Representation Learning for In-hand Object Manipulation

Gonçalo Reis Rodrigues Chambel

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Ana Catarina Fidalgo Barata
Prof. Bruno Duarte Damas

Examination Committee

Chairperson: Prof. João Fernando Cardoso Silva Sequeira
Supervisor: Prof. Ana Catarina Fidalgo Barata
Member of the Committee: Prof. Jacinto Carlos Peixoto do Nascimento

September 2021

Declaration

I declare that this document is an original work of my own and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to start by thanking both my supervisors, Prof. Ana Catarina Fidalgo Barata e Prof. Bruno Duarte Damas. The guidance, support and knowledge that they shared was beyond what I can express in words. Needless to say that this thesis would not be as it is without their help. I would also like to personally thank Prof. Alexandre José Malheiro Bernardino for enabling constant meetings with other masters students and providing helpful feedback throughout the making of this thesis.

I would also like to thank my course colleagues for sharing this journey with me and for allowing me to have the college experience I had. I would like to personally mention João Revés for being constantly interested in my work and for giving me almost 20 years worth of great memories.

This thesis would also not be possible if my work colleague and friend Joana Pinto was not so open with me taking time off to dedicate to the development of this thesis.

Last but not least, I would like to thank my closest family for allowing me to grow into the person I am today, and supporting and enabling my life choices. A special thanks to Beatriz Neto D'Almeida for keeping me motivated during this thesis and for sharing the pain of writing a thesis while working.

To each and every one of you – Thank you.

Abstract

More and more often, humans rely on robots to replace them in performing simple tasks. However, the ability of robots to adapt and perform more complex tasks is still reduced, specially when it comes to dexterous tasks. The goal of this thesis is to develop a model that is capable of using in-hand manipulation to control a set of objects using Deep Reinforcement Learning (RL) methods, based on the tactile feedback obtained from a simulated robotic hand. This is a complex problem for Deep RL algorithms, due to the large number of variables to account for. To bypass this, this work intends to explore the possibility of using a latent representation of the state, in order to test if the updated model learns faster and/or better to perform a set of predefined tasks in a simulated environment. This representation is obtained from another model and should be comprised of only the most important features of the original state. Moreover, to further reduce the number of variables, this work evaluates the performance difference in using different action spaces and synergies to control the robotic hand. The main result of this work is that by using the latent representation of the input, the controller is able to generalize the manipulation to objects that have unique shapes. Regarding the different action spaces, this work also demonstrates that by using synergies, with a dimension of less than half than the original dimension, the controller is able to achieve the same level of performance.

Keywords

In-hand object manipulation; Deep Reinforcement Learning; Representation Learning; Robotic hand.

Resumo

Os humanos estão cada vez mais dependentes de robôs que sejam capazes de nos substituir na realização de tarefas simples. No entanto, a capacidade dos robots de se adaptarem e realizarem tarefas mais complexas, especialmente tarefas que requerem destreza, é reduzida. Esta tese tem como objectivo o desenvolvimento de um modelo que seja capaz de manipular um conjunto de objectos usando uma mão simulada, através de métodos de Aprendizagem por Reforço, baseados em sensores de toque. Isto apresenta um problema para a maior parte dos métodos de Aprendizagem por Reforço, dado que existe um elevado número de variáveis a ter em consideração. De modo a contornar este problema, este trabalho propõe a possibilidade de usar uma representação latente do estado, de modo a testar se o modelo aprende mais rápido e/ou melhor a realizar um conjunto de tarefas. Esta representação é obtida através de um segundo modelo e deve condensar as características mais importantes do estado atual. Adicionalmente, este trabalho pretende avaliar a hipótese de usar sinergias para controlar a mão, de modo a reduzir o número de dimensões das ações. O resultado mais notável desta tese é que ao usar a representação do estado, o controlador é capaz de generalizar o controlo a objectos com formas únicas. Relativamente ao espaço das ações, também é mostrado que ao usar sinergias, é possível reduzir a dimensão das ações para menos de metade, mantendo o nível de performance.

Palavras Chave

Manipulação de objectos; Aprendizagem por Reforço; Aprendizagem de Representação; Mão robótica.

Contents

List of Figures

List of Tables

Acronyms

AE	Autoencoder
DAPG	Demo Augmented Policy Gradient
DBN	Deep Belief Network
DF	Decision Forest
DL	Deep Learning
DMP	Dynamic Motor Primitives
DOF	Degrees of Freedom
DP	Diritchlet Process
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
MDP	Markov Decision Process
ML	Machine Learning
MSE	Mean Squared Error
NPG	Natural Policy Gradient
NPREPS	Non-parametric Relative Entropy Policy Search
PPO	Proximal Policy Optimization
PSR	Predictive State Representation
REPS	Relative Entropy Policy Search
RL	Reinforcement Learning
SAC	Soft Actor-Critic
SVM	Support Vector Machine
TRPO	Trust Region Policy Optimization

1 Introduction

Contents

1.1 Motivation

Robots of all kinds have been present in our lives for the past years. They have become an essential part of our modern lives, whether to replace us when performing dangerous tasks, such as disarming bombs, or to make our lives easier when performing simpler tasks, such as moving objects from place to place.

