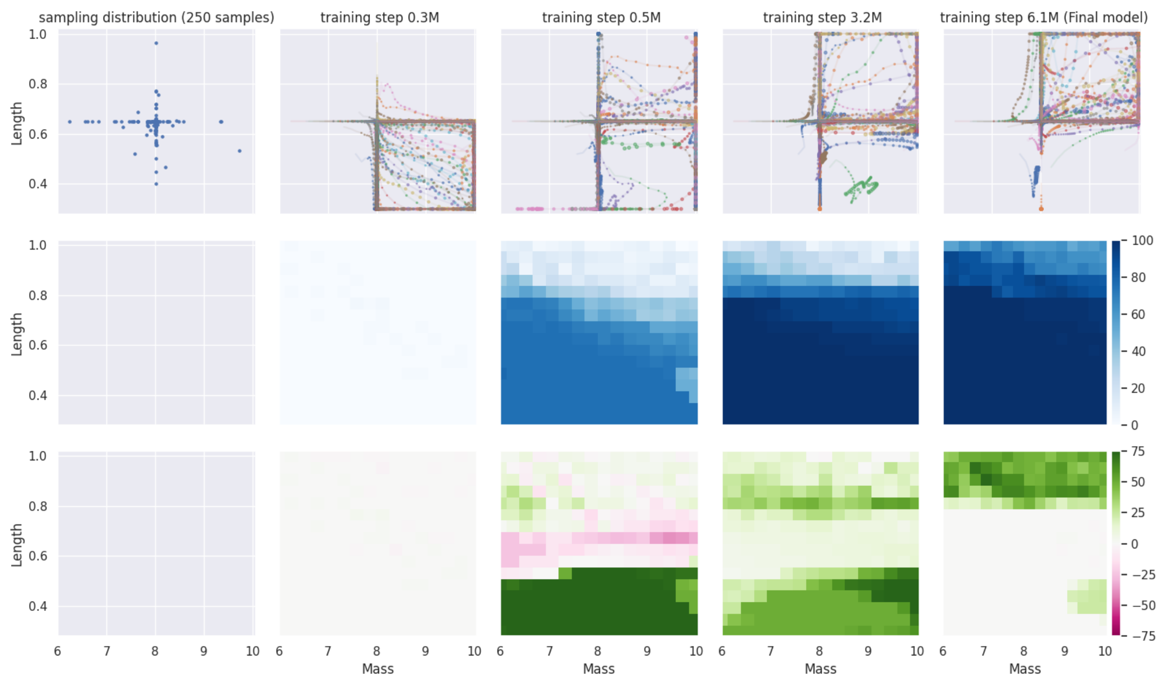


Figure 4.12: Top row: We show the movement of the agent evaluated at different training checkpoints trained with the RBF kernel distribution. For each checkpoint we sample a set of domain parameters and execute the adversary in the same configuration as is done in regular training. We show the movement of these domain parameters running for 4 episodes. The first plot shows the initial samples from the distribution. It's clear that the top right corner is the most difficult and you see that when the value function learns the space the domain parameters also move more towards the top-right corner. Middle row: The success rate of the agent during training with the adversary method where every column shows a different checkpoint in training. Bottom row: The difference in success rate between the baseline and our method. Green means a better performance for the adversary.

