Reinforcement Learning of Parameters in Complex Physical Systems

Nemo Fournier; Joint work with Tatiana López-Guevara; Supervised by Kartic Subr

May – July 2019

Abstract

In this work we focus on some issues that arise when we want to perform reinforcement learning on robotics setting and that are linked to what is known in the litterature as the *reality gap*. We have indeed come up with an approach that aims to quantify the dependance that the achievement of a task have with respect to some physical parameters of the environment, such as masses, friction, etc. We hope that our method will lead to a better overall understanding of the physical nature of a task, as well as global improvements in the speed of the training process by simplifying the space we have to learn on and we show results that support those hopes.

Contents

1 Introduction and motivations

Robotic engineering has nowaday reached a very advanced level of quality, and the recent robots have at their disposition some of the most precise, strong and reliable control / muscles. At this point, the main limitation to their abilities now comes from a software point of view. Historically, we used to program robots to execute a very precise and well defined task (*e.g.* robots in car factories), or to provide human assistance to those robots while they are performing the task they are assigned to; think for instance to the surgery robots operated by a human surgeon: although they are precise and reliable enough to perform the very act of sugrery, they still need a human operator.

The recent success stories in the field of artificial intelligence, and particularily the one of reinforcement learning have paved the way for autonomous learning of complex tasks, often achieving (sur)human performance in those tasks (most often games such as Go, chess, or even Atari games [MKS+13]), and better performance than agents that were manually programmed ([SHS+18]). There is therefore high hopes that those approaches can be used as well in the field of robotics to more reliably perform complex tasks. I will do introduction to the reinforcement learning field and of some state of the art methods used in the section 2.

The issue that then emerges is that the state of the art reinforcement learning methods are very *sample inefficient*, meaning that to properly learn a task, an agent needs to gather a huge amount of experience (for instance to train the AlphaZero agent, the authors of [SHS+18] needed to play more than 40 million games for chess and more than 140 million for go). Although this can be achieved for games such as chess or go since simulating a playout is essentially costless on a computer, the same thing cannot be said for robotics experiments, that can be time consuming and costly (it is as if we had to restore manually the position of the chessboard at the end of each playout, for millions of them).

Therefore, the current trend is to use physical simulations of the robotics environments, and perform the reinforcement learning of the task on those simulated environments. The major issue now lies in the accuracy of the physical simulations, and more precisely in the discrepancies between the experiment performed in the simulator, and the same experiment performed on a real robot, coming both from the approximations made by the solver of the physical simulator, and the lack of precise knowledge of the real environment. This is known in the litterature as the *reality gap*. I will present in the section 3 and 4 the strategies used to fill in this gap and some of their limitations.

I will then introduce and dwell on an approach that we proposed to enhance and better understand the nature of the reality gap, and especially the dependancy between the efficiency that an agent can achieve while performing the task and the knowledge this agent has of the parameters ruling its environment in the section 5 of this report.

2 Reinforcement Learning

Reinforcement learning (RL) is a field of machine learning exploring how an agent can learn to perform a task through interactions with its environment. It has several contexts of applications, ranging from game theory to control problems in automation, and involves a variety of branches of computer science, among which lie information theory, operational research, deep learning and many others. [SB18]

2.1 First approach

Reinforcement learning aims to mimic the way human beings learn to perform "intuitive" tasks, which is through trial and error interactions with the environment. To describe more precisely the process, an agent observes some state and that describes its current situation in its environment. The agent is then allowed to perform an action that will affect. For instance, in the "gridworld" toy problem that is represented in figure 1, the state of the agent would be its position on the square grid, while the actions it is allowed to take at each state is either move left, up, right or down. At each step the agent will receive a reward associated to its step and action, and will observe its new state. We call the machinery that rules its behaviour (basically mapping a state to an action) its policy.

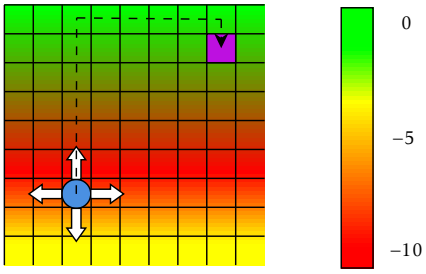

Figure 1: Toy example of a "gridworld" problem: the blue agent wants to reach the purple square while receiving the highest reward possible. An example of optimal trajectory is drawn.

The figure 2 illustrates this intuition of a feedback loop between the agent that acts on the environment and the environment that rules its next step and determines the reward the agent receives and want to maximize.

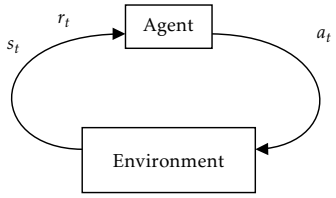

Figure 2: Standard representation of the reinforcement learning feedback loop.

2.2 Some formalism

The general framework for reinforcement learning is the one of Markov Decision Processes.

> **Definition 2.1 (Markov Decision Process)** *A Markov Decision Process (abbreviated MDP) $\mathcal{M}$ is a tuple $(\mathcal{S}, \mathcal{A}, r, \gamma, p, p_0)$ where $\mathcal{S}$ is the set of states, $\mathcal{A}$ the set of actions*

achievable by the agent, $r : \mathcal{S} \times \mathcal{A} \to \mathbf{R}$ is the reward function, γ is a discount factor, $p : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$ is the transition function that maps a couple (state, action) to a ditribution over states and p_0 is the distribution of the initial state.

The *discount factor* is a scalar that can be used to favor reward obtained early in the experiment (typically we multiply the obtained reward by γ^t to obtain the discounted reward, see equation (1) for instance).

