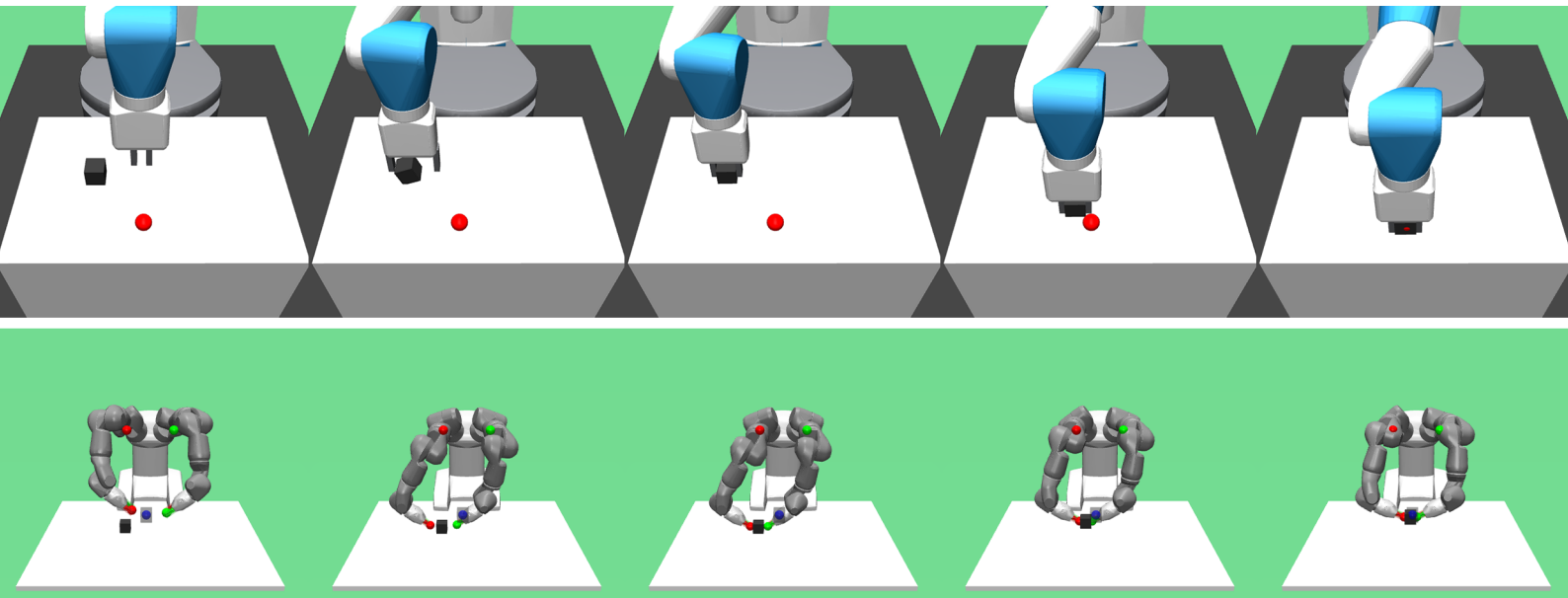
DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Reinforcement Learning for Dexterity Transfer Between Manipulators

CARLO RAPISARDA



KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Reinforcement Learning for Dexterity Transfer Between Manipulators

CARLO RAPISARDA

Master in Computer Science
Date: June 17, 2019
Supervisor: Rika Antonova
Examiner: Danica Kragić Jensfelt
School of Electrical Engineering and Computer Science
Swedish title: Reinforcement Learning för överföring av fingerfärdighet mellan manipulatorer

Abstract

Learning complex manipulation skills with robotic arms is a challenging problem in Reinforcement Learning. Training policies from scratch is often time-consuming and normally infeasible when using real robots. Existing techniques solve this issue by leveraging human priors in the form of shaped rewards or demonstrations, but most ignore the potential of using other robots as demonstrators. In this work, we attempt to transfer complex manipulation skills between robots with different morphologies, so that knowledge that has already been acquired can be leveraged to achieve new tasks. We use a recently proposed method to learn a shared feature space between states of the different robots, and then exploit this representation to transfer skills. For this purpose, we introduce a new technique we call Translated Behavior Cloning, which combines Transfer Learning methods with model-free RL. The results show that transferring manipulation skills between a single arm manipulator and a dual-arm robot, despite the large differences between them, is indeed possible, although further research is needed to make the training process more data efficient for real systems.

Sammanfattning

Att lära komplexa manipuleringsfärdigheter för robotarmar är ett utmanande problem inom Reinforcement Learning. Att träna en policy från grunden på en riktig robot är ofta tidskrävande och vanligtvis orimligt. Existerande tekniker löser detta genom mänskliga förkunskaper som utformade belöningar eller demonstrationer, men de flesta ignorerar potentialen i att använda andra robotar som demonstratörer. I detta arbete försöker vi överföra komplexa fingerfärdighetsförmågor mellan robotar med olika morfologier, så att kunskapen som redan har förskaffats kan användas för att uppnå nya uppgifter. Vi använder en nyligen föreslagen metod för att lära en delad attributrymd (feature space) mellan tillstånd av de olika robotarna, och utnyttjar denna representation för att överföra förmågor. För detta syfte introducerar vi en ny teknik som vi kallar Translated Behavior Cloning, som kombinerar lärandeöverföring (transfer learning) med model-free Reinforcement Learning. Resultaten visar att överföring av manipuleringsförmågor mellan en enarmad och tvåarmad robot, trots stora skillnader mellan dem, är möjlig även om ytterligare forskning behövs för att göra träningsprocessen mer data-effektiv för verkliga system.

Acknowledgements

This thesis was developed at the division of Robotics, Perception, and Learning of EECS at KTH; I would like to thank the students and researchers working there, who shared the lab's computing resources with me and granted me access to software licenses.

I would like to thank my supervisor Rika Antonova, for dedicating her time and effort sharing precious advice and knowledge with me, and Prof. Danica Kragic, for providing her support and evaluating this thesis.

I would like to express my gratitude to my friends and fellow students for supporting me, including Martin Hwasser, who helped me translate segments of this report to Swedish.

Most importantly, I would like to thank my family for enabling me to study at KTH, and for being there for me at all times.

Contents

Chapter 1

Introduction

Robotics is perhaps one of the most powerful realizations of automation, although robots capable of accomplishing tasks within our households are still few. This is because it is very difficult to hand-engineer software capable of dealing with unpredictability. The answer to this problem is almost definitely Machine Learning, but applying it to Robotics comes with additional challenges. For example, the input to our learned controller is often composed by partial and noisy information, while the output is a complex time-dependent set of signals. In addition, allowing the robot to learn by trial and error may not only be very inefficient, but also potentially dangerous for the environment and for the machine itself. Due to such limitations, training robots to perform tasks in unpredictable environments is currently expensive and time-consuming.

A different approach is instead to train the robot from demonstrations. As humans, we are used to the idea of learning new physical tasks by observing others while they perform them, and even some animals appear to do the same. Recent work explores the possibility of using human demonstrations to train manipulators, but such techniques require time-consuming data collection. An alternative approach could be for the robot to learn from another machine.

1.1 Research Questions

This thesis aims to address the following research questions:

- Is it possible to transfer complex object manipulation skills between simulated robots with different morphologies? What Transfer Learning methods could be applied to this problem?
- We focus on a technique that combines Transfer Learning with Reinforcement Learning to perform skill transfer. We address the question of how its sample efficiency compares to methods that learn from scratch.

1.2 Overview

We approach our research questions by first analyzing existing techniques, highlighting their strengths and weaknesses, and by then presenting our own solution for this particular problem. We introduce a new set of simulated environments powered by the MuJoCo physics engine, and developed specifically for skill transfer. We then propose a novel technique we call *Translated Behavior Cloning*, which aims to train a robotic agent by observing successful demonstrations performed by another robot, leveraging an existing method based on autoencoders to translate between the two.

We proceed by experimenting with robotic agents learning object manipulation skills from scratch, using state-of-the-art Reinforcement Learning methods for continuous control, such as DDPG and PPO. This first set of experiments serves to validate our implementations of the relevant RL algorithms, as well as to motivate the need for more efficient methods.

In a second set of experiments, we validate our novel technique on several simulated environments, and successfully transfer object manipulation skills between robots with different morphologies. In particular, we are able to exploit an already solved task – Pick and Place – to transfer the skills needed to achieve the Push task from a single-arm manipulator to a dual-arm robot. We report results on the effectiveness of the presented method, as well as data on its sample efficiency when compared to learning from scratch. Finally, we discuss the results and comment on the importance of temporal alignment between two robots solving a task when collecting training data.

To our knowledge, the proposed method is the first that aims to transfer complex skills between high DoF manipulators with non-trivial differences in morphology, actuation, and dynamics. Our hope is that these experiments, together with the developed simulated environments will provide a starting base for further research regarding this challenging problem.

1.3 Contributions

The main contributions of this work can be summarized in the following points:

- We introduce a new set of simulated environments designed to test and develop methods for transferring skills between robotic manipulators. The environments use the standard interface of OpenAI's Gym package, they are open-source and available on GitHub[1].
- We propose a novel technique we call *Translated Behavior Cloning* that aims to train a robotic agent by observing successful demonstrations performed by another robot, leveraging an existing method to translate between the two.
- We validate our methods with experiments on the simulated environments, while presenting an analysis on the importance of trajectory alignment during training.

1.4 Ethics, Societal Aspects and Sustainability

When working with highly transformative technologies in the areas of Artificial Intelligence and Robotics, it is very important to consider the potential impact of one's research project to others and to the environment. In the context of this thesis, one of the issues connected to robotic agents is their impact on the job market. This is something that we have been witnessing for decades now, but the more recent advances in the world of autonomous systems are bound to affect exponentially more individuals in the close future. Methods like the ones presented here may play a role in this process. When talking about AI, one of the ethical aspects that is often mentioned and that

[1] `http://github.com/carlo-/gym`

should be taken into account is represented by Autonomous Weapons Systems. This work doesn't directly contribute to the development of such systems, but it's important to remember that many of the technologies used for military purposes are based on academic research. Finally, we consider possible sustainability aspects. Many are not aware or ignore that training large Machine Learning models may potentially use copious amounts of energy[2]. In our particular case, this aspect is less of an issue, as computers used during the project are mostly powered by electricity from renewable energy sources.

[2] https://www.technologyreview.com/s/613630/training-a-single-ai-model-can-emit-as-much-carbon-as-five-cars-in-their-lifetimes/

Chapter 2

Background

This chapter serves as an introduction to the theory behind some of the key concepts on which this thesis is built. In particular, we report the fundamentals of Reinforcement Learning, the basics of Artificial Neural Networks, and some more specific topics such as Autoencoders. For a more rigorous set of definitions, theory, and proofs, the reader should refer to the excellent literature from Sutton and Barto [1] for Reinforcement Learning, and from Goodfellow, Bengio, and Courville [2] for Deep Learning concepts.

2.1 Reinforcement Learning

Unless otherwise stated, formal definitions throughout this section are taken from Sutton and Barto [1].

The basic components of every Reinforcement Learning setting are 1) an agent taking actions, 2) an environment with which the agent can interact, and 3) a feedback signal (known as reward) from the latter to the former that indicates whether the action taken was good or not. These pieces are equivalent to the controller, plant, and sensor measurements of the typical setting in Control Theory. Reinforcement Learning methods can be quite powerful, because the agent doesn't need to have any prior knowledge about the world it lives in, and can instead learn to solve its task by trial and error, i.e. simply by exploring the possible actions and assimilating the feedback it receives.

2.1.1 Markov Decision Processes

When working with Reinforcement Learning, it is often convenient to formulate the specific problem we are trying to solve as a *Markov Decision Process* (MDP), as its assumptions and formal structure allow us to apply several methods with strong theoretical guarantees.

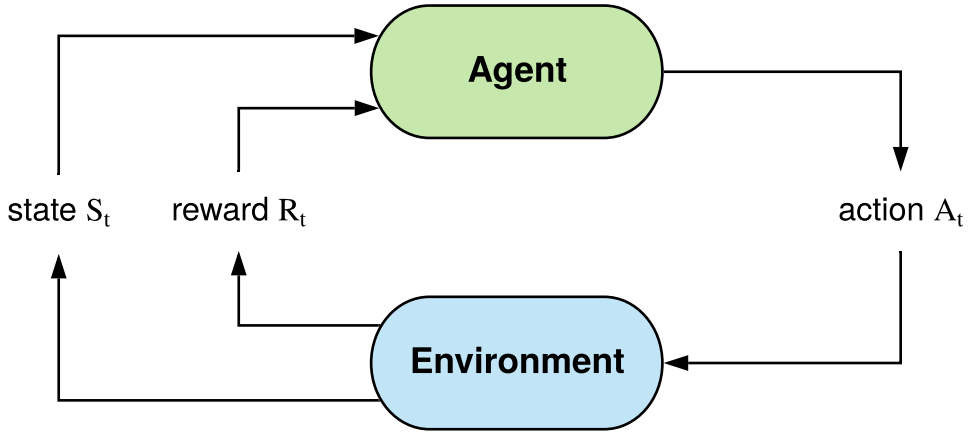


Figure 2.1: Illustration of an MDP.

Let us indicate S_t as the state of the environment at time t, A_t as the action taken by the agent after observing S_t, and R_t as the reward received after taking A_t, leading to a new state S_{t+1}. At each step, the agent observes the state, takes one action, and receives some reward, resulting in a sequence[1] of tuples of the form (S_t, A_t, R_t). By definition, the environment must satisfy the *Markov property*, meaning that each state must depend only on the previous state and action, and not on the entire history. In addition, in case of a *finite* MDP, the set $\mathcal{S}$ of possible states and the set $\mathcal{A}$ of possible actions are finite. Formally, the dynamics of the system can be defined as:

$$p(s', r|s, a) \doteq \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.1}$$

where $s', s \in \mathcal{S}, a \in \mathcal{A}$. Note how the probability of observing a particular state and reward only depends on the single preceding state and action pair. In the notation above, time is discrete and can be defined arbitrarily depending on the problem we are trying to solve. We define MDPs that have a fixed number T of steps (interactions with the environment) as *finite horizon* MDPs.