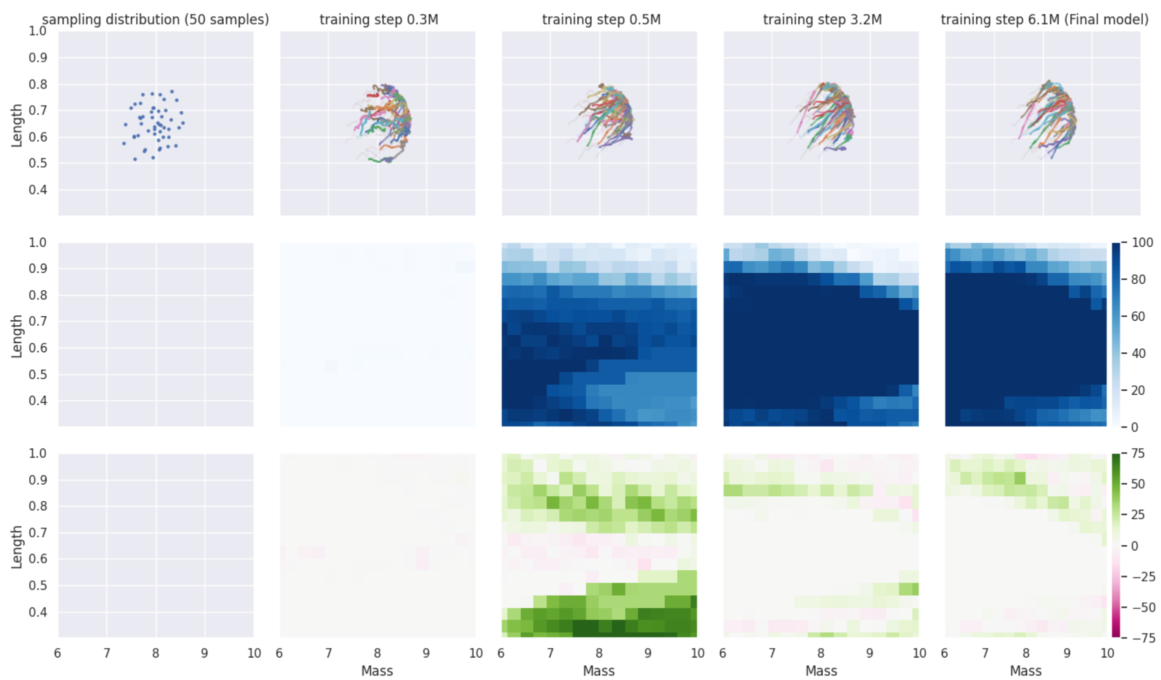


Figure 4.13: Top row: We show the movement of the agent evaluated at different training checkpoints trained with the Gaussian distribution. For each checkpoint we sample a set of domain parameters and execute the adversary in the same configuration as is done in regular training. We show the movement of these domain parameters running for 4 episodes. The first plot shows the initial samples from the distribution. The parameters are bounded to lie within the 99% support of the Gaussian distribution. Middle row: The success rate of the agent during training with the adversary method where every column shows a different checkpoint in training. Bottom row: The difference in success rate between the baseline and our method. Green means a better performance for the adversary. The second checkpoint especially has a big difference in success rate.

5 Conclusion

In conclusion, this thesis presents how changing the domain parameters during training using differential simulator gradients improve the agent's success rate. This adversary domain update happens during the episode and is guided by the gradients of the differential simulator which can accurately find worst-case domain parameters. Training PPO with the adversary using these gradients and a privileged value function results in a robust RL agent that exhibits higher performance in challenging regions while maintaining equivalent performance others. We show that our method improves over regular domain randomization for 3 different distributions: Gaussian, Uniform and a specially created RBF kernel distribution.

We show that the adversary finds the difficult regions since it moves the domain parameters towards the regions with low success rate. We explain in detail the effect of the rollout length for the outcome in attack strength of the adversarial step and show that the addition of a privileged value function improves the accuracy of the value function.

The method comes at a cost: A complex training pipeline, that is difficult to tune and depending on the configuration is much slower than a default Domain Randomization pipeline. Still, it does show an interesting direction for future research to use the value function and a differential simulator in a different ways during training.

Future work in this area should explore integrating recurrent neural networks instead of a history of observations. Applying the approach to the Soft Actor-Critic algorithm instead of PPO might work even better since it saves all the steps with adapted dynamics by the adversary in a replay buffer. Moreover, investigating hybrid methods by combining the differential simulator with a learned dynamics model might make the process more efficient and scalable.

References

[1] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *ArXiv* abs/1707.06347 (2017). URL: https://api.semanticscholar.org/CorpusID:28695052.

[2] OpenAI et al. "Solving Rubik's Cube with a Robot Hand". In: *ArXiv* abs/1910.07113 (2019).

[3] N. Rudin et al. "Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning". In: *ArXiv* abs/2109.11978 (2021).

[4] Ankur Handa et al. "DeXtreme: Transfer of Agile In-hand Manipulation from Simulation to Reality". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)* (2022), pp. 5977–5984. URL: https://api.semanticscholar.org/CorpusID:253107794.

[5] Joonho Lee et al. "Learning quadrupedal locomotion over challenging terrain". In: *Science Robotics* 5 (2020). URL: https://api.semanticscholar.org/CorpusID:224828219.