To constitute a correct MDP, the transition function must have the Markov property, that is that the future is influenced by the past only through the present:

$$p(s_{t+1} \mid s_0, a_0 \dots s_{t-1}, a_{t-1}, s_t, a_t) = p(s_{t+1} \mid s_t, a_t)$$

Often the observed state space does not carry enough information to make the transition function markovian (because of some hidden scalar that we cannot observe but also determines how the system evolves): in this case we talk of Partially Observable Markov Decision Process. By abuse, we will also refer to them as MDP.

Definition 2.2 (Policy) *A deterministic policy is a function $\mathcal{S} \to \mathcal{A}$ mapping states to actions. A stochastic policy maps a state s to a distribution $\pi(\cdot \mid s)$ over the action space $\mathcal{A}$.*

We will focus on stochastic policies, that are better suited for complex and continuous control over continuous state spaces.

We say that $\tau = (s_0, a_0, s_1, a_1, s_2, \dots, s_a)$ is a trajectory over the MDP $\mathcal{M}$ using a policy π if $s_0 \sim p_0$, and for $t \geq 0$, $a_t \sim \pi(\cdot \mid s_{t-1})$ and $s_{t+1} \sim p(s_t, a_t)$. We denote $\mathrm{T}_{\mathcal{M},\pi}$ the distribution of such trajectories.

The reward being the feedback signal from the environment to the agent, the agent will seek to maximize the (discounted) expected reward over the trajectories, denoted $J(\pi)$, that is defined by

$$J(\pi) = \mathop{\mathbb{E}}_{\tau \sim \mathrm{T}_{\mathcal{M},\pi}} \left[\sum_{t=0}^{H} \gamma^t r(s_t, a_t) \right] \tag{1}$$

Solving the reinforcement learning problem associated to a MDP means finding an optimal policy π^* that can be defined as $\pi^* = \arg\max_{\pi} J(\pi)$.

2.3 Policy Gradient Methods

Depending on the nature of the problem we want to solve, several approaches can be used. Some of them involve exact solving of the MDP, using strategies such as dynamic programming (this typically works for problems such as the gridworld game depicted in figure 1), for others such as one armed bandits problems, UCT (Upper Confidence Bound applied to Trees) can provide good approximate solutions [SB18].

Another approach to solve the reinforcement learning, that is find an optimal policy for a given MDP is to parametrize the policy by a vector θ. For instance if one wants the policy to be a (deep) neural network, the parameter θ could be the vector of weights of the network. Using deep neural networks to represents the policies leads to the recent branch of *deep reinforcement learning*.

The parametrized policy is denoted by π_θ. For the sake of clarity, we also denote $J(\theta)$ for $J(\pi_\theta)$. This leads us to introducing the class of *policy gradient* methods. The idea behind those methods is to perform gradient descent on the parameter of the policy, where $J(\theta)$ is the value we aim to maximize (since it accounts for the reward obtained following the policy π_θ). The idea can be summed up into the following update rule for θ, where α is some learning rate parameter:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla J(\theta_t)$$

Computing the value of $\nabla J(\theta)$ is often intractable because of the expectancy term in the expression of J, especially for environments whose dynamics is not known or complex, so the policy gradients methods rely on estimates of this gradient. Several methods have been introduced from the simple REINFORCE algorithm that relies on Monte Carlo estimation of this gradient by averaging multiple trajectories to more advanced the DDPG algorithm [LHP+15] that exploit an actor-critic architecture, where an additional network is trained to evaluate the choices of the actual policy network. The main difference between each of those methods are in the way they collect data and in the way they define the loss and estimate its gradient. In our implementations we will mostly use the recent PPO algorithm derived in [SWD+17] and that provide state of the art performance despite its relative simplicity. The key idea behind it is to perform policy update whose size is controlled, in terms of impact on the performance of each update. We will anyway treat this algorithm as a black-box since we are not exactly interested in its inner functionning.

3 Robotics

3.1 Reinforcement learning for robotic control

We are interested in applying the techniques that have been developped in the general framework of reinforcement learning and that we have introduced in the previous section in the field of robotics. That is we want to be able to see a robot as an autonomously agent interacting with the environment and learning to perform a given task, guided by a reward signal.

Some example of basic tasks that we are interested could be: moving the end effector of a robotic arm to reach a specific position, walking as steadily as possible for a legged robot, grabbing or pushing an object toward a desired goal. It is easy to describe those problems as reinforcement learning problems, in the MPD framework. For instance, if we want to formalize the task of having a robotic arm pushing an object toward a desired goal:

- A state $s \in \mathcal{S}$ is a tuple containing the configuration of the robotic arm (position, speed of each of its joints) and the position of the object
- An action $a \in \mathcal{A}$ corresponds to a signal that we can apply to the arm (for instance to control the torque of each one of the motors)
- A reward r could be the function that associates the negated distance between the object and the goal

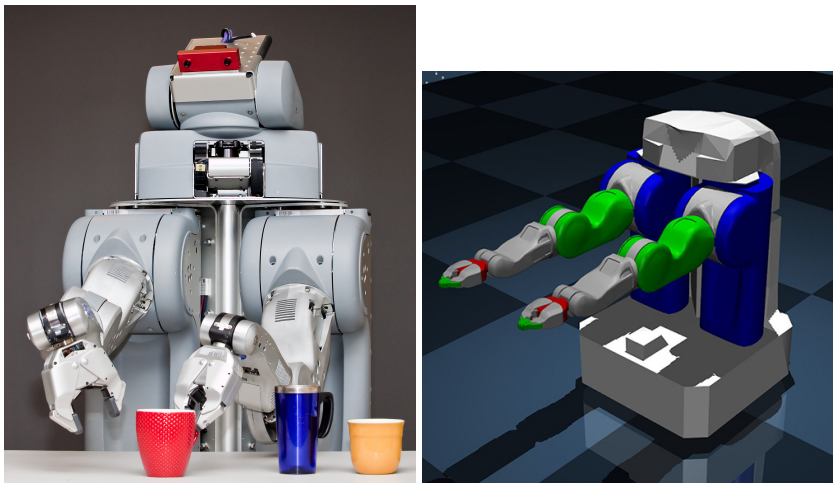

Figure 3: A real PR2 robot, available at the robotics laboratory of the university, and its simulated equivalent, here on the Mujoco [TET12] simulator.