[1] Such a sequence is often also called *trajectory* or *episode*. The latter term is particularly used when each sequence is fully independent from the others.

Similarly, we define MDPs where $T \to \infty$ as *infinite horizon* MDPs.

The objective of the agent is not as simple as maximizing the immediate reward at each step, but rather to reach long-term goals that will ultimately yield the highest rewards. In other words, we wish to maximize the *expected return*:

$$G_t \doteq \sum_{k=t+1}^{T} R_k \tag{2.2}$$

In the case of infinite horizon MDPs ($T \to \infty$), the sum in (2.2) may diverge, therefore we instead define the expected *discounted* return as:

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.3}$$

where $\gamma \in [0, 1]$ is referred to as *discount factor*. The role of γ is indeed to guarantee that G_t is always a finite quantity, but also to regulate of "far" we want our agent to look into the future.

Given the above definitions, we can now introduce the concept of *policy* $\pi(a|s)$ as the probability of choosing action a when at state s, and of *state-value function*[2] under the policy π as:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t | S_t = s\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right], \ \forall s \in \mathcal{S} \tag{2.4}$$

which outputs the expected discounted return when starting from s and following the policy π. It is also useful to define the *action-value function*[3] under the policy π as:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a\right] \tag{2.5}$$

which represents the value of being at state s and taking action a. Since our goal is to maximize G_t, it is easy to realize that both $v_\pi(s)$ and $q_\pi(s, a)$ are very good tools to optimize for the return.

Although we omit the proof, it can be shown that for any MDP, there exists a policy that is better than all other policies. We call such a policy the *optimal*

[2]The state-value function is sometimes simply called *value function*.

[3]The action-value function is sometimes simply called *Q function*.

policy[4] and denote it as π_*. We can also define its corresponding state/action-value functions as:

$$v_*(s) \doteq \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S} \tag{2.6}$$

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \tag{2.7}$$

If we know the action-value function under an arbitrary policy π, we can easily obtain a better policy as:

$$\begin{aligned} \pi'(s) &\doteq \operatorname*{argmax}_a q_\pi(s,a) \\ &= \operatorname*{argmax}_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] \end{aligned} \tag{2.8}$$

with π' guaranteed to be better than π, unless already optimal (proof in [1]). Eq (2.8) is sometimes referred to as *Policy Improvement*.

2.1.2 Policy and Value Iteration

As we have seen in the previous sections, the quantities in (2.4) and (2.5) could be very useful to maximize G_t and therefore find an optimal policy, although neither of them is normally available. One way to obtain these functions is through *Dynamic Programming* (DP) methods, which can be used to find exact solutions analytically in polynomial time (in $|\mathcal{S}|$ and $|\mathcal{A}|$).

If an exact solution is not needed, estimates can be obtained with iterative algorithms such as *Iterative Policy Evaluation*, useful to estimate the function $V(s) \approx v_\pi(s)$ down to a fixed error threshold. In fact, Policy Evaluation can be easily combined with (2.8), into what is called *Policy Iteration*, allowing us to estimate $\pi \approx \pi_*$.

Alternatively, we could use *Value Iteration*, a variant of Policy Evaluation that involves fewer operations, and therefore tends to converge faster. In this case, the key idea is to run the Policy Evaluation step only for a single "pass" through $\mathcal{S}$, without waiting for the estimate error to reach a certain value. Because of this, we can rewrite the algorithm with a single update rule.

[4] Note that although we refer to the optimal policy as a singular function, there may exist several optimal policies with the same state/action-value functions.

An important aspect of the methods presented in this section is that we never directly estimate the action-value function, but rather rely on our knowledge about the model (for example, note how we exploit $p(s', r|s, a)$ in (2.8)). This may not be an issue for a simple problem, but for a lot of real-world scenarios, this assumption cannot be fulfilled.

2.1.3 Monte Carlo Methods

If the dynamics of our environment are not known a priori, we can try to estimate the state-value function from observations; one way to achieve this is to use *Monte Carlo* methods. The main idea consists in simply collecting experience by interacting with the environment while averaging the returns observed at each step.

Algorithm 1: First-visit Monte Carlo

Input : A policy π

$V(\cdot)$ arbitrary
Returns$(\cdot) \leftarrow$ empty list
while *not converged* **do**
    Generate trajectory of length T with π
    $G \leftarrow 0$
    for $t = T-1, T-2, ..., 0$ **do**
        $G \leftarrow \gamma G + R_{t+1}$
        if S_t *not yet seen in current trajectory* **then**
            Append G to *Returns*(S_t)
            $V(S_t) \leftarrow$ average(*Returns*(S_t))

As we collect more experience, the estimate eventually converges to the true state-value function. A particular version of this approach, *First-visit Monte Carlo*, is shown above (as in [1]).

2.1.4 Temporal-Difference Learning

Similarly to Monte Carlo methods, *Temporal-Difference Learning* (TD) allows us to estimate the state-value function directly from experience. Unlike MC though, algorithms based on TD learning update V at each step rather than waiting for the end of each trajectory, while still being able to converge to the true v_π (although this is only guaranteed if certain conditions on the update

step-size are met; refer to [1] for a detailed discussion on this matter). Because the estimate is updated more frequently, TD methods tend to converge faster than Monte Carlo methods.

Algorithm 2: TD(0) for estimating v_π

Input : A policy π
Parameter : The step-size $\alpha \in (0, 1]$

$V(\cdot)$ arbitrary
for *each episode* **do**
  $S \leftarrow S_0$
  for *each step of the episode* **do**
    $A \leftarrow \text{argmax}_a\, \pi(S)$
    $R, S' \leftarrow$ *Environment*(A)
    $V(S) \leftarrow V(S) + \alpha\,[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$

Temporal-Difference Learning is at the core of many fundamental algorithms in RL, such as *SARSA* [3] and *Q-learning* [4].

2.1.5 Exploration versus Exploitation

One of the core problems in the area of Reinforcement Learning is the balance between the amount of *Exploration* that the agent performs and the amount of *Exploitation* of what has been learned. Intuitively, in order to discover new parts or characteristics of the environment, the agent has to try new actions and observe new states. At the same time, such new states may actually correspond to undesirable or dangerous situations (with low or negative reward), therefore, in some cases, it may be convenient to use the already acquired knowledge and perform only the actions that are known to be good. If the agent is learning in a simulated environment, we may encourage it to explore as much as possible to quickly obtain an accurate model of the dynamics; on the other hand, if the agent is a real robot, we usually want to limit exploration to prioritize the safety of the machine and its surroundings. Most RL algorithms include parameters that can be tuned to regulate this balance. For examples of more advanced methods for safe exploration, García and Fernández [5] provide a detailed survey on this topic.

2.2 Artificial Neural Networks

2.2.1 Basic concepts and software packages

Artificial Neural Networks are widely used to learn approximations of arbitrarily complex functions by simply observing data. They are built connecting a number of elementary units called *perceptrons*, each of which takes one input, multiplies it with a scalar value (usually called *weight*), and returns the result of this operation as output. Whenever a single unit receives more than one input, we multiply each value with a different weight, and output the sum as the result. We can visually assemble these units in parallel (vertically, forming a *layer*) and in series (horizontally, where the output from one unit becomes the input to the next one); when we have multiple layers, we talk about *Multi-Layer Perceptron* (MLP) networks. In general, we can think of a Neural Network as a graph, where each edge between two nodes is associated with a weight.

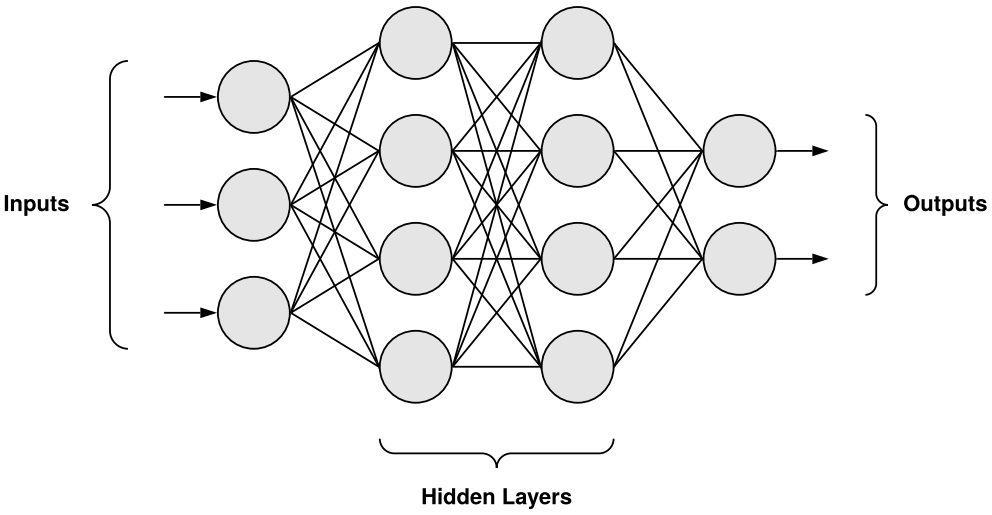


Figure 2.2: Illustration of a fully connected feed-forward neural network.

If all units are connected with all others in the neighbouring layers, we say that the network is *fully connected*. Layers between the input and output units are often called *hidden*; if the neural network has at least one hidden layer, we say the network is a *deep* neural network. Mathematically, we can think of each layer of a neural network as a function $f(\mathbf{x})$ that takes a vector as input and

returns a vector as output:

$$f(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{2.9}$$

where $\mathbf{b}$ controls the *bias* and $\mathbf{W}$ is a matrix containing all weights between the layer and the previous one, allowing us to approximate any linear function. In order to approximate non-linearities, we can wrap f with a non-linear function g, often called *activation function*.

Training is usually performed with *Backpropagation*, a method that consists in keeping track of gradients each time we feed data to the network, and then moving back, propagating the errors while adjusting the weights. There are many software packages that allow users to build and train Neural Networks with relative ease, all of which perform automatic differentiation, have convenient Python bindings, and can leverage graphics cards to efficiently parallelize matrix operations. The most widely used packages are TensorFlow [6] and PyTorch [7], the latter being the preferred library for this work.

2.2.2 Autoencoders

Autoencoders are a particular type of neural networks that are very useful when we wish to learn a compact representation of our dataset. The structure of such networks can be divided into two parts: the *encoder* – which has the role of compressing the information down to a small number of dimensions – and the *decoder* – which instead takes the compressed representation and tries to reconstruct the original input as well as possible. The compressed representation is often called *latent* representation, and the set of its possible values spans a *latent space*.

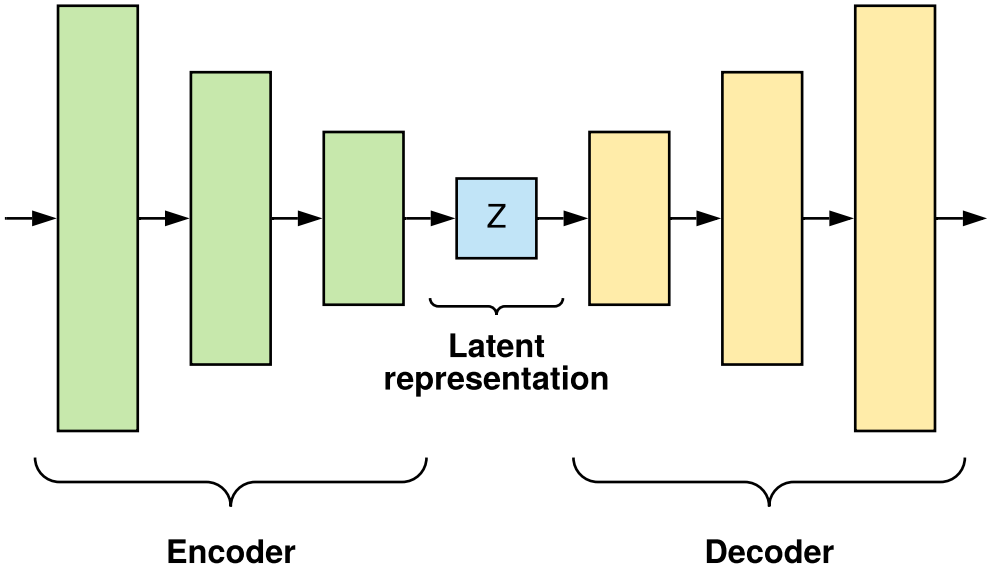


Figure 2.3: Basic structure of an autoencoder

The two parts are often symmetric, with the number of hidden units first decreasing in the encoder and then increasing back in the decoder. This structure makes the central part of the network a bottleneck for the information going through, encouraging the compression. The network is trained end-to-end, and the reconstructed output resulting from the decoder is then compared with the original input, meaning that autoencoders can be trained in an unsupervised manner. The reconstruction loss also encourages the network to learn a latent representation that automatically includes only the most important information about the input [2].

Recent work built on top of vanilla autoencoders has enabled very powerful applications. For instance, Variational Autoencoders (VAE) [8] make strong assumptions on the probabilistic properties of the latent space, and have been successfully employed to compress data such as images and text, and to generate new synthetic samples ([9] [10]).