But even though robots are becoming an important part of our daily lives, most of them still perform tasks that are simple for us, although not for a robot. For example it took years of development to design a robot capable of moving objects in a warehouse (such as the HANDLE robot from Boston Dynamics for example [10]), while this can be a relatively easy task for a human. With the increasing capacity of technology, comes an increasing demand for robots that are more versatile, capable of learning for themselves and adapt to new environments and tasks, robots that can perform more complex tasks. The need to have more complex robots allied with the increasing possibilities that technology enabled, allowed robots to become more useful to society. There are some examples that are already present in our lives like self-driving cars (e.g. as the Waymo cars), and robots that can walk and interact with people (e.g the ATLAS robot from Boston Dynamics [11]).

An important breakthrough that allowed robots to learn more complex tasks was the appearance of cameras that are able to take depth images [12]. This improved the way robots perceived the surrounding environment. Such breakthrough affected the way robots were developed, specially domestic robots. This allowed for robots to be able to perform tasks like picking up objects and placing them somewhere else, with much ease compared to not using vision. Although the ability to estimate the depth of objects was key in robots' development, most robots are still far from achieving human-like performance in several tasks.

Another aspect that contributed to the development of more complex robots were the advances in Artificial Intelligence and Internet of Things. This allowed for robots to now be able to assist humans in every-day tasks, such as cleaning the floor. But despite these breakthroughs, there are still some research fields with potential to evolve, such as the ability to precisely manipulate, in-hand, objects, *i.e.* manipulating an object by solely using the fingers and wrist for example. Although this may seem a simple task for most people, it is a challenge to implement the same level of precise manipulation in a robot. There have been promising advances in this field that will be described in Chapter 2, but this is a problem that is not yet solved. For instance, it is hard to devise a robot capable of using the full range of motion that a human hand has, to perform simple tasks such as picking up an object and manipulate it. It is also hard to develop a robot that is capable of performing multiple tasks at the same time, *i.e.* a robot that can, for example, grab a pen and rotate it, or write with it, or even lift it off the palm of the hand.

An additional key aspect of in-hand manipulation, that this thesis intends to explore, is related to the fact that, in early life stages, humans do not have the ability to make full use of the potential of the human hand. In fact, babies tend to grab objects with all 5 fingers and rarely make more precise grasps such as the pinch grasp. Moreover, babies tend to grab and manipulate objects without a specific objective, other than curiosity. This type of manipulation is a key aspect of this work which will be further described in Chapter 3.

Finally, this thesis intends to explore the possibilities of using Representation Learning, within the field of in-hand manipulation. Representation Learning is a Machine Learning (ML) field that allows a system to autonomously discover the representation of a signal. It can be used in various scenarios [13] but the goal in this thesis is to be able to extract the more useful sections of the signal, that are important when it comes to in-hand manipulation, allowing the model to learn faster.

1.2 Problem Statement

As discussed in the previous section, the advances in Computer Vision provided powerful tools for robots to perceive the surrounding environment, and more importantly, to act upon it. In the field of in-hand manipulation, this means that now robots had a way to ”view” the object being manipulated and control it, in order to achieve higher precision in a particular task. However, while vision-based methods have achieved good results in this field (as it will be shown in Chapter 2), they can lead to hard to solve problems, such as not being able to infer important properties of the manipulated object (e.g. its weight or friction). Additionally, visual information can be hampered by possible occlusions caused by the hand itself.

There is another reason as to why developing an in-hand manipulation capable robot is a difficult task. A healthy human hand has 27 Degrees of Freedom (DOF) [14] which means that if one was to design a physical or even simulated accurate model of the hand, it would need to have around the same number of DOF. Although it is feasible to design such model, implementing an algorithm that is able to control that model, i. e. actuate on each DOF of the hand in order to achieve a particular task, is much more difficult. Suppose we have a general model that has 27 actuators, where each actuator controls a different aspect of the model. If it is also assumed that each actuator can only take up to 10 discrete values, the total number of possible configurations of that model is 10^{27}, which is considerably more than the number of grains of sand in our planet (which is estimated to be about 7.5^{18}). Implementing a hand-coded algorithm would be physically impossible at the current time, but with the advances in Reinforcement Learning (RL) it is possible to considerably reduce the number of combinations that need to be tested. There will be a more in depth explanation of RL in Chapter 2 but in a nutshell, RL is a subset of ML in which the algorithms are able to learn valuable information from exploring random states (combinations) to then make a more informed decision when a new state is achieved. Even though this allowed for the development of interesting manipulation models, using a model with many DOF, like the human hand, is still a obstacle for state of the art RL methods.