- The transitions functions are imposed by the physical laws constraining the system (gravity, friction, behaviour of the motors, etc)

Therefore an agent maximizing the reward over trajectories would learn to apply the correct control signal to the arm in order to minimize the distance between the object and the goal, *ie.* push the object as close as possible to the goal.

Once we have described a control task as such a reinforcement learning problem, it is tempting to straightforwardly apply the methods that we described earlier when introducing the framework of reinforcement learning. But in the field of robotics, such an idea is often bound to fail, for several reasons. The main one is the low sample efficiency of current reinforcement learning algorithms: the agent needs to explore a lot and perform a lot of tries before being able to perform the task well, which can quickly become expensive and time consuming when a real robot is monopolized to learn a task, as well as a human operator potentially needed to "reset" the environment. Moreover, since the beginning of the learning process is essentially composed of random exploration, this can lead to dangerous behaviour performed by the real robot, either for itself of for its environment.

3.2 Reinforcement Learning using simulations

The current trend to overcome those issues is to perform reinforcement learning in a simulated environment, implementing all the aspects of a MDP associated to a robotic control task.

Some simulators have been developped and have become state of the art tools to model robots, their environment and have an accurate physical simulation to be able to perform reinforcement learning. We can for instance cite MuJoCo [TET12], DART [LGH+18] or NVIDIA FleX, each of them having its strong and weak points, such as the ability to perform fluid simulation, or perform simulations faster than real time. On top

of those simulators people have developped benchmark environments [BCP+16] to evaluate the efficiency of different reinforcement learning methods, as well as environments implementing some models of robot widespread in the different robotic laboratories. For instance figure 3 shows the modelization and implementation of a PR2 robotic controller in the Mujoco simulator.

Although those simulations now allow to apply reinforcement learning methods to robotic control tasks, with often impressive results when evaluated in simulation (see for example the results of the PPO algorithm on some benchmark environments [SWD+17]), directly transfering the learned strategy to the real robot almost always leads to disappointing results, even for simple tasks (see for instance [PAZA18] where a transfered policy for a pushing task fail if the algorithms are applied directly). What is at stake here are the unavoidable discrepancies between the simulator and the real robotic environment, whether they lie in the dynamics of the system, in the configuration of the experiment, because of observational or actional noise, or from a mixture of those factors. Strategies have then come up to overcome this issue known as the *reality gap* in *Simulation to Real (Sim2Real) transfer*.

4 Crossing the reality gap

A whole range of strategies have been deployed to overcome the reality gap. Some of them involve performing the reinforcement learning in two steps, the first and heavier one in simulation, and then refining the strategy by performing *on policy* learning in the real world. The sample efficiency issue is mostly overcome by the heavy learning phase in simulation and the second phase helps to cross the reality gap since the end of the learning is performed under the "true" conditions. Some other approaches choose to use a simulation during the execution in the real world to assess the discrepancies between simulation and the real world at execution, and try to adapt and correct the strategy at running time in order to reduce the discrepancies between what is simulated and what is happening in the real world [ZKBB17].

I will in this section focus on the two methods that inspired the work we did during this internship, that are *dynamics randomization* and learning a *universal policy*.

4.1 Dynamics randomization [PAZA18]

Peng et al. have proposed in [PAZA18] to parametrize the dynamic model of the simulation, that is to use a vector ϕ containing the different scalars ruling the simulation process, and that are easily available in the simulated environment. For example, in the case of a simulated robotic arm, those scalar could be the mass of each part of the robot, the damping of each joint, the different friction coefficients involved in the solid friction simulation, etc. We can formalize this using a modified MDP whose transition dynamic is now given by $p(s_{t+1} \mid s_t, a_t, \phi)$ instead of $p(s_{t+1} \mid s_t, a_t)$. Therefore, given a policy π, the distribution of trajectories will also depend on that vector ϕ of parameters ruling the dynamics of the environment, and we can adapt the notations of the section 2.2 by denoting $\mathrm{T}_{\mathcal{M},\pi,\phi}$ the distribution of trajectories on $\mathcal{M}$ following π under the dynamics given by $s_{t+1} \sim p(s_t, a_t, \phi)$. And follows the definition of the parametrized expected reward of a policy π under dynamics given by ϕ:

$$J_\phi(\pi) = \mathop{\mathbb{E}}_{\tau \sim \mathrm{T}_{\mathcal{M},\pi,\phi}} \sum_{t=0}^{H} \gamma^t r(s_t, a_t)$$

The idea of the authors is to train a policy that maximizes the reward over the trajectories and over the different dynamics. To do so they introduce a distribution ρ over the parameter space and during training, whenever the environment is reset, a new parameter vector is sampled from ρ and will provide a different dynamic to the MDP.

They therefore seek to learn a policy π^* such that:

$$\pi^* = \arg\max_\pi \mathop{\mathbb{E}}_{\phi \sim \rho} \Big[J_\phi(\pi) \Big]$$

Along with some structural adaptations made to the deep policy network, such as a recurrent branch in order to allow the policy to model the dynamics of the environment from observations of the past, the author thus aim to train a policy that will perform well over a wide range of environment's dynamics. They show that one can still standard reinforcement learning algorithm to train such a policy.