2.3 RL with Neural Networks

The power of most modern Reinforcement Learning methods comes from the idea of using Deep Neural Networks as function approximators to represent the policy and the Q-function of the learner. Such methods have enabled re-

searchers to solve challenging tasks that were previously considered out of the realm of what computers can do; because of this, several open-source libraries that implement various Deep RL algorithms have been developed, such as OpenAI's Baselines [11], and Berkeley's RLLib [12]. In this section, we report the details of the RL algorithms used in our experiments, following the notation from [13].

2.3.1 Proximal Policy Optimization (PPO)

First introduced by Schulman et al. [14], Proximal Policy Optimization (PPO) is an on-policy RL algorithm capable of dealing with both continuous and discrete action spaces. Several variants exist; we describe the simpler *PPO-clip*. The algorithm alternates between collecting new observations and improving the policy, while approximating the value function as well. The objective function used to update a PPO policy is the following:

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right) \tag{2.10}$$

where θ_k are the parameters of the old policy, g is defined as:

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases} \tag{2.11}$$

and A is the advantage function[5]. The idea behind PPO-clip is that the $\min$ operator limits the impact of each update, so that the policy improves in a stable manner.

2.3.2 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al. [15]) is an off-policy Reinforcement Learning algorithm for continuous control. It builds on Q-learning using Deep Neural Networks as function approximators, avoiding the limitations of tabular methods. The algorithm uses two separate networks for predicting the best action given a state (the *actor*, μ), and for estimating the Q-function (the *critic*, Q); both networks are trained with SGD, and are updated periodically with target networks that lag behind. Since DDPG is an

[5]Given a policy π, its advantage function is defined as $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$

off-policy method, it also requires a replay buffer that stores the agent's experience and that can be sampled.

The basic idea of the algorithm is the following: we begin by collecting some trajectories using a random policy, and we store this experience into the replay buffer $\mathcal{D}$; once we have enough data, we sample a batch of transitions $B = \{(s, a, r, s')\} \subset \mathcal{D}$ and compute the Q-function target:

$$y(r, s') = r + \gamma Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s')) \tag{2.12}$$

We then update the Q-function network with:

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s') \in B} (Q_\phi(s, a) - y(r, s'))^2 \tag{2.13}$$

and the deterministic policy with:

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s)) \tag{2.14}$$

The target networks are instead updated with Polyak Averaging [16] to stabilize the training:

$$\begin{aligned} \phi_{\text{targ}} &\leftarrow \rho \phi_{\text{targ}} + (1 + \rho)\phi \\ \theta_{\text{targ}} &\leftarrow \rho \theta_{\text{targ}} + (1 + \rho)\theta \end{aligned} \tag{2.15}$$

where ρ is a hyperparameter between 0 and 1.

2.3.3 Hindsight Experience Replay (HER)

First introduced by Andrychowicz et al. [17], Hindsight Experience Replay is a method that allows RL agents to learn from mistakes as well as from successful episodes. It can be used with any off-policy algorithm such as DDPG [15], DQN [18], and SAC [19], and its implementation is rather simple. The basic idea is to store trajectories in the replay buffer and "pretend" that the achieved goal at each episode was actually the desired one. Having access to the reward function of the environment, when a trajectory is sampled for training we recompute the rewards with a modified goal, effectively learning from successful episodes that were never actually experienced by the agent. This simple idea has been shown to work well even when exploration is very difficult and the rewards are sparse [20] [21] [22].

Chapter 3

Related Work

Although our specific research questions have never been addressed before, it is possible to divide them into several sub-problems, some of which have already clear answers, while others are still very much a matter of study. In this chapter, we will review existing work relevant to the problem subject of this thesis, highlighting strengths and weaknesses of some of the methods, while connecting it to our specific research questions.

3.1 Learning Robotic Manipulation

Until not long ago, the problem of learning complex robotic manipulation tasks with Reinforcement Learning was considered intractable. In the past two decades, developments in terms of hardware, algorithms, and human effort have resulted in some success. For instance, Levine, Wagener, and Abbeel [23] use Guided Policy Search to learn complex tasks such as screwing caps on bottles without the use of any models. Gu et al. [24] instead learn deep Q-functions by parallelizing the policy updates across a pool of multiple robots, efficiently solving tasks such as door opening. Popov et al. [25] have managed to solve the pick and place task using DDPG [15], but their solution involves a complex shaped reward function and the gripper used is limited to 3 DoF. Rajeswaran et al. [26] instead train a 24 DoF hand with a moving forearm to accomplish a series of complex manipulation tasks, including pick and place, tool use, and in-hand manipulation; this is done with a proposed algorithm called Demonstration Augmented Policy Gradient that consists in first training from human demonstrations with Behavior Cloning, and then fine-tuning with a novel augmented loss (also based on demonstrations). Another impressive step forward was made by OpenAI et al. [27], who managed to train a

control policy for a simulated Shadow Dexterous Hand (also 24 DoF) to perform precise in-hand manipulation of objects without any demonstrations; in this case, the obtained policy was robust enough to transfer to a physical robot successfully. Another very interesting research direction is learning with vision systems, for instance, Levine et al. [28] use a convolutional neural network to predict the quality of a possible grasp, and then exploit this system together with a hand-engineered controller to solve the pick and place task on single arm manipulators.

3.2 Transfer Learning in Robotics

Humans are able to learn new tasks quickly by taking advantage of previous knowledge, therefore it is reasonable to desire the same behavior when training intelligent systems. Because of this, Transfer Learning (TL) is a well-researched area of Machine Learning with the goal of exploiting knowledge acquired in the past to solve a new task. When it comes to TL applied to Deep Learning, an important development in this direction is represented by the Model-Agnostic Meta-Learning paradigm (MAML) from Finn, Abbeel, and Levine [29]; the method is able to pre-train a neural network model on multiple tasks simultaneously, so that a high-quality, specialized model can be obtained after few fine-tuning iterations on a specific task. MAML has been successfully used to solve RL problems, but can only be applied if the type information that the agent can observe is fixed across the different tasks.

When applying RL to robotics, it is common (and usually necessary) to encode information about the robot state as observations (for example, the agent should be informed about its position and the current configuration of its joints, either from a vision system or from other kinds of sensors), therefore, different robots with different morphologies will necessarily observe a different world, making MAML not applicable for this problem. Devin et al. [30] try to overcome this issue by splitting RL policies into "task-specific" and "robot-specific" neural networks that ideally can be recombined to solve different tasks with different robots. Perhaps the most relevant work for this project is the approach from Gupta et al. [31], who were able to transfer policies between two robots with different morphologies and actuator types by learning a common feature space between the state observations of the two. Unlike existing research, the presented project aims to transfer policies between realistic, high DoF manipulators, such as an anthropomorphic hand to a two-arm robot equipped with parallel grippers.

3.3 Learning from Demonstrations

Programming precise instructions for a robot to complete a task can be extremely tedious and in some cases even infeasible, therefore it would be valuable to be able to demonstrate how to accomplish a certain task directly, and have the machine simply imitate the shown solution. Ekvall and Kragic [32] present a planning approach that exploits human instructions to adapt the constrains of the problem and solve object manipulation tasks. Konidaris et al. [33] introduces an algorithm to split demonstrations into different skill branches, and then use the resulting "skill tree" to control a mobile manipulator from expert trajectories. More recently, Rozo, Jiménez, and Torras [34] proposed a method that exploits human demonstrations with a robotic arm (via teleoperation) combined with force perception to learn tasks such as pouring a liquid into a glass. Other very exciting directions include the possibility of robot learning from visual demonstrations; for example, Yang et al. [35] employ two convolutional neural networks to extract task information from videos of human demonstrators, and then control a dual-arm manipulator to reproduce similar tasks.

More recent work focuses on policies represented by neural networks, and take advantage of behavior cloning (BC) to imitate trajectories from a demonstrator in a supervised manner. For example, Nair et al. [36] record human demonstrations with Virtual Reality equipment, and then load these trajectories into the replay buffer of an off-policy RL algorithm, considerably improving the sample efficiency of a manipulator learning the pick and place task. Similarly, Rajeswaran et al. [26] use a CyberGlove III to record precise joint angles of a human hand as demonstrations, and then exploit BC to learn complex manipulation tasks with a simulated anthropomorphic hand. For a much more detailed overview, the survey from Argall et al. [37] provides an extensive summary on robot learning from demonstration, although some of the referenced methods are now outdated.

Chapter 4

Methods

4.1 Overview

In this work, we aim to transfer object manipulation skills between robots with different morphologies, using a novel method we call *Translated Behavior Cloning*. The overall procedure can be explained with three main points:

1. We start with two agents (agent_a and agent_b) that are able to successfully solve a task – called *Proxy Task* – in two different environments (respectively env_a and env_b) controlling two different robots (a *student* and a *teacher*). The agents can be either pre-trained, hand-engineered, or human demonstrators.

2. We proceed by generating a dataset of *twin* trajectories with both agents executing the proxy task, making sure that – at each episode – the goals of the two agents are aligned, and the starting and ending states are equivalent. We then use this dataset to learn a shared feature space with the method described in [31].

3. Finally, we teach the second robot to perform a second task by having the first robot demonstrate a successful policy, and by exploiting the learned feature space to translate between the two.

Details on the second point will be given in Sec. 4.2, while the third point – our main contribution – will be described in Sec. 4.3 later on.

To accomplish our goal, we start from the work of Gupta et al. [31], who use an architecture consisting of two autoencoders to learn a shared feature

space between states of two different robots. Once the feature space has been obtained, they use it as a heuristic to guide the exploration when learning a new task, introducing a new component into the reward signal. Although the method seems promising, the authors only report results with robots that move in a 2D world and have a very limited number of DoF (3 to 4). In addition, the morphologies of the two robots used in the transfer only differ slightly, for example by a single joint. Instead, our goal is to transfer manipulation skills between complex robots that move in 3D space and have a large number of DoF. Their morphologies differ considerably, going from one arm to two arms, or from anthropomorphic fingers to parallel grippers, meaning that the optimal strategies to solve the same task on two different robots may require very different sequences of actions.

4.2 Learning a shared feature space

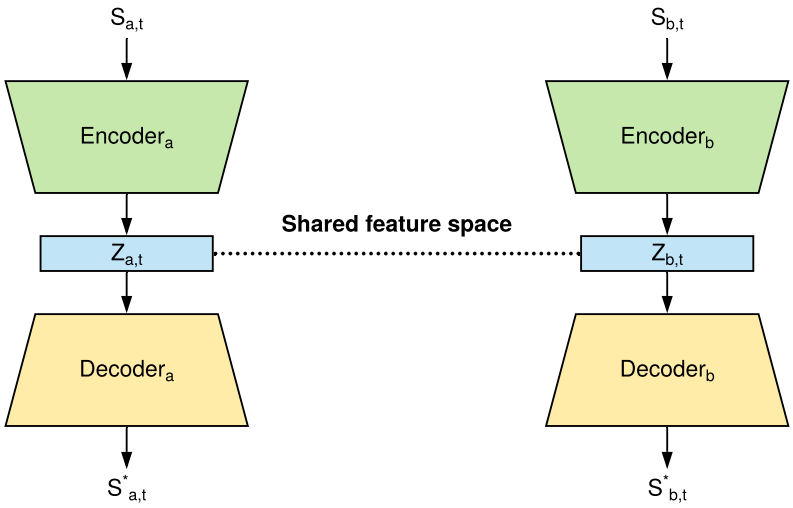


Figure 4.1: The two autoencoders trained to learn a shared feature space between states of two different robots. Note that the networks have two distinct latent spaces, but we show them as connected because we encourage them to be as similar as possible.

Given two controllers capable of solving a proxy task, we begin by collecting a dataset of *twin* trajectories of the two robots. An important consideration is that the initial and final states must be equivalent. The dataset is then fed into two identical autoencoders (one for each robot) that learn to compress and

reconstruct the input (see Figure 4.1). The two networks are trained together, with a loss that encourages the latent spaces to be as similar as possible at each step. In practice, the goal is to learn a shared feature space between states of the two robots, so that at a later time we are able to tell how similar is the current state of a robot to that of the other robot. Ideally, this method abstracts away the details of the robots' morphologies (that is, the number of joints, lengths, physical space they occupy), but also the types of their actuators, and their different dynamics, extracting a description that encodes the different stages of the task regardless of who or which machine is performing it.

4.2.1 Training

Training is performed end-to-end with Stochastic Gradient Descent [38], in particular, we use the Adam optimizer [39]. As in [31], both networks are trained at the same time, with a MSE loss function that encourages both a good reconstruction of the outputs with respect to the inputs,

$$\mathcal{L}_a = ||S_a - S_a^*||_2 \qquad \mathcal{L}_b = ||S_b - S_b^*||_2 \tag{4.1}$$

and a similarity loss [40] that forces the two latent spaces to be as alike as possible for aligned input states:

$$\mathcal{L}_{sim} = ||Z_a - Z_b||_2 \tag{4.2}$$

The total loss fed to the optimizer is a weighted sum of the three:

$$\mathcal{L}_{tot} = \lambda_a \cdot \mathcal{L}_a + \lambda_{sim} \cdot \mathcal{L}_{sim} + \lambda_b \cdot \mathcal{L}_b \tag{4.3}$$

where λ_a, λ_b, and λ_{sim} are hyperparameters. These must be chosen carefully, so that none of the loss contributions prevails over the others.

4.2.2 Alignment of the states

For the learned feature space to be meaningful, it is important that during training the inputs to the two autoencoders are semantically equivalent, i.e. the states should represent their respective robot at the same stage of the task execution. We have already assumed that this is always true at initialization and termination of each episode, but we can't say the same for any of the intermediate steps a priori. In some cases, temporal alignment may be enough, but if the robots execute the task with different strategies or at different paces, the temporal alignment is lost. To solve this issue, Gupta et al. [31] employ

the Dynamic Time Warping (DTW) algorithm [41] to periodically realign the training dataset. DTW is a well-known algorithm to align two sequences, and it does so based on a distance metric defined between each sample of a sequence and each of the other sequence. The complexity of the algorithm[1] is quadratic with the length of the input sequences, therefore, in our implementation, DTW was written using Cython and parallelized to speed-up the computation.