[6] Fabio Muratore et al. "Robot Learning From Randomized Simulations: A Review". In: *Frontiers in Robotics and AI* 9 (2022).

[7] Bhairav Mehta et al. "Active Domain Randomization". In: *Conference on Robot Learning*. 2019. URL: https://api.semanticscholar.org/CorpusID:104291994.

[8] Lilian Weng. "Domain Randomization for Sim2Real Transfer". In: *lilianweng.github.io* (2019). URL: https://lilianweng.github.io/posts/2019-05-05-domain-randomization.

[9] 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)* (2017), pp. 1–8.

[10] Zhongyu Li et al. "Reinforcement Learning for Robust Parameterized Locomotion Control of Bipedal Robots". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)* (2021), pp. 2811–2817.

[11] Eugene Vinitsky et al. "Robust Reinforcement Learning using Adversarial Populations". In: *ArXiv* abs/2008.01825 (2020).

[12] Lerrel Pinto et al. "Robust Adversarial Reinforcement Learning". In: *International Conference on Machine Learning*. 2017.

[13] Yongyuan Liang et al. "Efficient Adversarial Training without Attacking: Worst-Case-Aware Robust Reinforcement Learning". In: *ArXiv* abs/2210.05927 (2022). URL: https://api.semanticscholar.org/CorpusID:252846499.

[14] Aravind Rajeswaran et al. "EPOpt: Learning Robust Neural Network Policies Using Model Ensembles". In: *ArXiv* abs/1610.01283 (2017).

[15] Pingchuan Ma et al. "RISP: Rendering-Invariant State Predictor with Differentiable Simulation and Rendering for Cross-Domain Parameter Estimation". In: *ArXiv* abs/2205.05678 (2022).

[16] Anay Pattanaik et al. "Robust Deep Reinforcement Learning with Adversarial Attacks". In: *Adaptive Agents and Multi-Agent Systems*. 2018.

[17] Huan Zhang et al. "Robust Deep Reinforcement Learning against Adversarial Perturbations on State Observations". In: *arXiv: Learning* (2020).

[18] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples". In: *CoRR* abs/1412.6572 (2015).

[19] Ajay Mandlekar et al. "Adversarially Robust Policy Learning: Active construction of physically-plausible perturbations". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017), pp. 3932–3939.

[20] Yifeng Jiang et al. "SimGAN: Hybrid Simulator Identification for Domain Adaptation via Adversarial Reinforcement Learning". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)* (2021), pp. 2884–2890.

[21] Marc Rigter, Bruno Lacerda, and Nick Hawes. "RAMBO-RL: Robust Adversarial Model-Based Offline Reinforcement Learning". In: *ArXiv* abs/2204.12581 (2022).

[22] H. J. Terry Suh et al. "Do Differentiable Simulators Give Better Policy Gradients?" In: *International Conference on Machine Learning*. 2022. URL: https://api.semanticscholar.org/CorpusID:246472918.

[23] Marcin Andrychowicz et al. "Learning dexterous in-hand manipulation". In: *The International Journal of Robotics Research* 39 (2018), pp. 20–3. URL: https://api.semanticscholar.org/CorpusID:51894399.

[24] Wenhao Yu, C. Karen Liu, and Greg Turk. "Preparing for the Unknown: Learning a Universal Policy with Online System Identification". In: *ArXiv* abs/1702.02453 (2017). URL: https://api.semanticscholar.org/CorpusID:10615022.

[25] Lerrel Pinto et al. "Asymmetric Actor Critic for Image-Based Robot Learning". In: *ArXiv* abs/1710.06542 (2017). URL: https://api.semanticscholar.org/CorpusID:23865056.

[26] Huan Zhang et al. "Robust Reinforcement Learning on State Observations with Learned Optimal Adversary". In: *ArXiv* abs/2101.08452 (2021).

[27] Michael Lutter et al. "Robust Value Iteration for Continuous Control Tasks". In: *ArXiv* abs/2105.12189 (2021).

[28] C. Daniel Freeman et al. "Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation". In: *ArXiv* abs/2106.13281 (2021).

[29] Keenon Werling et al. "Fast and Feature-Complete Differentiable Physics for Articulated Rigid Bodies with Contact". In: *ArXiv* abs/2103.16021 (2021).