The hope is then than when transfering the policy on a real robot, the real environment will just be seen by the agent as a new variation of the environment's dynamics and thus it will perform well. What the article demonstrates is that this approach is actually working, and the results of the transfers are actually improved, for a task such as pushing an object.

4.2 Universal Policy with Online System Identification [YTLT17]

In [YTLT17], Yu et al. approach the issue the same way, using a parametrization of the dynamics of the system. They also learn the policy by sampling the dynamics' parameters from a distribution ρ. The main difference is that this time they choose to make the dependance of the policy with respect to the parameters explicit, by learning what they coin to be a *universal policy* (UP). That is the policy is now a function $\pi(s_t, \phi)$ instead of $\pi(s_t)$. We denote by π_ϕ the instantiation of a universal policy with a given parameter ϕ, ie. $\pi(\cdot, \phi)$.

Using the notations we introduced in 4.1, the authors now want to optimize for π^* such that:

$$\pi^* = \arg\max_\pi \mathop{\mathbb{E}}_{\phi \sim \rho} \Big[J_\phi(\pi_\phi) \Big]$$

That is they seek to find a universal policy π^* that performs well under a wide range of parameters.

The issue is now that the policy has to have information about the environment it is about to perform on in order to perform well. The policy has indeed to have access to the correct vector of parameters of the current dynamics ϕ. Even though that is fine in simulation (and that is the reason why we are able to train such a universal policy), we do not have access to such information during transfer on a real robot.

The author therefore perform a second learning phase in which they train an *online system identification* (OSI) module. That is a function that will take as input the past of

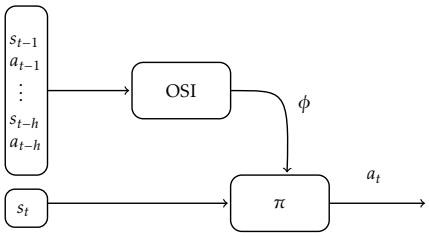

Figure 4: Universal Policy with Online System Identification from [YTLT17] strategy to fill the reality gap

the current trajectories (the last couples of state and action) and try to output the vector ϕ of the parameters that would lead to such an observed dynamics. This function is learned using a classic supervised learning setting: we sample environment dynamics, let the policy perform on the corresponding environment in simulation to generate trajectories, and we learn a function that will try to map the trajectories to a correct parameter vector.

Once again the authors show good results in adaptability of the final policy coupled with its online system identification module.

4.3 Limitation of those approaches

Even though those two approaches lead to encouraging results when it comes to transferring the final policy on a real setting, they come with their drawbacks. In particular both of them involve sampling over the parameter space during training: this leads to an increased sampling complexity of the overall reinforcement learning process. This approach is particularily limited when parameter spaces of high dimensionality are considered. Exploring such spaces to capture enough notions of how the dynamics of the system behave and affect the task to perform well requires too much sampling (an exponential number of samples in the dimension of the parameter space), which adds to the overall high sampling inefficiency of the current reinforcement learning methods. Some further studies such as [MDG+19] have tried to overcome this issue by modifying the sampling process over the parameter space by dynamically trying to pick parameter variations that maximize the information obtained by the agent, while [YTLT17] and [PAZA18] were using a uniform sampling scheme over the parameter space. Although this *active dynamics randomization* has been shown to speed up the learning process in lower dimension, it still struggles to achieve good performance in. During the bibliographical work that I did, the higher dimensionality considered by authors doing dynamics randomization over the parameter space was of dimensionality 25, in [YLT18].

Therefore most of the implementations of those methods rely on hand-crafted subset of parameters that are supposedly relevant for the task we are trying to learn, as well as their "relevant range" in order for the uniform sampling scheme to work.

5 Embedding for Universal Policies and OSI

During this internship, I have tried to enhance the approach of dynamics randomization, and especially the approach chosen by [YTLT17] of training both a universal policy and an online system identification module, in order to adress the issue of high dimensionality parameter spaces. I wanted to be able to identify automatically the relevant dynamics parameters for a given task, and be able to sample from them without carrying the weight of sampling from irrelevant dimensions.

The main intuition behind this approach is that some parameters might be more relevant for a task than other, and thus we should be able to discard them while we randomize without affecting the policy performance. Think for example to a pushing task: moving an object toward a precise point with a robotic arm. One can easily conceive that knowing the friction coefficients of the object we want to push will be more valuable than knowing the color of this same object. We would like to be able to discover this without prior human knowledge of the task and the dynamics of the environment (what the authors did by hand-picking the relevant subset of parameters), and go deeper by trying to quantify the relevance of each parameter with respect to a specific task.

I have chosen to see this problem as a dimensionality reduction problem: we want to be able to go from the whole parameter space Φ, which might be of high dimensionality toward a smaller parameter space, which should contain only the information about the relevant parameters for the current task. I have chosen to learn an embedding, that is a mapping $\Psi : \Phi \to \widetilde{\Phi}$, where $\widetilde{\Phi}$ is the low dimensionality embedded parameter space.

The new problem I formulated is the following: I wanted to learn both an embedded universal policy π and an embedding Ψ maximizing the reward over episodes under varied dynamics.

$$\pi^*, \Psi^* = \arg\max_{\pi,\Psi} \mathop{\mathbb{E}}_{\phi\sim\rho}\left[J_\phi(\pi_{\Psi(\phi)})\right]$$

In other words, I want to learn a universal policy π that is able to perform well when the environments dynamics ϕ are sampled from Φ following the distribution ρ, while not having access to the full parameter vector ϕ but only the embedded parameter vector $\Psi(\phi)$. I thus have to learn at the same time an embedding Ψ that can provide sufficient and relevant enough information for the universal policy to be able to adapt well.