At the beginning of the training step, we initially treat the temporal alignment of the episodes as good enough; we then train the networks, and exploit the learned feature space as a metric function for DTW. We therefore re-align the dataset, re-initialize the network parameters [2], and proceed by training the autoencoders again. This cyclic process is repeated until convergence. Regarding this last point, it is important to note that there are no theoretical guarantees about the convergence of this procedure, nor about its monotonicity in terms of performance of the architecture. More on this topic will be discussed later on in this report.

[1] There exist variants of Dynamic Time Warping that return approximate results and have a lower time-complexity. It is unclear whether [31] used such variants, therefore we prefer to use the exact algorithm.

[2] An alternative to this could be to train for less epochs at each cycle, and then continue to use the same network parameters without re-initializing them. Again, it is not clear whether [31] preferred this option.

4.3 Learning a new task from demonstrations

As mentioned earlier, we use the two autoencoders described in Gupta et al. [31] to learn a shared feature space between states of the two robots; we can then exploit this representation to solve a new task on one of the robots. In particular, we now have two tools to work with: 1) a metric that can tell us how similar is the state of one robot to the state of the other, and 2) a system that can generate reasonable trajectories on one robot starting from real trajectories of the other. In order to perform the transfer and learn a new task on the second robot, we can consider the following strategies:

- Add a new component to the reward function that encourages the policy to imitate the teacher agent, as in [31]. The component can be annealed during training, so that it's dominant when the agent is still exploring.
- Design a simple controller that roughly follows the decoded trajectories, and then use Behavior Cloning to learn a policy directly from these demonstrations.
- Use the same controller as above to pre-train a policy, and then fine-tune it with on-policy methods.

Although [31] found that the first option works well for their experiments, unfortunately, this strategy presents a number of flaws: 1) it requires hand-tweaking of the weight of the additional reward component, which is potentially very time-consuming, 2) it uses a single scalar value to guide complex behavior, losing important information that could speed up the training, 3) it forces the use of dense rewards, which may cause the training phase to get stuck in a local minimum and the agent to stop learning.

The same reasoning can be applied to choose the appropriate algorithm to learn a policy. For instance, PPO requires complex shaped rewards, and even then, it doesn't work well for complex robotics tasks with difficult exploration, as we will see later on. The algorithm that instead works well for these tasks is DDPG with HER, and it has been shown to work particularly well when the rewards are sparse. Because of this, using DDPG with HER together with Behavior Cloning seems the appropriate choice for our objective.

Connecting the pieces together, we obtain a powerful method that exploits the translation capabilities of the learned shared feature space with the advantages

of Behavior Cloning: we call this method *Translated Behavior Cloning*. Figure 4.2 illustrates a simplified overview of it.

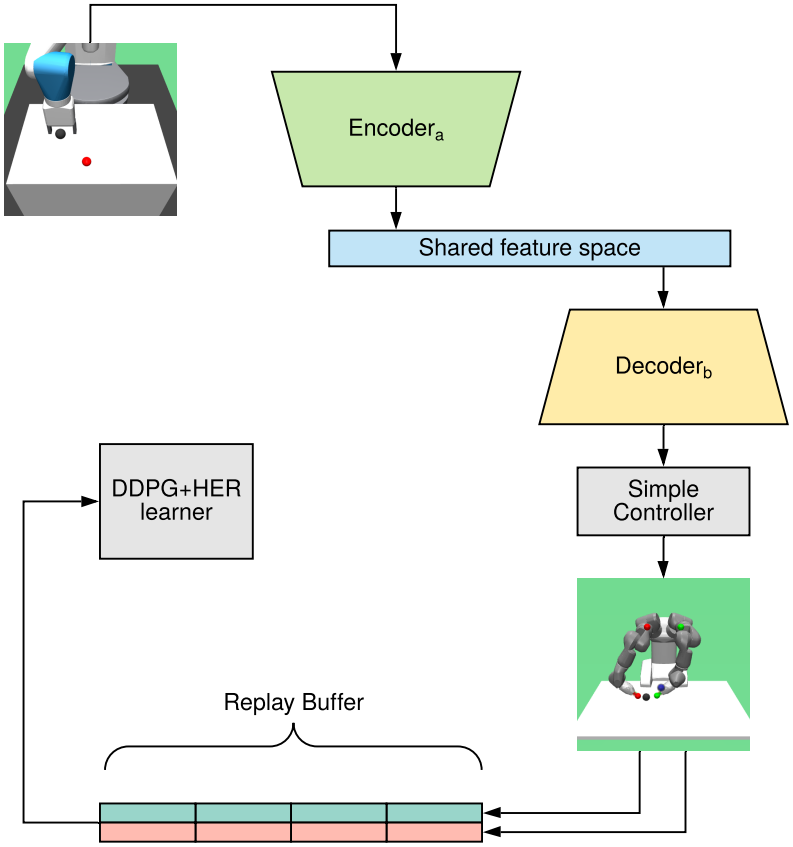


Figure 4.2: Overview of the Translated Behavior Cloning method. In this case, the Fetch robot is acting as the teacher agent, while YuMi is being controlled to imitate trajectories as the student agent. Note that the images here represent the simulated environments, and do not imply that we are feeding frames to the networks.

At each step of the task, we use the learned autoencoders to "cross-decode" the state of the teacher robot into the desired state of the student robot. A simple controller then tries to achieve each desired state, controlling the student robot in a simulated environment. The resulting trajectories are then stored into the replay buffer of DDPG, which will sample transitions using the HER algorithm and learn a successful policy.

The use of a hand-engineered controller at the output of the decoder may seem limiting, but it is important to note that this component can be extremely sim-

ple. In fact, the controller doesn't even have to always succeed, because DDPG with HER will manage to learn even from failed trajectories that the controller might generate. Its role is merely to follow the observations decoded by the network, so that the physical interactions with the environment are always realistic and reflect the data out of the MuJoCo simulator. Without it, the learned policy might fail to learn the dynamics of the environment (particularly those related to the objects the robots interact with); in addition, we wouldn't have the information needed to approximate the optimal action given a state, since the autoencoders do not deal with control signals, but only observations.

4.4 Choosing a Proxy Task

The initial assumption of the framework is that we already have two agents that are able to accomplish the same task controlling two different robots. In practice, the agents could be represented by learned policies (obtained for example with Reinforcement Learning methods), hand-engineered controllers based on planning, or even human operators controlling a manipulator or performing the task directly. In this work, we always use hand-engineered controllers to perform the proxy task, due to the difficulty of learning controllers with RL from scratch (as we empirically show in Sec. 6.1). We also assume that we can reset both environments to the same initial conditions; for instance, if our task involves a moving object, we assume that we can reset the object position to the same point (with respect to the robot base) in both environments. These assumptions may be too strict for some settings, but if the environments are simulated, they are reasonable and usually easy to satisfy.

Choosing a proxy task may be a key aspect for the entire method to work, as the quality of the learned shared feature space depends on it directly. The task must involve at least as many observable environmental elements as our target task(s), since if some are missing, the network wouldn't be able to correctly learn their importance for achieving the goal. Another key aspect that should be considered is the variance and size of the generated dataset; the larger and more diverse the dataset is, the better will be the generalization capacity of the resulting trained models.

Chapter 5

Simulated Environments and Tasks

In order to train and evaluate the agents, the open-source Gym package [42] from OpenAI was used; the package provides a large set of environments with a standard interface, and has been widely adopted by the RL research community. The robotic environments are simulated using the MuJoCo physics engine [43], and include a Fetch manipulator, and a Shadow Dexterous Hand attached to a fixture. An extension of OpenAI's Gym was developed to customize the existing environments and include new ones for a dual-arm manipulator, specifically the YuMi IRB 14000 from ABB. A link to the repository including the source code is reported in Chapter 1.

5.1 Fetch Environment

The Fetch environment [20] consists of a 7 DoF Fetch manipulator with a simple parallel gripper as end-effector, operating above a flat surface. The action controls the position of the end-effector directly, together with the distance between the fingers of the gripper; the orientation of the EE is instead fixed, so that the fingers are perpendicular to the table at all times. Joint limits and collisions are not explicitly discouraged within the reward function, although the simulator treats them as soft constraints. The implementation is the one included in [42]. Figure 5.1 includes frames of the Fetch environment at different stages of the Pick and Place task. A detailed description of the observations from this environment is reported in Table A.1.

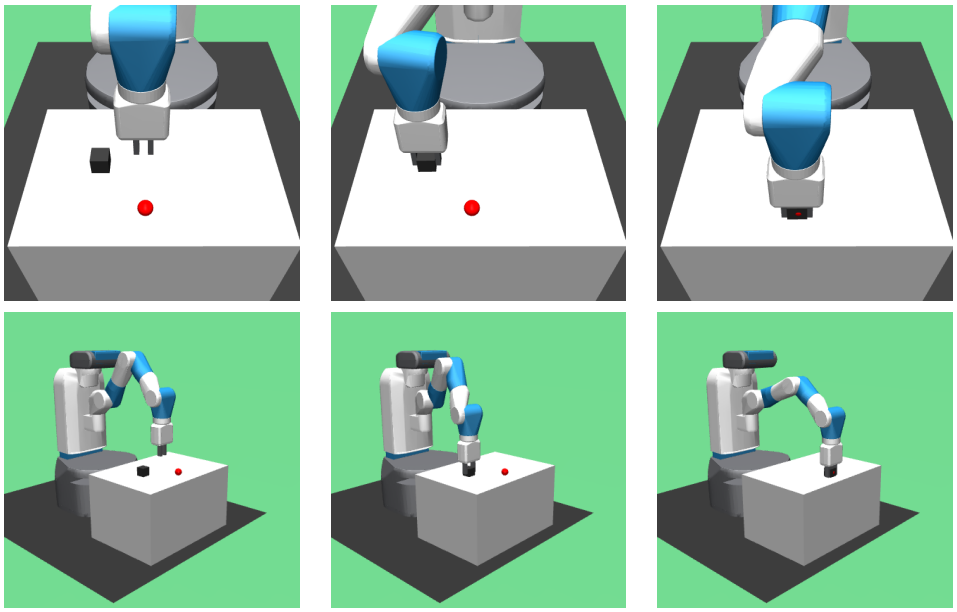


Figure 5.1: Two views of the same Fetch environment at different stages of the Pick and Place task. The red dot here indicates the goal position.

5.2 Shadow Hand Environment

The Shadow Hand environment is a custom adaptation of the Hand environments included in [42]. The joints of the hand have the same properties and are actuated in the same way as in the original implementation, but the forearm has been modified and is now free to move in the 3D space above a table, enabling new tasks such as Pick and Place. For our purposes, we extended the action space to include position control of the forearm, similar to the end-effector control of the Fetch environment. As in the previous case, the orientation of the forearm is fixed so that the palm of the hand is always facing downwards. Figure 5.2 includes frames of the Shadow Hand environment at different stages of the Pick and Place task. A detailed description of the observations from this environment is reported in Table A.2.

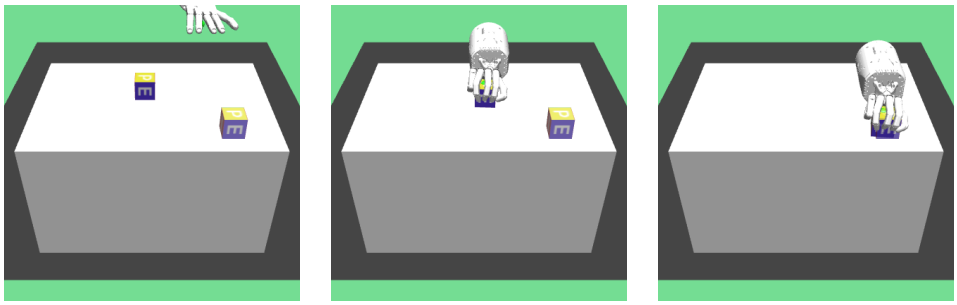


Figure 5.2: The Shadow Hand environment at different stages of the Pick and Place task. The goal position is represented by the semi-transparent cube (lower-right corner of the table).

5.3 YuMi Environment

The YuMi environment was built starting from [44], and consists of a YuMi IRB 14000 dual-arm manipulator (7 DoF for each arm) actuated with position control of all joints. The end-effectors are equipped with parallel grippers, and operate above a table similar to that used in the other two environments.

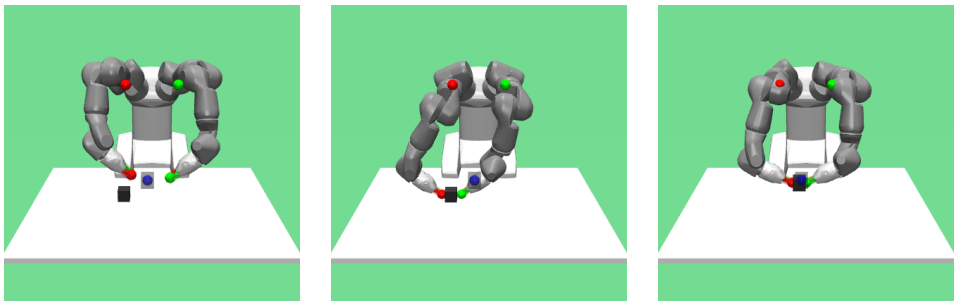


Figure 5.3: The YuMi environment at different stages of the Pick and Place task. The goal position in this case is represented by the blue dot.