[30] Taylor A. Howell et al. "Dojo: A Differentiable Simulator for Robotics". In: *ArXiv* abs/2203.00806 (2022).

[31] Brian Plancher et al. "GRiD: GPU-Accelerated Rigid Body Dynamics with Analytical Gradients". In: *2022 International Conference on Robotics and Automation (ICRA)* (2021), pp. 6253–6260. URL: https://api.semanticscholar.org/CorpusID:237513745.

[32] Luke Metz et al. "Gradients are Not All You Need". In: *ArXiv* abs/2111.05803 (2021). URL: https://api.semanticscholar.org/CorpusID:243938586.

[33] Jie Xu et al. "Accelerated Policy Learning with Parallel Differentiable Simulation". In: *ArXiv* abs/2204.07137 (2022).

[34] Matthijs T. J. Spaan. "Partially Observable Markov Decision Processes". In: *Reinforcement Learning*. 2012. URL: https://api.semanticscholar.org/CorpusID:15106781.

[35] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *ArXiv* abs/1312.5602 (2013). URL: https://api.semanticscholar.org/CorpusID:15238391.

[36] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning". In: *CoRR* abs/1509.02971 (2015). URL: https://api.semanticscholar.org/CorpusID:16326763.

[37] John J. Hopfield. "Neural networks and physical systems with emergent collective computational abilities." In: *Proceedings of the National Academy of Sciences of the United States of America* 79 8 (1982), pp. 2554–8. URL: https://api.semanticscholar.org/CorpusID:784288.

[38] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533. URL: https://api.semanticscholar.org/CorpusID:205242740.

[39] Ivo Grondman et al. "A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42 (2012), pp. 1291–1307. URL: https://api.semanticscholar.org/CorpusID:1185580.

[40] Thomas M. Moerland et al. "Model-based Reinforcement Learning: A Survey". In: *Foundations and Trends® in Machine Learning* (2020). URL: https://api.semanticscholar.org/CorpusID:220265929.

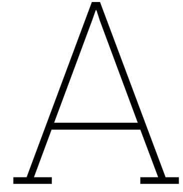
[41] Joshua Achiam. "Spinning Up in Deep Reinforcement Learning". In: (2018).

[42] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). URL: https://api.semanticscholar.org/CorpusID:6628106.

[43] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. "Autograd: Effortless gradients in numpy". In.

[44] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: `http://github.com/google/jax`.

[45] Qiqi Wang. "Forward and adjoint sensitivity computation of chaotic dynamical systems". In: *J. Comput. Phys.* 235 (2012), pp. 1–13. URL: `https://api.semanticscholar.org/CorpusID:11370154`.



Hyperparameters

Table A.1 shows the hyperparameters used for the experiments in this paper. Some details on the parameters:

1. Context resample probability: The probability of resampling the domain parameters at each step of the environment. The value is choosen such that we expect a resample every 4 episodes.
2. RBF distribution $m_{1,2,3}$ and $l_{1,2,3}$ parameters: three parameters that are uniformly sampled between zero and one that are the inputs to the RBF kernel, the function transforms it in a value between 6 and 10 for mass and 0.3 and 1.0 for length.
3. Max grads: The maximum value of the gradients in the adversary update step. The gradients are clipped to this value.

Parameter	Value
Learning rate	3e-4
Parallel envs	32
Batch size	320
Num mini-batch	32
Gradient updates per batch	8
Stacked observations with actions	6
Privileged value function (include domain parameters to observation)	True
Policy network	128 width, 4 layers
Value network	256 width, 5 layers
Discount factor γ	0.98
Domain randomization parameters	
Context resample probability	1/(320*4)
Gaussian distribution	$m \sim \mathcal{N}(8, 0.1), l \sim \mathcal{N}(0.65, 0.005)$
Uniform distribution	$m \sim \mathcal{U}(6, 10), l \sim \mathcal{U}(0.3, 1.0)$
RBF distribution	$m_{1,2,3} \sim \mathcal{U}(0, 1)$, $l_{1,2,3} \sim \mathcal{U}(0, 1)$
Adversary parameters	
Learning rate	0.05
Env steps per adversary update	20
Num of gradient updates	2
Num of prediction steps	5
Rollout length	8
Max grads	5

Table A.1: Hyperparameters used for the experiments in this paper