We have therefore tried to investigate how we could effectively learn such an embedding and integrate it into the universal policy and online system identification pipeline proposed by [YTLT17] and seen in figure 4, how we could use it to have a better understanding of the task a robot is learning and how we could mesure the impact it has on learning and on transferability.

I chose to explore the field of autoencoders since they are a well studied and tested dimensionality reduction method.

5.1 Introduction to autoencoders and variational autoencoders

The idea of autoencoders is a fairly simple idea of dimensionality reduction: we aim to tune a neural network in order for it to behave like the identity function on a set of input, while imposing some intermediary representation of smaller dimension. It is a non-supervised learning method.

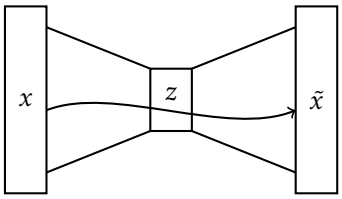

Figure 5: Standard autoencoder modelisation as a feedforward neural network; an intermediary layer of small dimension is used as the intermediate representation z and divides the network into an encoding (at its left) and a decoding (at its right) section. We try to minimize the discrepancy between x and its decoded version $\tilde{x}$ over the distribution of our data.

Although autoencoders by way of some regularization effort can lead to good dimensionality reduction, they do not come as a well controlled generative model. That is, we are not easily able to sample from the embedded space because it has no specific structure or regularity properties, which we still would like to be able to do in our context. The authors of [KW13] have proposed a more sophisticated approach where instead of mapping an input vector x to its intermediate embedded representation, we instead map this vector x to a distribution in the latent space. While training a variational autoencoder, we therefore optimize for two things: we want to be able to have a decent reconstruction behaviour over the initial data distribution, but we also encourage the latent distribution to be close to a target distribution (in the standard case, a multivariate unitary gaussian distribution, whose dimensionality is the dimensionality of the latent space). Doing so we are then able to sample from this target distribution in the latent space, and through the decoder we obtain new data points that hopefully follow the input distribution.

Some authors have derived a generalization of the variational autoencoder framework in [HMP+17] where they introduce the notion of β-variational autoencoder (β-VAE) that aims to provide more insightfulness of the learned latent space by *disentangling* as much as possible the different dimensions of such a space. That is the method we chose to implement as our dimensionality reduction module.

5.2 Autoencoders in the universal policy pipeline

We used the same architecture as what was done for the actual universal policy π in [YTLT17], that is a standard feedforward neural network. The only difference is that it now receives as parameter input the embedded vector $\Psi(\phi)$. The architecture that we are training can be represented as shown on figure 6

To train both the embedded universal policy and the β-VAE, we use the same PPO algorithm as Yu et al. have used, but we perform gradient updates over the whole network coefficients (that is both the policy and the yellow β-VAE network), and we also perform an extra gradient update step over the whole network using the β-VAE loss. The data collection is basically the same as most of the reinforcement learning algorithm, except that we sample a new parameter vector each time (as did the previous papers [YTLT17] and [PAZA18]), and we rollout the policy using the embedded version of this parameter vector. This can be formalized as the algorithm 1.

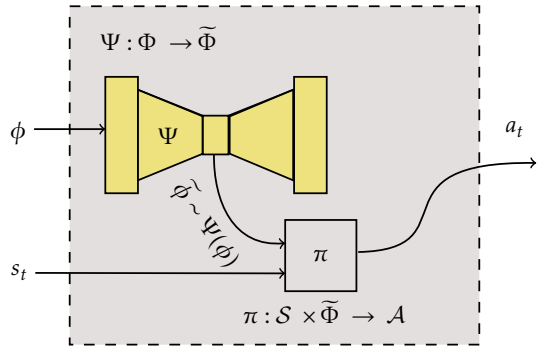

Figure 6: The new architecture we proposed. Here π is based on the work of [YTLT17] and we only restricted its second input to live in $\widetilde{\Phi}$. The autoencoder (the yellow part) here has a standard variational autoencoder structure, and we denote Ψ its encoding part. We were able to set the dimensionality of $\widetilde{\Phi}$ through the actual architecture of the yellow network, i.e. the size of the intermediate layers.

We implemented and used this architecture to learn a standard robotic control task in simulation. We chose the *hopper* environment, consisting of a robotic leg learning to balance and jump as far as possible, and the measure of performance is the distance travelled. We compared the result of our learning process against the method of [YTLT17] that does not use any embedding. We performed our first experiments using a 5-dimensional parameter subspace (we randomized some mass parameters, some friction, power and elastic restitution ones and left the other fixed) to see that our method did not alter deeply the learning process, and was leading to similar learning performance than the baseline.

We then have wanted to make sure that our embedding was effectively capable of filtering out irrelevant dimensions for the task. Since from a human perspective it was fairly hard to manually assess whether a parameter was relevant or not for this *hopper* task, we manually introduce irrelevant parameters that did not affect the dynamics of the system. Those parameters were indeed random numbers sampled at the same time as the other parameters. We ran the learning process again, for a 5 dimensional parameter space (two supposedly relevant – mass and friction – and three artificial and irrelevant random parameters) using a 1 dimensional embedding and we tried to come up with a method to analyze the embedding.

Our embedding is using a β-VAE architecture and therefore associates a distribution over $\mathbf{R}$ to each 5 dimensional parameter vector, a distribution which we can sample from. To try to visualize the embedding of our 5 coefficients (2 being relevant – some mass and friction – and the other 3 being artificial irrelevant parameters), we begin by sampling parameter vectors uniformly from the 5 dimensional parameter space, and then we sample an embedded vector (a real number in our 1 dimensional embedding case) for each of those 5 dimensional parameter vectors using the VAE. We then project this scalar field over all the 2D planes formed by each couple of parameter, using color to represent the value of the sampled embedded vector. All of those projections are represented on figure 7.