A more constrained variant of this environment was also developed to simplify exploration. In this case, the positions of the end-effectors are controlled directly, so that the agent has to output the position delta of the grasp center[1] in cartesian space, together with another value specifying the distance between the two grippers (i.e. a total of 4 control signals). As we will see later on,

[1] By grasp center it is intended the average position of the end-effectors in cartesian space.

this simplification makes a dramatic difference in the effectiveness of RL algorithms. The assumption that we can control the position of the grippers directly is fair, as there are several methods based on Inverse Kinematics that allow us to do this quite easily, and even on real robots. A detailed description of the observations from this environment is reported in Table A.3.

Factors such as self-collisions, singularities, and joint limits are taken into account by the simulator, although for simplicity, we don't take any action to actively avoid them during training; similarly, we don't enforce any limit on the joint velocities required to achieve the desired end-effector positions. In theory, our proposed method is general enough to work regardless of the dynamics and constraints of the environment, therefore, we have no reason to believe that taking into account such factors in future work would be particularly challenging.

5.4 Tasks

5.4.1 Pick and Place

The Pick and Place task is one of the most commonly used in robotics research. The goal is to reach for an object, grasp it, and then move it to a target location. For our purposes, we ignore the object's orientation, and only consider its cartesian position to determine whether a state is successful or not. In addition, we set a 5cm threshold distance to the target, i.e. we say that an episode is successful if the object comes within 5cm of the goal location. For each episode, the object's initial position is randomly sampled (excluding unreachable poses), and so is the target position. The objects used for this task were a small cube and a sphere, with physical properties changing depending on the experiment (details in a later section). Asymmetric objects as well as concave geometries were briefly investigated, but unfortunately could not be used in the final experiments due to technical challenges connected to the MuJoCo simulator.

5.4.2 Push

The setup of the Push task is quite similar to that of the Pick and Place task. At each episode, we begin with an object at a random location on a table, and the agent's goal is to push it to another random location without lifting it. A successful state is determined in the same way as for the Pick and Place

task, and with the same 5cm threshold. This task may seem simpler compared to others, but the fact that the object has to constantly interact with the table introduces new interesting dynamics. In particular, if pushed with enough force, light objects or rolling objects like spheres or cylinders will continue to move on the surface without the agent's control, and may eventually fall on the floor and become unreachable. For this task, a small cube and a sphere were used as for the Pick and Place task.

5.4.3 Button

To make the Pick and Place task more interesting, a simple button can be added to the scene. In this case, the object is unreachable until the button is pressed, so that the agent needs to explore more and learn a multi-step behavior in order to achieve the goal. The position of the button can be easily configured, but for our purposes, we set it to the center of the table. Since the agent cannot directly observe the pose of the button, this is kept fixed and it is not randomized, unlike the initial position of the object in the other tasks. The button is implemented in the Fetch environment, as well as the YuMi environment. Figure 5.4 shows this task in action.

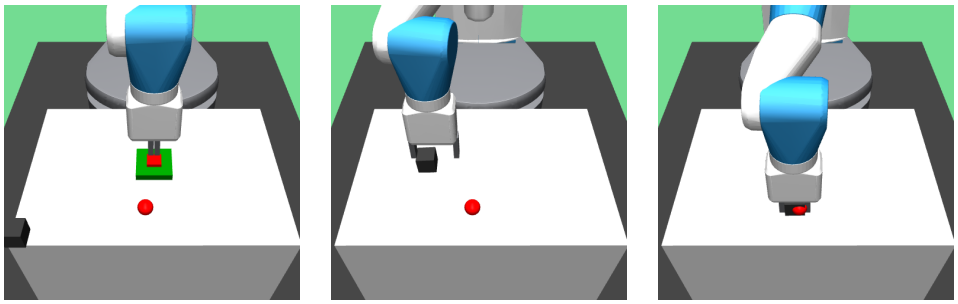


Figure 5.4: The Fetch robot at different stages of the Pick and Place with button task. Once red area around the green object is pressed, the cube becomes available and the task can be completed.

5.4.4 Rotating Platform

Finally, we can add another modifier to the Pick and Place task, increasing the difficulty of the exploration phase even more: a rotating platform. In this case, the robot is forced to interact with the platform, rotating it until the object positioned at its far end becomes reachable; once the object is close enough to the gripper, the task becomes the normal Pick and Place task.

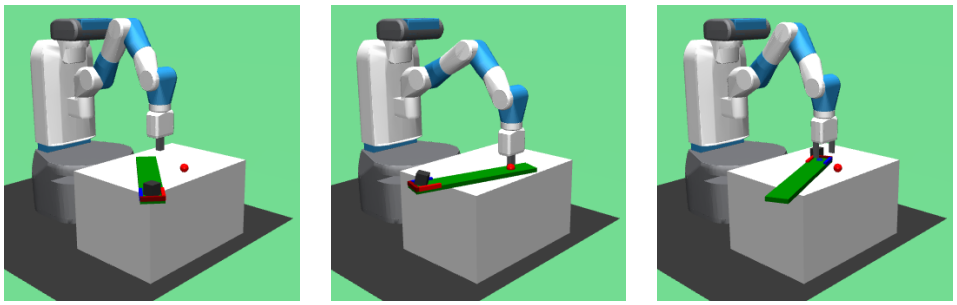


Figure 5.5: The Fetch robot at different stages of the Pick and Place with platform task. In order to reach the object and achieve the task, the agent is forced to first push the rotating platform until it rotates, bringing the cube closer to the gripper.

Chapter 6

Experiments

In this chapter, we present two sets of experiments with different goals. We begin by reporting the results on experiments with robotic agents learning object manipulation skills from scratch. In this case, we use state-of-the-art Reinforcement Learning methods for continuous control, such as DDPG and PPO. This first set of experiments serves to validate our implementations of the relevant RL algorithms, as well as to motivate the need for more efficient methods. In a second set of experiments, we validate our novel technique on several simulated environments. We report results on the effectiveness of the presented method, as well as data on its sample efficiency when compared to learning from scratch. Finally, we discuss the results and comment on the importance of temporal alignment between two robots solving a task when collecting training data.

6.1 Learning from Scratch

6.1.1 Learning manipulation on Fetch

As a first experiment, we start by considering the 7 DoF Fetch robotic arm and the Pick and Place task on a simulated environment, as described previously in Chapter 5. Following Plappert et al. [20] (who also designed and implemented the Fetch environment), we use DDPG with the HER sampler. We use the PyTorch implementation from [45] as a base and modify it[1] for our needs, while keeping the configuration details identical to those used in [20]. This experiment will also serve as validation of the correctness of our implementation. We run the algorithm on 7 parallels threads for a total of 70 epochs, and

[1] `https://github.com/carlo-/hindsight-experience-replay`

take advantage of a GPU to speed up training.

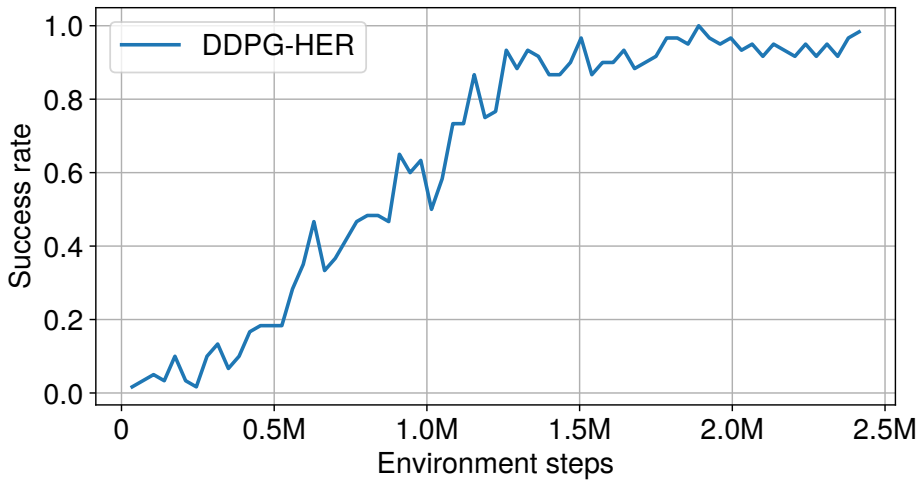


Figure 6.1: Success rate (avg. over 300 episodes) of a DDPG-HER agent as it learns the Pick and Place task from scratch on the Fetch environment. In this case, a sparse reward signal was used.

For this particular task, our implementation of DDPG with HER consistently yields policies that accomplish the goal with high success rate (above 90%) without the need of any human priors (such as demonstrations or a shaped reward signal), but it does so only after a large number of interactions with the environment. This is in line with what has been found in previous work [20].

We can also compare these results with those obtained with PPO, another powerful RL algorithm for continuous control. We use the implementation from [46], which allows us to take advantage of message-passing for parallel computing. Specifically, for this experiments we use 20 CPU cores to collect experience and 7 GPU cards to optimize the neural networks, considerably reducing the amount of time needed to train the agent.

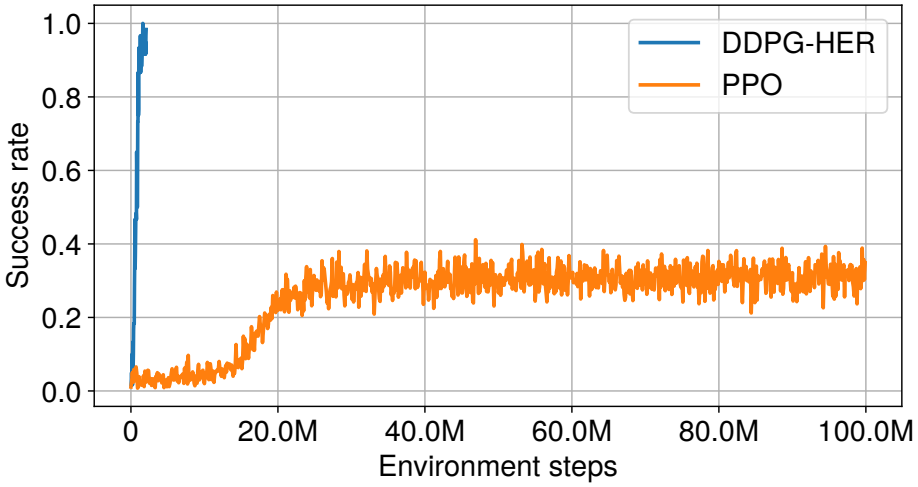


Figure 6.2: Success rate (avg. over 300 episodes) of a PPO agent while learning the Pick and Place task from scratch on the Fetch environment, in comparison with the performance of the DDPG-HER agent from the previous experiment. Unlike the DDPG-HER policy, the PPO policy was trained using a shaped reward signal.

In this case, when using the same sparse reward signal as for DDPG, the agent is unable to learn anything. Following Popov et al. [25], we instead design a "stepped" reward function that guides the exploration towards each sub-step to the goal. Specifically, the agent receives some reward for reaching for the object, then some more for grasping it, more once the object has been lifted, and so on.

As illustrated in Figure 6.2, this second strategy results in better controller capable of solving the task with a 30% success rate. In particular, the policy is able to push the object to the target position, but when the goal is above the table, the controller is incapable of lifting the object, and therefore fails the task in the remaining 70% of cases. We can speculate that with additional tweaking of the reward signal, PPO might find a policy achieving success rates similar to those obtained with DDPG, although the amount of effort required to design such a function would start to look comparable to the engineering needed to design the controller entirely by hand. Another important remark concerns the sample efficiency of the algorithms. The results show how the PPO policy only makes some progress after over 10M interactions with the environment, as opposed to DDPG-HER, which solves the task after collecting 1.5M observations.

6.1.2 Learning manipulation on Shadow Hand

Similar experiments were conducted with the Shadow Hand environment. Despite the task being identical, this environment is much more challenging compared to Fetch, since the learned policy must output separate control signals for each of the fingers' joints, and even a small movement of one of the fingers can cause the hand to drop the object. Once the box falls, there isn't much time for the agent to go back and try again before the episode ends; additionally, the object may become unreachable altogether until the next reset if it falls off the table.

We begin by using DDPG with HER as before. In this case, most of the initial attempts to learn the pick and place task failed, often revealing a need to tweak some parameters of the MuJoCo simulator. Some more successful policies learned to drag the object around the table, but never lifted it when the target position was above the surface.

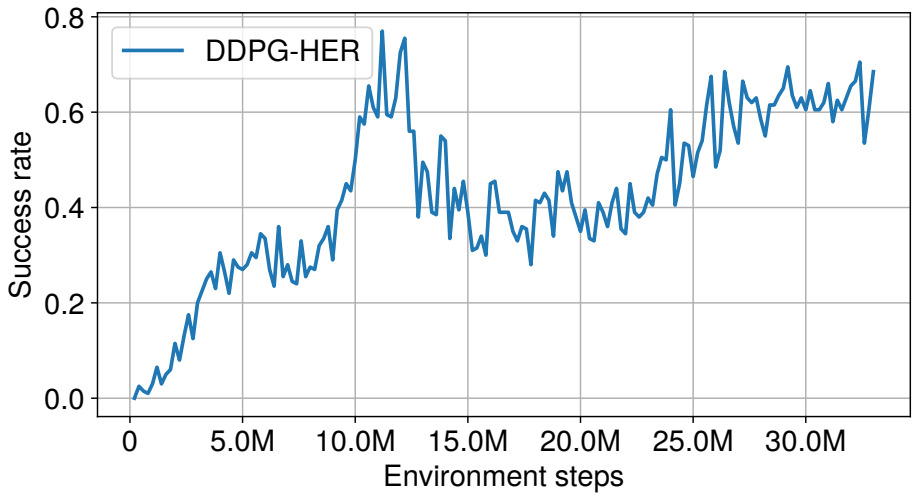


Figure 6.3: Success rate (avg. over 300 episodes) of the DDPG-HER agent as it learns the Pick and Place task from scratch on the Shadow Hand environment. In this case, a sparse reward signal was used.

Several tricks were needed to consistently learn policies that solve the task fully. For instance, inspired by [25], we initialize 20% of the episodes with the object already grasped, so that the agent only has to move the forearm to the goal to receive some reward. In addition, we increased the simulated friction between the fingers and the object to reduce the chance of dropping it. Thanks

to these changes, the DDPG agent was able to achieve success rates above 70%, as shown in Figure 6.3, although only after observing over 10M interactions with the environment. Similarly to what we found in the previous experiment, PPO with a shaped reward signal fails to learn much, with a success rate that remains close to zero even after 100M interactions with the environment.

6.1.3 Learning manipulation on YuMi

Lastly, we experiment with the pick and place task in the YuMi environment. In this case, the robot has less joints compared to the Shadow Hand, but the fact that the two end-effectors can move independently and that they are both required to accomplish the task makes this environment even more challenging. Unsurprisingly, both DDPG with HER and PPO fail to learn anything, even after a large number of environment interactions, as the probability of randomly actuate all joints, pick the object, and bring it to a specific target position is practically zero.

In order to simplify the environment and learn robust controllers for the pick and place task with YuMi, we can introduce motion constraints to the manipulator, and therefore compress the action space down to 4 dimensions, considering the two end-effectors as a single parallel gripper; 3 values control the position of the gripper in Cartesian space (i.e. the center position between the two end-effectors), and one controls the distance between its "fingers" (i.e. the distance between the actual grippers); see also Section 5.3.

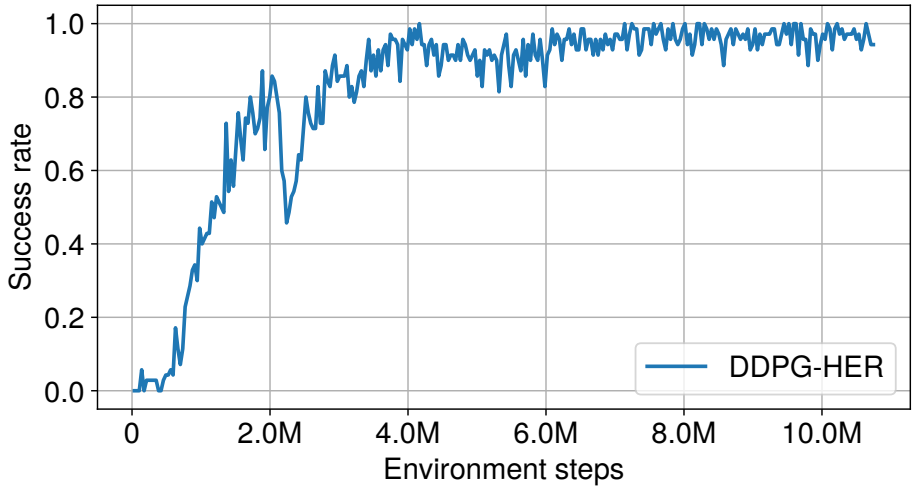


Figure 6.4: Success rate (avg. over 300 episodes) of the DDPG-HER agent as it learns the Pick and Place task from scratch on the YuMi (constrained) environment. In this case too, a sparse reward signal was used.

In this case as well, PPO fails to solve the task, even with carefully shaped reward signals. Instead, Figure 6.4 shows the success rate of a DDPG-HER policy. The learning curve is almost monotonic, and the agent consistently solves the task with a success rate above 90% after observing around 3M environment interactions.

6.1.4 Discussion

As we have seen from these initial experiments, the results suggest that learning a manipulation task on a robot is not possible unless we introduce human priors and/or allow a large number of environment interactions. This motivates the need for methods that reuse past knowledge, or transfer knowledge between agents, so that the number of tasks that must be learned from scratch is reduced. This is especially important if we wish to learn such manipulation skills on real robots: in the real world, objects cannot be moved to an arbitrary position in space with infinite precision, and manipulators cannot be left unsupervised to perform millions of random interactions with their environment. The need for knowledge transfer and reuse is clear.

6.2 Transfer from Shadow Hand to YuMi

We will now present the main experiments performed using our proposed method for skill transfer. We begin by considering the transfer of the object manipulation skill from a Shadow Hand robot to a YuMi manipulator. This is a particularly interesting set-up, as the Shadow Hand is an anthropomorphic robot and can be mapped directly to a human hand, meaning that we could substitute it with a human demonstrator.

6.2.1 Environments and Tasks

In this case, the proxy task was Pick and Place, while the target was Push. For both environments, the object used was a small box, with a side length of 5cm, and a mass of 200g. The initial position of the object was randomly sampled at the beginning of each episode, taking into account the space of reachable end-effector positions for both robots, YuMi in particular. The goal position was randomly sampled with the same criterion, so that the policy is able to generalize well. We used the constrained variant of the YuMi environment as for the previous experiments. The length of each episode was 50 steps for the YuMi environment and 200 steps for the Shadow Hand environment.

6.2.2 Setup

In order to accomplish the proxy task, both robots were controlled by simple hand-engineered agents, although learned controllers could be used as well. The autoencoders were trained with the Adam optimizer [39], using a learning rate of $1e-4$ and batch size 128. The size of the latent space was 15, and the encoders and decoders were built using 3 fully connected layers (60 hidden units) and ReLU activation. The factors of the different contributions of the autoencoder's loss function were set in proportion to the dimensions of the inputs (for instance, if n is the number of input dimensions of the autoencoder a, we set $\lambda_a = \frac{1}{n}$); the factor for the similarity loss (λ_{sim}) was instead fixed to 2. The networks were trained for 14 cycles, each consisting of 80 epochs. After each cycle, the episodes were re-aligned using DTW and the learned similarity loss as metric, and all parameters of the networks were re-initialized. The shared feature space was then used to translate trajectories of a target task – Push box – and train a DDPG-HER agent using Behavior Cloning.

6.2.3 Results

A total of 5000 *twin* episodes of the proxy task where generated, each with the same initial states and goals for both environments. This dataset was then split into training and validation sets (the latter being 20% of the total amount of data) and then used to train the autoencoders. The results of this run are shown in Figure B.1.

In order to assess the quality of the shared feature space obtained at each cycle, we use the success rate of a simple controller as a metric. The controller follows cross-decoded trajectories as explained in Chapter 4, and tries to solve the proxy task for 300 episodes (Figure 6.5). Interestingly, we observe that training the autoencoders for several cycles does not seem to improve the quality of the translated trajectories over time.

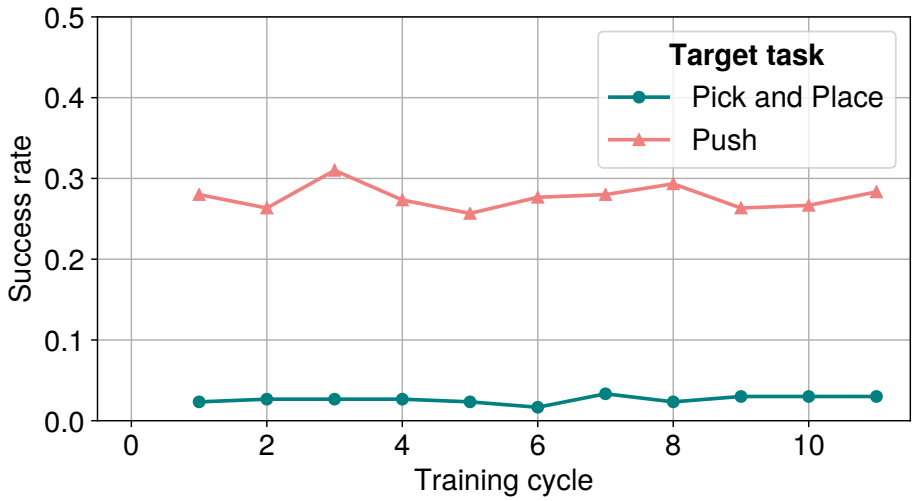


Figure 6.5: Success rate of a controller imitating YuMi trajectories decoded from the autoencoders, starting from real Shadow Hand trajectories. The tasks used for the evaluation were Push as well as Pick & Place (same as the proxy task), and the success rate was computed after each cycle by considering 300 episodes each time.

Additionally, we see that demonstrations of the Push task are properly translated and lead to a success rate of 30% with zero-shot transfer (i.e. without the need of any fine-tuning), while the Pick and Place task (the same used as proxy task to train the autoencoders) is not immediately transferred. This could be due to at least two reasons: 1) The Pick and Place task is challenging, while

the controller used to follow the translated trajectories is extremely simple; a single mistake during an episode will cause the object to fall without the possibility of recovery. 2) The Shadow Hand agent solves the Pick and Place task quite slowly (taking up to 200 steps), while the YuMi agent can be much faster (up to 50 steps), meaning that it is quite difficult to align trajectories of the two robots; differently, the Push task can be accomplished by both in a comparable number of steps.

Having assessed the performance of the shared feature space, we picked the best parameters of the autoencoders according to this metric, specifically those from the 3rd cycle of training. We then used the networks to translate successful episodes from Shadow Hand to YuMi on the target task. Finally, we trained DDPG with HER on the target task with YuMi, exploiting the generated trajectories with Behavior Cloning. The results of the final performance of this setup are shown in Figure 6.6, together with the success rate of a policy learned from scratch for reference.

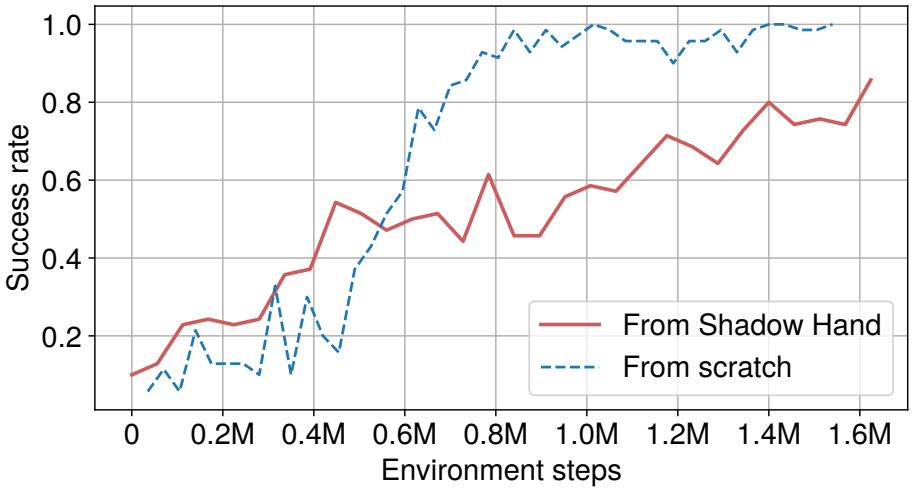


Figure 6.6: Push box task as target. Success rate (avg. over 300 episodes) during training from teacher vs learning from scratch.

In this case, we observe that the policy learning from translated demonstrations from Shadow Hand has initially an advantage over the agent learning from scratch, but after 500k environment steps the second agent takes over and learns a more robust policy.

6.3 Transfer from Fetch to YuMi

This final set of experiments attempts the transfer of object manipulation skills from a Fetch robot to a YuMi manipulator. From a certain perspective, this second setup may seem to correspond to a more relaxed problem compared to the previous one, since the mapping between a single arm and two arms with parallel grippers is simpler. On the other hand, having two end-effectors to work with enables a large set of possible interactions with an object that would not be possible with a single gripper; for this reason, the mapping in task space between these two different morphologies may actually be rather complex.

6.3.1 Environments and Tasks

The YuMi environment is the same as the one considered in the previous section, while the Fetch environment is the original implementation from [42], with the exception of the object being manipulated. The proxy task was Pick and Place of a box (side of 5cm, mass of 2Kg, as in the original Fetch environment), while several target tasks were explored (Table 6.1). Details of each task are given in Chapter 5.

Task	Object type	Size	Steps
Push	Box	5cm (side)	50
Push	Sphere	5cm (diameter)	50
Pick & Place with platform	Box	5cm (side)	80
Pick & Place with button	Box	5cm (side)	50

Table 6.1: Target tasks explored in the experiments in this section.

6.3.2 Setup

In order to accomplish the proxy task, as in the previous case, both robots were controlled by hand-engineered agents, although learned controllers could be used as well. The autoencoders were trained with the Adam optimizer with learning rate $1 \cdot 10^{-4}$ and batch size 128. The size of the latent space was 15, and the encoders and decoders were built using 3 fully connected layers (60 hidden units) and ReLU activation. The factors of the different contributions of the autoencoder's loss function were set as in the previous set of experiments.

6.3.3 Results

As a first step, a total of 5000 *twin* episodes where generated, each with the same initial states and goals for both environments. This dataset was split into training and validation sets (the latter being 20% of the total amount of data) and was then used to train the autoencoders, with the goal of learning a shared latent space between states of the two robots. The results of this first phase can be seen in Figures B.2 and B.3. Similarly to what we saw earlier on, we see that the quality of reconstructed trajectories doesn't improve at each cycle, and even peaks at the end of the first, i.e. before we run the DTW algorithm for the very first time. We also observe that the success rate does not seem to converge to a fixed value, which is in contrast with what Gupta et al. [31] found in their work.

As before, we picked the best pair of autoencoders based on the zero-shot transfer performance, and proceeded by training a DDPG-HER agent with demonstrations translated from the teacher robot. As shown in Figures 6.7, 6.8, and 6.9, with our method we were able to successfully transfer manipulation skills from Fetch – a single arm manipulator – to YuMi – a dual arm robot – for most tasks. More specifically, all tasks were transferred with a success rate above 90%, with the exception of the Pick & Place task with the rotating platform.