Algorithm 1 Embedded Universal Policy

Initialize randomly the UP network π and the β-VAE network
for $i = 1$ to K **do**
  initialize an empty episode buffer
  $t \leftarrow 0$, $\phi \sim \rho$, $s_0 \sim p_0$
  while the size of the episode buffer is lower than M **do**
    $\widetilde{\phi} \sim \Psi(\phi)$
    $a_t \leftarrow \pi(s_t, \widetilde{\phi})$
    $r_t \leftarrow r(s_t, a_t)$
    $s_{t+1} \sim p(s_t, a_t, \phi)$
    $t \leftarrow t + 1$
    if s_{t+1} is terminating **then**
      $\tau \leftarrow (s_0, a_0, \ldots, s_t, a_t)$
      add (τ, ϕ) to the replay buffer
      $t \leftarrow 0$, $\phi \sim \rho$, $s_0 \sim p_0$
    end if
  end while
  Jointly update π and the β-VAE using the PPO and the β-VAE losses using the episode buffer as the data
end for

What we observe is that the embedding tries to encode some structured information for the first 2 dimensions – the relevant ones: mass and friction – while it does not seem to carry information along the 3 others. We can either see this through the histograms on the diagonal of the figure 7, representing the distribution of the mass of the embedded vectors when when we marginalize over a single parameter. Histograms have indeed a non uniform structure for the mass and the friction while they are pretty uniform for the 3 others. A uniform structure means that the weight of the embedded values are distributed uniformly across the whole range of the parameter, i.e. that regardless of the value of the parameter, the embedded vector will be spread across all its range when we sample from it, which basically means that no relevant information is conveyed by its value. The projections show what looks like random noise for the irrelevant - irrelevant couples (right down corner) We tried to formalize this idea that marginalizing over the irrelevant parameters lead to a uniformity in the distribution of the values of the embedding since no information is encoded for this irrelevant parameter.

More formally, for the d-th parameter (i.e. the dimension d out of the D dimensions of the parameter space), we define the distribution $\mathbf{P}_d$ of weights of the embedding as follows:

$$\mathbf{P}_d(x) \propto \mathop{\mathbb{E}}_{\substack{\phi \sim \mathcal{U}([0,1]^D) \\ \widetilde{\phi} \sim \Psi(\phi|_{d=x})}} \left[\left| \widetilde{\phi} \right| \right]$$

Where $\phi|_{d=x}$ means that every dimension of ϕ has been sampled uniformly from their normalized range, except for the d-th whose value is set equal to x. We were able to sample from those distributions, and we measured their distance to the uniform distribution. For our experiments, we chose to use the Wasserstein distance between distri-

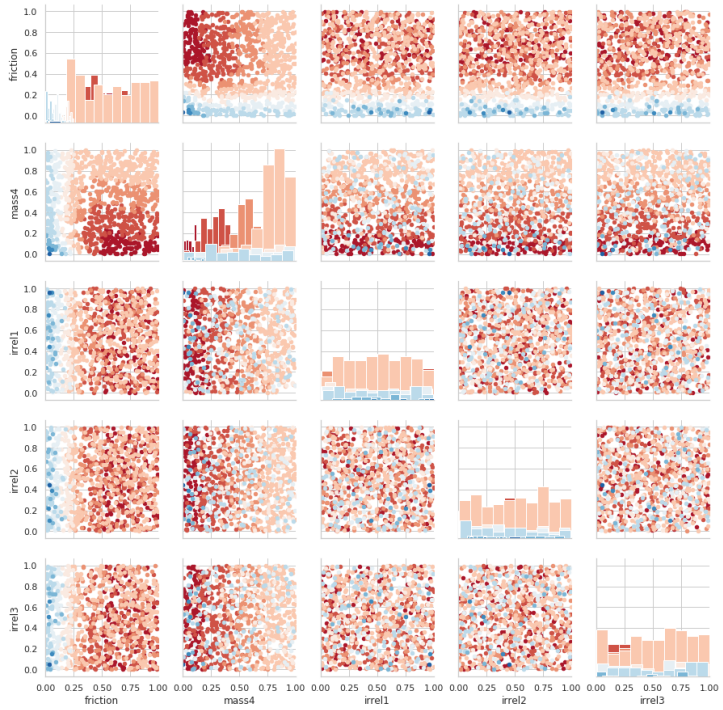

Figure 7: Projection over all the 2D planes formed by the couples of parameters of the embedding of samples of the 5 dimensional space over. On the diagonal we find the histogram of values of the embedding marginalized over a single dimension.

butions, and obtained for each dimension what we called a *relevance metric*. The results of this metric for our toy problem with irrelevant dimensions can be seen in figure 8. We can see that our process is effectively able to filter out the irrelevant dimension and to quantify the relevance of parameters.

We adapted our process for an higher dimensional embedding (a 3 dimensional one in our experiments), by treating each dimension of the embedding independantly. On figure 9 we can compare the relevance results obtained using either a 1 dimensional embedding or a 3 dimensional embedding. We can observe that we achieve consistant result regardless of the dimensionality of the embedding used, but that using an higher dimensional one can provide more subtle relevance analysis. We also encounter a behaviour of the β-VAE already described in [HMP+17] that is that some dimension of the embedding are totaly unused (here visible through the small blue circles at the center of the relevance disk in figure 9).

5.3 Embedding in the UPOSI pipeline

We have then tried to take advantage of the lower dimensionality of the embedded space and its structure that allows to sample from it and then reconstruct back to a high dimensional parameter vector using the decoder for the second phase of the process that

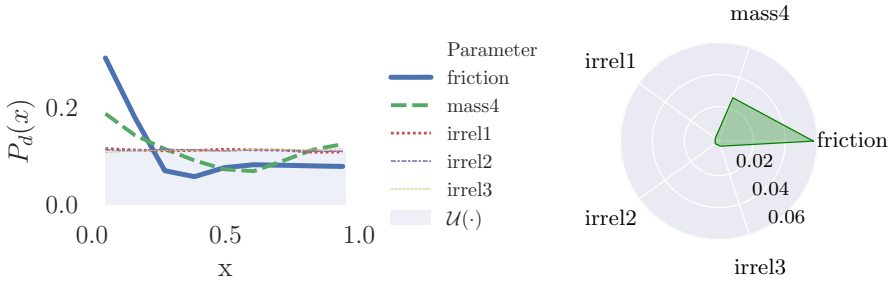

Figure 8: Toy problem on the Hopper environment with 2 real parameters (friction, mass) + 3 manually added irrelevant parameters (setting the parameters has no effect on the environment). (Left) Distribution $\mathbf{P}_d$ computed for each of the $5d$ dimensions. (Right) Relevance metric computed for all dimensions.

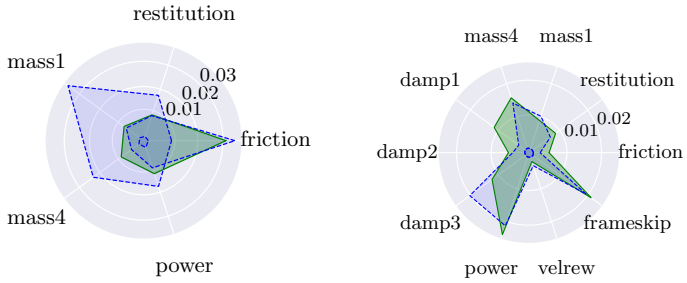

Figure 9: Relevance metric computed for using the embedding for a $5d$ parameter space (left) and a 10d parameter space (right) on the Hopper environment. Blue dotted line corresponds to a 3 dimensional embedding, each of its dimension analyzed separately and green solid line corresponds to the 1 dimensional one.

[YTLT17] introduced. To be able to be deployed on a real robot, our universal policy needs an online system identification module to be trained in order to provide the universal policy with the right information about the dynamics of the current environment.

Since our modified process now uses a policy that receives an embedded parameter vector, we trained an OSI module predicting embedded vectors instead of actual vectors. If we recall how such an OSI system is trained, it is just a supervised learning procedure in which we learn to map trajectories to their correct dynamics vector. In [YTLT17] they learn this mapping by sampling the whole parameter space, simulating trajectories using the corresponding dynamics and optimizing a regresser. We adapted the procedure to sample only from the latent space, and we used the decoder part of the β-VAE to recover a parameter vector that allows for the simulation of the dynamics to collect the corresponding trajectory, while our regresser was optimizing to predict the embedded vector from the trajectory.

The result of this procedure is a full embedded UP-OSI pipeline as seen in the figure 10, similar to the original one (figure 4), except that the output of the OSI module is now

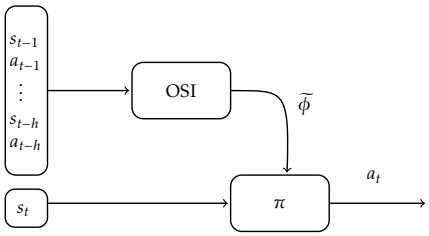

Figure 10: Embedded Universal Policy with Embedded Online System Identification

an embedded $\widetilde{\phi}$ instead of ϕ.

We have then wanted to evaluate such a modification of the pipeline proposed by Yu et al. We therefore trained some universal policies and online system identification modules for parameter space of different dimension on the hopper environment. We trained a standard UP-OSI pipeline (i.e. following the exact procedure from [YTLT17]) as well as a 1 dimensional embedded pipeline and a 3 dimensional embedded pipeline (that is whose embedded vectors are of dimension 1 and 3) using the new procedure that we described. We compared the three approaches on the same test setting consisting of real rollouts with dynamics sampled uniformly on the whole parameter space. We compared them in term of OSI error (that is: how well is our OSI module able to predict the correct (embedded) parameter vector) and in term of average reward obtained by the (embedded) universal policy fed by its system identification module. The results are visible in figure 11. We include several checkpoints that corresponds to checkpoints saved during the training of the OSI module that assess the "speed" at which the OSI module learns.

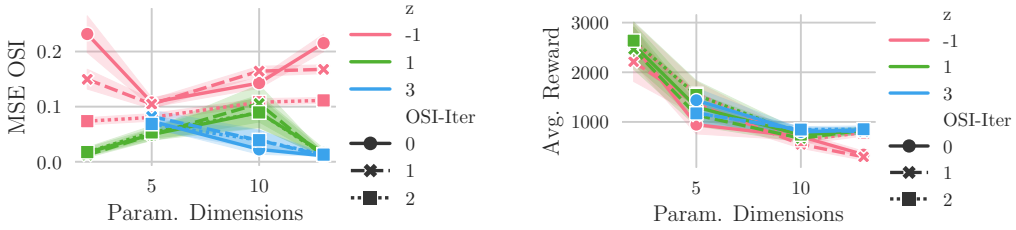

Figure 11: Effect of the embedding in terms of (Left) OSI prediction error and (Right) Average reward on the task both using Dart-Sim. $z = -1$ corresponds to the baseline without any embedding, $z = 1$ to the 1 dimensional embedding and $z = 3$ to the 3 dimensional one.

We observe that our process leads to lower OSI error as the dimensionality of the parameter space increases as well as better average reward. The OSI error being lower with our approach makes sense, since our embedding is learned in a way such that dimensions that does not affect the dynamics of the system are discarded. And, if we think the other way around, such parameters that do not really affect the dynamics of the system will be hard to estimate from trajectories ruled by this very dynamic (once

again, think about trying to predict the color of a ball only from its trajectory): in the non embedded OSI such parameter predictions will lead to high error while prediction in the embedded space will not be affected by those errors.