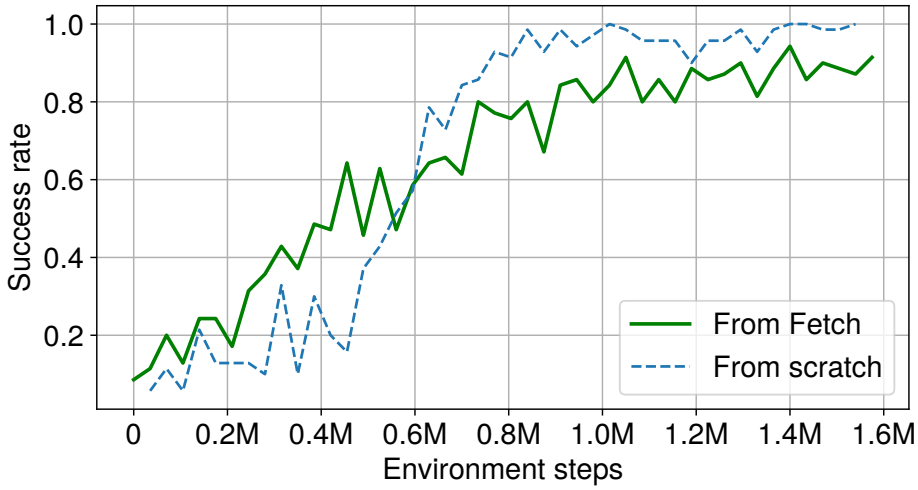


Figure 6.7: Push box task as target. Success rate (avg. over 300 episodes) during training from teacher vs learning from scratch.

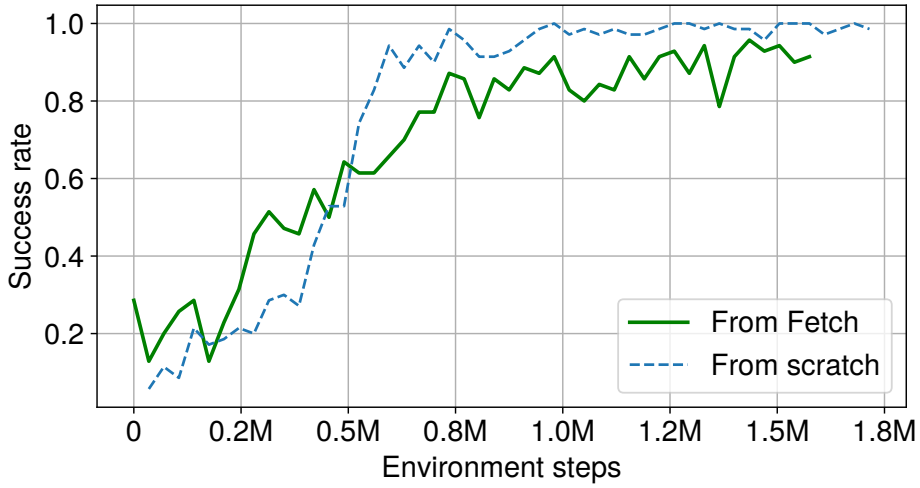


Figure 6.8: Push sphere task as target. Success rate (avg. over 300 episodes) during training from teacher vs learning from scratch.

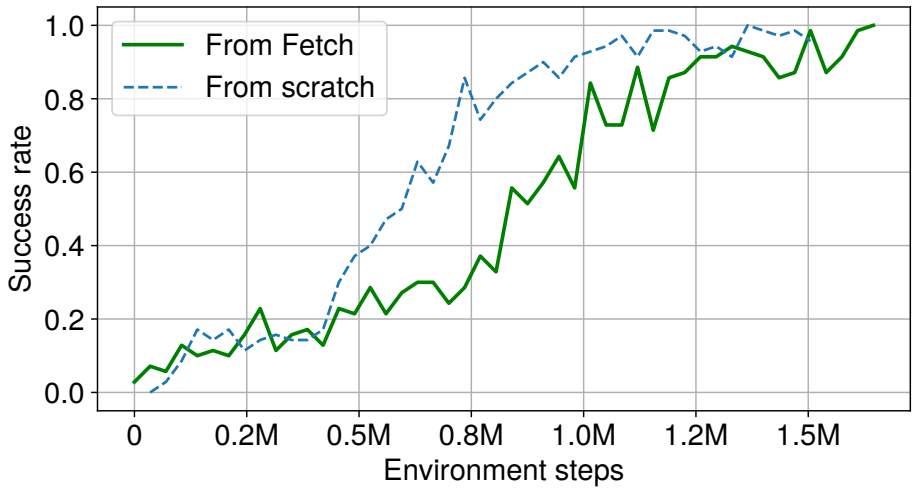


Figure 6.9: Pick & Place with button task as target. Success rate (avg. over 300 episodes) during training from teacher vs learning from scratch.

For the Pick & Place with platform task instead, the controller was unable to generate reasonable trajectories to insert into the replay buffer of DDPG, therefore we cannot report any data regarding this experiment. This is probably due to the fact that this particular task is too different from the proxy task, and requires the agent to execute trajectories in a subset of the cartesian space that

rarely appears in the training dataset. Other issues are related to the difficulty of the task itself. For instance, the object is initially positioned at the far end of the platform and there is a high chance that it will fall off during exploration, remaining unreachable until the next reset.

In general, by observing the plots in this section, we can immediately recognize a trend: learning from both Fetch and Shadow Hand is possible, but seems to require more interactions with the environment compared to learning from scratch. This could be due to the fact that policies trained with Behavior Cloning know very well how to act in the space of states that are relevant for completing the task, but output noise when stepping outside this area. Differently, policies trained from scratch begin by heavily exploring the state space almost uniformly, and therefore they may quickly become more robust, speeding up the learning process. This is also supported by the fact that TBC policies seem to learn faster at the beginning of each run, but then slow down later on because of the initial bias imposed by the translated demonstrations.

6.4 Positive Impact of Transfer on Exploration

Other interesting results suggest that TBC may be quite useful when the task we are trying to learn requires a lot of exploration. Figures 6.10 and 6.11 show how the agent is immediately encouraged to reach for the object when training with TBC. Instead, the agent that learns from scratch is forced to first explore randomly, and then eventually catches up when stumbling upon the object. We can speculate that if the object weren't so easily accessible, the random policy would have had a hard time making progress. Environments such as the one with the rotating platform could be used to verify this aspect. We also observe that the agents trained with TBC solve the Push tasks by getting much closer to the objects. This is because the proxy task – Pick & Place – requires the agent to surround the object with the gripper(s) before grasping it, hence getting as close as possible to its center, and this behavior is being transferred to the Push task. Differently, the agent trained from scratch has learned to push with any part of the end-effectors, including the external parts of the grippers.

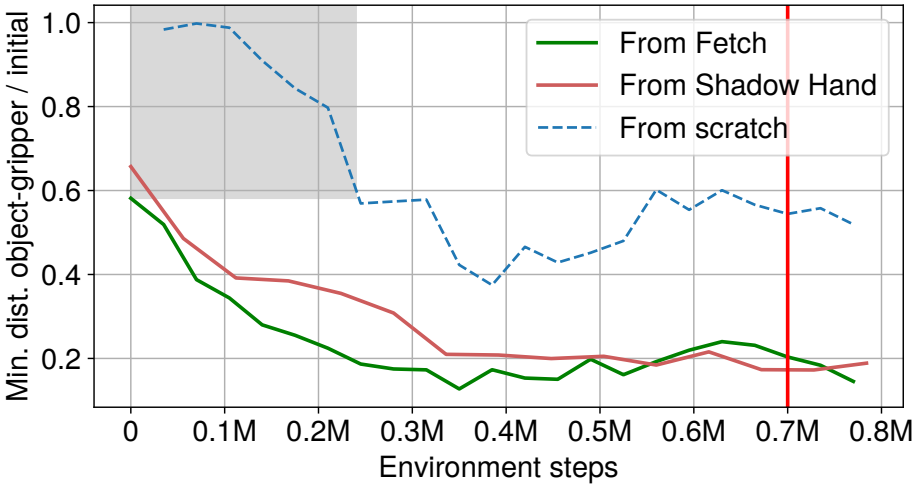


Figure 6.10: Minimum distance object-gripper over initial, averaged from different test episodes during training, with Push box as target task. The gray area is a rough estimate of the region where the agent isn't interacting with the object, while the red vertical line represent the number of steps needed to learn the task from scratch with $> 80\%$ success rate. For the same plot with the full x-axis, see Figure B.4.

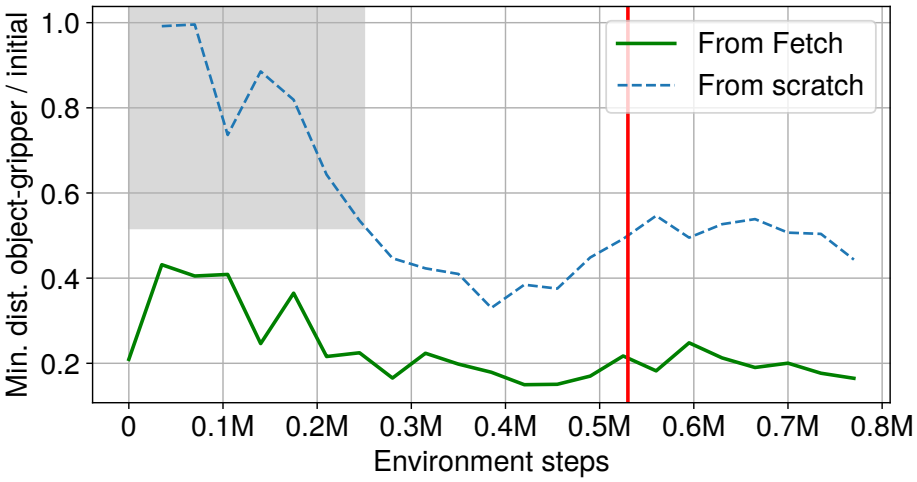


Figure 6.11: Same visualization as Fig. 6.10, but with Push sphere as the target task. For the same plot with the full x-axis, see Fig. B.5.

Chapter 7

Conclusion

In this work, we have presented a new set of simulated environments to develop and test techniques for transferring object manipulation skills between robots with different morphologies. The environments provide a framework to learn and transfer tasks such as Pick & Place and Push, as well as multi-step tasks involving rotating platforms and buttons. They are built using the standard APIs from OpenAI's Gym package, they are open-source and available on GitHub.

We have introduced a novel technique, called Translated Behavior Cloning, that aims to solve the skill transfer problem by first leveraging an existing method to learn a shared feature space between states of the two robots, and by then exploiting this representation to transfer skills with model-free Reinforcement Learning techniques. The framework requires little hand-engineering, and it is designed to work with DDPG with HER, although it could be adapted to to work with other off-policy algorithms, such as SAC.

We saw positive results in the transfer from a manipulator with a single end-effector to a dual-arm robot, as well as encouraging results in the transfer starting from an anthropomorphic hand, despite the considerable differences in terms of morphology and actuation. Interestingly, the results highlighted the fact that the cyclic use of the Dynamic Time Warping algorithm employed by [31] doesn't seem to particularly help the learning process of the shared feature space. The details of their implementation are not fully specified in the publication, therefore it is possible that this discrepancy may simply be due to different hyperparameters.

We also saw that TBC is unfortunately not as sample efficient as desired, and therefore cannot be applied to physical machines with reasonable amounts of resources. Although this aspect highlights a recurrent issue of Reinforcement Learning applied to robotics, we aim to address it in future research projects.

Finally, we reported very encouraging results highlighting the impact of TBC on exploration. Policies trained from translated demonstrations seem to be immediately aiming for the goal without any fine-tuning, while agents learning from scratch spend hundreds of thousands of steps in subsets of the state-space that are not relevant for the task. This characteristic of TBC policies could potentially make a difference between solving a task that requires hard exploration and not solving it at all, although it is not obvious how to design environments to properly test this hypothesis.

7.1 Future Work

Further research could focus on experimenting with different tasks, such as rearranging multiple objects, as well as different manipulators. The algorithm we presented could be modified so that the dominance of the behavior cloning loss is annealed during training; in this way, we may be able to guide the exploration without biasing the policy too much. The structure to learn the shared feature space could be substituted with architectures similar to those used for text translation (such as Recurrent Neural Networks), so that past states are taken into account; such a technique may also be less sensible to the alignment of the trajectories of the two robots. Future work should also investigate the importance of the choice of the proxy task, and how multiple tasks could be combined to transfer even more complex skills. Another important aspect to consider is the sample efficiency of the learning algorithms used; since the goal is ultimately to exploit these techniques with real robots, the number of interactions with the environment needed should be reduced as much as possible.

Bibliography

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[3] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. Vol. 37. 1994.

[4] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[5] Javier García and Fernando Fernández. "A comprehensive survey on safe reinforcement learning". In: *Journal of Machine Learning Research* 16.1 (2015), pp. 1437–1480.

[6] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/.

[7] Adam Paszke et al. *Automatic differentiation in PyTorch*. 2017. URL: https://pytorch.org/.

[8] Diederik P Kingma and Max Welling. "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[9] Huaibo Huang et al. "IntroVAE: Introspective Variational Autoencoders for Photographic Image Synthesis". In: *NeurIPS*. 2018.

[10] Samuel R. Bowman et al. "Generating Sentences from a Continuous Space". In: *CoNLL*. 2016.

[11] Prafulla Dhariwal et al. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[12] Eric Liang et al. "RLlib: Abstractions for Distributed Reinforcement Learning". In: *ICML*. 2018.

[13] Josh Achiam. *Spinning Up in Deep RL*. 2019. URL: https://spinningup.openai.com (visited on 06/01/2019).

[14] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017).

[15] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning". In: *CoRR* abs/1509.02971 (2015).