We also observe that our embedded OSI module training converges faster than the non embedded one (there is a high variability between checkpoints for $z = -1$ on the left plot). The impact of this can be seen on the right plot, where for high dimension, we have to wait until the last checkpoint to achieve similar performance on the task for the non embedded universal policy, while we achieve very good result from the firsts checkpoints on the embedded ones. This can be explained by the fact that to train the embedded OSI, we have to sample either a 1 dimensional or a 3 dimensional space, while for the higher dimensions, we have to sample a 13 dimensional space in order to train the conventionnal OSI module, which obviously is more sample ineficient.

5.4 Transferability evaluation

We then have wanted to assess the transferability (as in Sim2Real transfer) of our strategy. We have spent some time trying to set up an experiment with real robots, but some technical difficulties to set them up as well as the low availability of the robots of the laboratory during the weeks of main robotic conferences deadlines forbid us to do some serious evaluation. We chose another approach, still relevant, that is Sim2Sim transfer, meaning that we trained an agent on a simulated environment using a specific simulator software, and then we implemented the same environment on another simulator software and deployed the policy learned on the first simulator on the second one without any training on the second one. The discrepancies between the two simulators (because of different design choices in how the inner mechanics are implemented) and between the two environments are sufficient to provide some clues as to how our strategy would transfer in a real setting.

The simulator we were using until now was DART [LGH+18]. Since we had already trained a lot of policies for the hopper environment, we chose to stick with it and implemented its counterpart in the Mujoco simulator [TET12]. We then deployed the UP-OSI pipeline we trained on the DART hopper environment directly on the Mujoco hopper environment, and performed the same evaluation as the one we performed in the previous section.

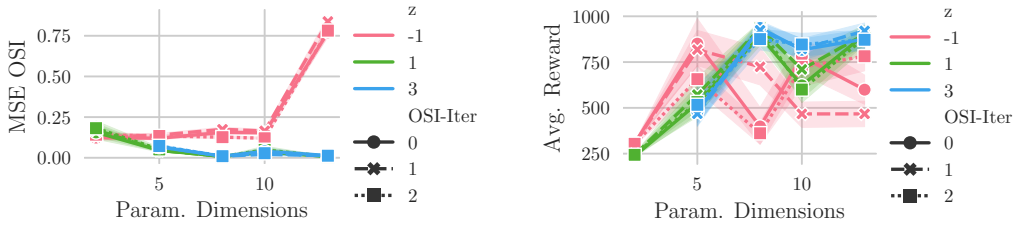

Figure 12: Transferring a Hopper UP-OSI agent trained on a Dart Simulator to Mujoco. We are interested in evaluating the effect of the embedding in terms of (Left) OSI prediction error and (Right) Average reward on the task.

Obviously, both methods achieve lower average reward once the policies were transfered to the Mujoco environment. But still, once again, we can observe that the impact of our embedding approach is positive, both in term of OSI error (especially for high dimensional parameter space) and in term of average reward (and especially in term of convergence speed of our method).

6 Conclusion

We have therefore come up with an approach that is able to provide good insight about how a robotic task that we are learning is sensitive to some of the parameters ruling the dynamics of the environment. At the same time, we have developped a way to sample from a low dimensionality subspace that contains the relevant information and that we have shown to speed up the process of training a system identification module.

Although I think that a more solid evaluation of the method is still to be done, our first results show that this direction might be worth pursuing. During the internship we have tried to submit a paper to the CoRL conference, but its requierements in term of real robotics experiment ended up monopolizing a huge amount of time only to set up the real robots, time which we would have better spent doing a more variety of simulated experiments and *Sim2Sim* transfers.

We have hope that this method can lead to rich results, in terms of speed of the overall reinforcement learning process, and also in term of parameter estimation. Indeed, now that we are able to quantify how a parameter is relevant to achieve a precise task, using some Bayes machinery we can see the other way around and quantify how well is a certain task able to provide an estimate of a parameter. We could explore how to learn OSI specific policies (i.e. trying to find a policy that can lead to very good estimates of the environment parameter), and then perform the task using the task specific policy using our more precise estimate of the parameters relevant for this task.

I would like to thank Kartic Subr and Tatiana López-Guevara for their time and the valuable help and advices they provided me with, it was a real pleasure to work along with them, as well as with the people I shared my office and my lunches with.

References

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[HMP+17] Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. *ICLR*, 2(5):6, 2017.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[LGH+18] Jeongseok Lee, Michael X. Grey, Sehoon Ha, Tobias Kunz, Sumit Jain, Yuting Ye, Siddhartha S. Srinivasa, Mike Stilman, and C. Karen Liu. DART: Dynamic animation and robotics toolkit. *The Journal of Open Source Software*, 3(22):500, Feb 2018.

[LHP+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[MDG+19] Bhairav Mehta, Manfred Diaz, Florian Golemo, Christopher Joseph Pal, and Liam Paull. Active domain randomization. *ArXiv*, abs/1904.04762, 2019.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[PAZA18] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2018.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[TET12] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[YLT18] Wenwei Yu, Chuanjian Liu, and Greg Turk. Policy transfer with strategy optimization. *ArXiv*, abs/1810.05751, 2018.

[YTLT17] Wenhao Yu, Jie Tan, C Karen Liu, and Greg Turk. Preparing for the unknown: Learning a universal policy with online system identification. *arXiv preprint arXiv:1702.02453*, 2017.

[ZKBB17] Shaojun Zhu, Andrew Kimmel, Kostas E. Bekris, and Abdeslam Boularias. Fast model identification via physics engines for data-efficient policy search. In *IJCAI*, 2017.