[16] Boris T Polyak and Anatoli B Juditsky. "Acceleration of stochastic approximation by averaging". In: *SIAM Journal on Control and Optimization* 30.4 (1992), pp. 838–855.

[17] Marcin Andrychowicz et al. "Hindsight Experience Replay". In: *NIPS*. 2017.

[18] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), p. 529.

[19] Tuomas Haarnoja et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *arXiv preprint arXiv:1801.01290* (2018).

[20] Matthias Plappert et al. "Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research". In: *CoRR* abs/1802.09464 (2018).

[21] Ricson Cheng, Arpit Agarwal, and Katerina Fragkiadaki. "Reinforcement Learning of Active Vision for Manipulating Objects under Occlusions". In: *CoRL*. 2018.

[22] David Warde-Farley et al. "Unsupervised Control Through Non-Parametric Discriminative Rewards". In: *CoRR* abs/1811.11359 (2019).

[23] Sergey Levine, Nolan Wagener, and Pieter Abbeel. "Learning contact-rich manipulation skills with guided policy search". In: *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2015, pp. 156–163.

[24] Shixiang Gu et al. "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates". In: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2017, pp. 3389–3396.

[25] Ivaylo Popov et al. "Data-efficient Deep Reinforcement Learning for Dexterous Manipulation". In: *CoRR* abs/1704.03073 (2018).

[26] Aravind Rajeswaran et al. “Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations”. In: *CoRR* abs/1709.10087 (2018).

[27] OpenAI et al. “Learning Dexterous In-Hand Manipulation”. In: *CoRR* abs/1808.00177 (2018).

[28] Sergey Levine et al. “Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection”. In: *The International Journal of Robotics Research* 37.4-5 (2018), pp. 421–436.

[29] Chelsea Finn, Pieter Abbeel, and Sergey Levine. “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks”. In: *ICML*. 2017.

[30] Coline Devin et al. “Learning modular neural network policies for multi-task and multi-robot transfer”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)* (2017), pp. 2169–2176.

[31] Abhishek Gupta et al. “Learning Invariant Feature Spaces to Transfer Skills with Reinforcement Learning”. In: *CoRR* abs/1703.02949 (2017).

[32] Staffan Ekvall and Danica Kragic. “Robot learning from demonstration: a task-level planning approach”. In: *International Journal of Advanced Robotic Systems* 5.3 (2008), p. 33.

[33] George Konidaris et al. “Robot learning from demonstration by constructing skill trees”. In: *The International Journal of Robotics Research* 31.3 (2012), pp. 360–375.

[34] Leonel Rozo, Pablo Jiménez, and Carme Torras. “A robot learning from demonstration framework to perform force-based manipulation tasks”. In: *Intelligent service robotics* 6.1 (2013), pp. 33–51.

[35] Yezhou Yang et al. “Robot learning manipulation action plans by" Watching" unconstrained videos from the world wide web”. In: *Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.

[36] Ashvin Nair et al. “Overcoming Exploration in Reinforcement Learning with Demonstrations”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)* (2018), pp. 6292–6299.

[37] Brenna Argall et al. “A survey of robot learning from demonstration”. In: *Robotics and Autonomous Systems* 57 (2009), pp. 469–483.

[38] Herbert Robbins and Sutton Monro. “A stochastic approximation method”. In: *The annals of mathematical statistics* (1951), pp. 400–407.

[39] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2015).

[40] Sumit Chopra, Raia Hadsell, and Yann LeCun. “Learning a Similarity Metric Discriminatively, with Application to Face Verification”. In: *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1*. 2005, pp. 539–546.

[41] Meinard Müller. “Dynamic time warping”. In: *Information retrieval for music and motion* (2007), pp. 69–84.

[42] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[43] 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* (2012), pp. 5026–5033.

[44] Isac Arnekvist. *Open AI gym - extended with YuMi environment*. `https://github.com/isacarnekvist/open-ai-yumi`. 2018.

[45] Tianhong Dai. *PyTorch implementation of Hindsight Experience Replay*. `https://github.com/TianhongDai/hindsight-experience-replay`. 2019.

[46] Ashley Hill et al. *Stable Baselines*. `https://github.com/hill-a/stable-baselines`. 2018.

Appendix A

Details of the Environments

Input	Dimensionality
gripper position	3D
gripper velocity	3D
object position	3D
object relative position	3D
finger positions	2D
finger velocities	2D
object orientation	3D
object velocity	3D
object angular velocity	3D
achieved goal	3D
desired goal	3D

Table A.1: Observations used as input to the policy for the **Fetch** environment. For DDPG-HER policies, the goals were kept separate and used only by the HER sampler; instead, for PPO the desired goal was appended to the observation vector.

Input	Dimensionality
forearm position	3D
forearm orientation	3D
forearm velocity	3D
palm position	3D
object position	3D
object orientation	3D
object velocity	3D
object relative position	3D
hand joint angles	24D
hand joint velocities	24D
achieved goal	7D
desired goal	7D

Table A.2: Observations used as input to the policy for the **Shadow Hand** environment. Goals are handled as explained in Table A.1.

Input	Dimensionality
grasp center position	3D
grasp center velocity	3D
position of grippers	6D
velocity of grippers	6D
object position	3D
object orientation	3D
object relative position	3D
object velocity	3D
achieved goal	3D
desired goal	3D

Table A.3: Observations used as input to the policy for the **YuMi** constrained environment. Goals are handled as explained in Table A.1.

Appendix B

Additional Results

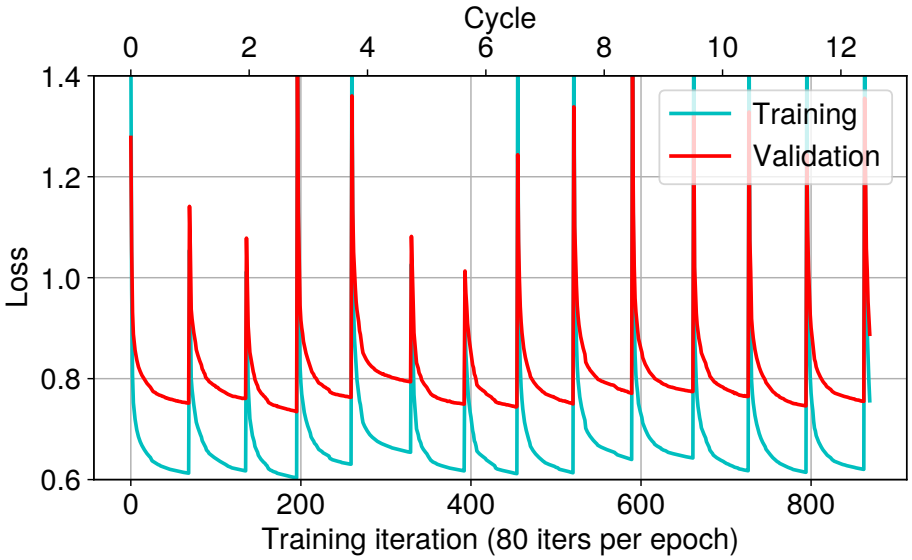


Figure B.1: Loss (averaged) when training the two autoencoders on the Pick & Place task with Shadow Hand and YuMi. After each cycle, the networks are re-trained from scratch, which explains the spiking pattern.

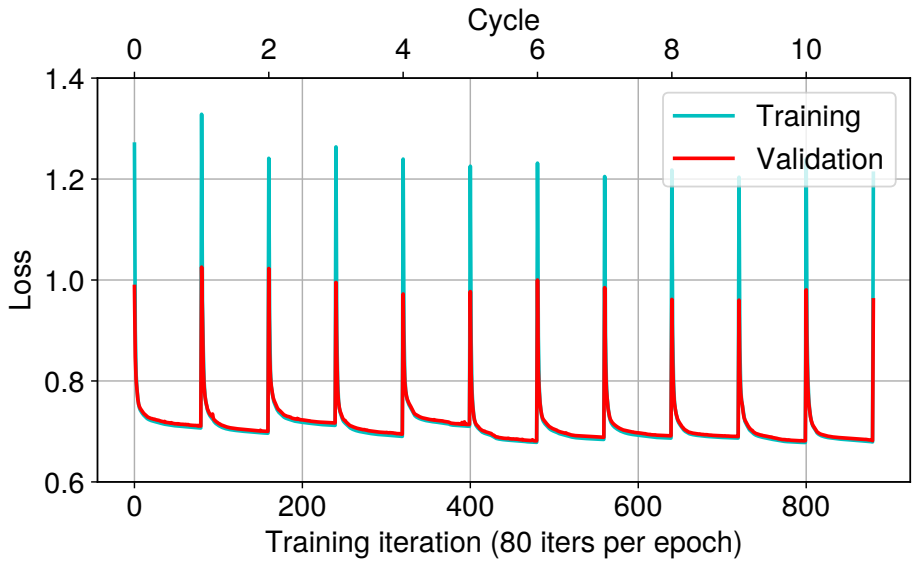


Figure B.2: Loss (averaged) when training the two autoencoders on the Pick & Place task with Fetch and YuMi. After each cycle, the networks are re-trained from scratch, which explains the spiking pattern.

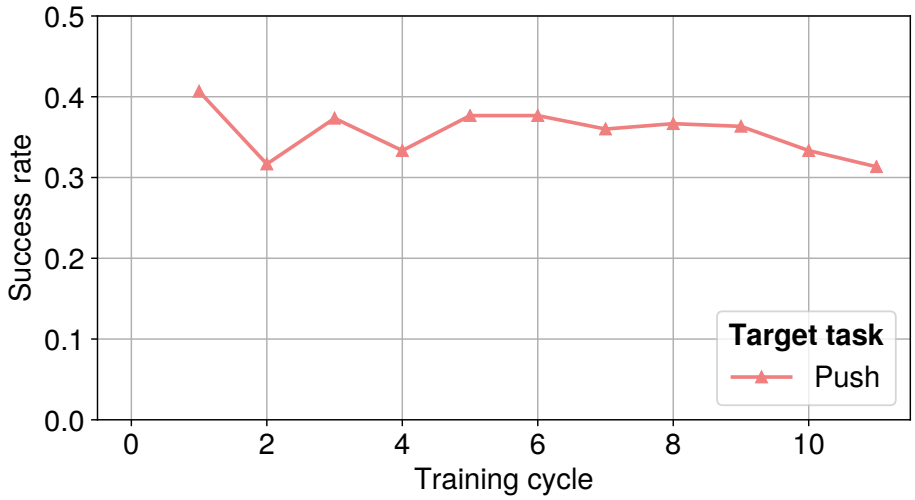


Figure B.3: Success rate of a controller imitating YuMi trajectories decoded from the autoencoders, starting from real Fetch trajectories. The task used for the evaluation was Push, and the success rate was computed after each cycle by considering 300 episodes each time.

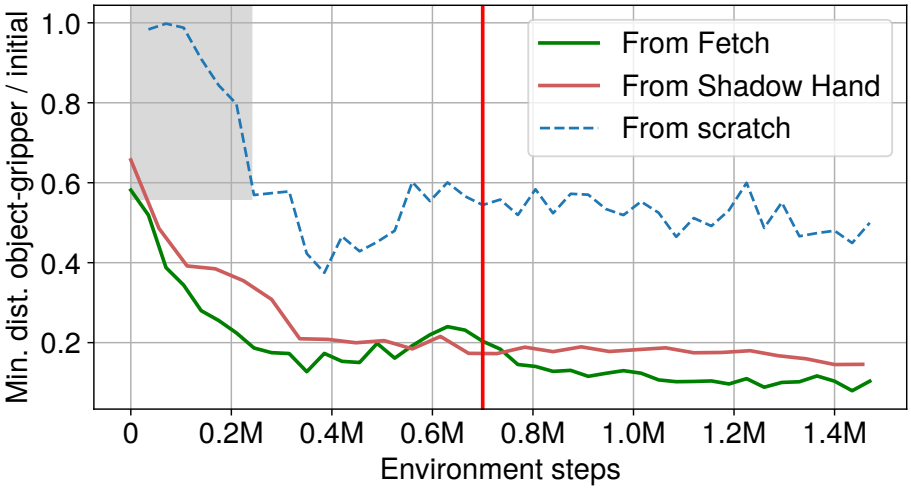


Figure B.4: Full version of Fig. 6.10.

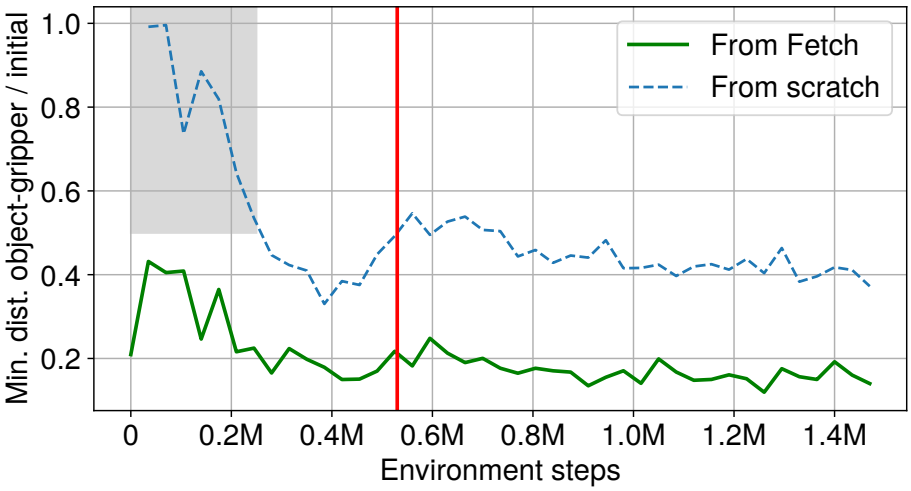


Figure B.5: Full version of Fig. 6.11.

TRITA -EECS-EX