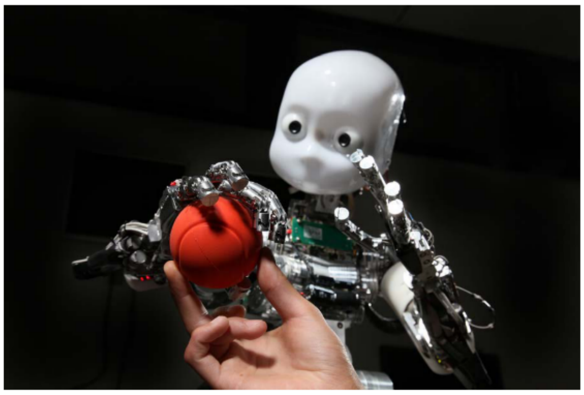
Directly related to the problem above, most robots tend to have one very specific goal in mind when they are implemented. In fact having such goal is a must with RL methods. These methods often work by defining a specific task to be completed, which in turn will define how the current model is evaluated. So if one wanted to implement a robot without a specific task in mind (or with more than one), using RL methods would most likely not be sufficient.

1.3 Thesis Objectives

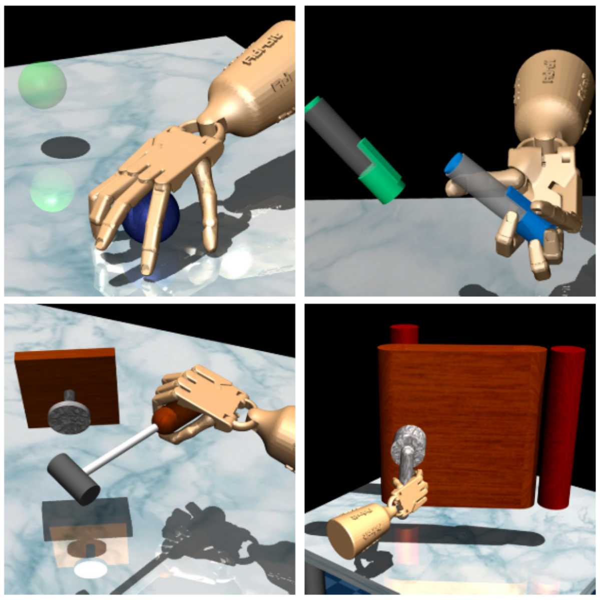
The motivation behind this thesis is to have a model that can manipulate any object and efficiently learn to perform any given task. As indicated before, using vision-based methods may lead to hard to solve problems, so we propose to design a model that will mostly rely on tactile information. Although intuitively one would argue that humans use vision to manipulate an object, such as a pen, we are still able to do most manipulation tasks with our eyes closed, solely relying in our sense of touch and prior knowledge. With vision, the information used to train the model is not related to the sense of touch, making it harder to learn strategies that adapt to new tasks, or even to new objects. An example of two applications, one using visual and another using tactile feedback, are presented in Figure 1.1. Implementing a manipulation model based on tactile information is most likely the next logical step when it comes to robot manipulation since it provides a way to manipulate objects that is much closer to how humans do it. With this, robots become more efficient in manipulation-like tasks, thus increasing their value to society. And although implementing robots while not following the laws of human nature has also been proven to work (e.g. using vision for manipulation tasks), the way humans act has always been a source of motivation behind robot development, which can lead to surprising results.

The work developed in robot manipulation with tactile data has only scratched the surface of what can be done with tactile sensors, but it was already possible to demonstrate its potential over vision [15]. In this thesis, it is indented to further explore different methods for robot manipulation. More specifically, there are two main goals described in this report.

- The first goal is to develop a model based on tactile feedback (rather than vision) and on the object's information (e.g. position and rotation), that is able to perform different tasks, including random exploration of the object itself.
- The second goal is to develop a model that can help the previous model in two ways: firstly by making the learning process of different tasks easier, based on previous learned tasks; and also to aid the initial model in making more informed decisions when manipulating a certain object.



(a) Example of a robot using vision to manipulate an object. Image taken from [15].



(b) Example of a robot using touch to manipulate an object. Image taken from [5] .

Figure 1.1: Different ways robots manipulate a set of objects.

It is important to note that both goals will be developed and tested in a simulated environment only.

1.4 Organization of the Document

This report is organized as follows: In Chapter 2, the state of the art is presented, along with an introduction of the most important concepts for this thesis. The methodologies and data used throughout the thesis are discussed in Chapter 3. In Chapter 4 the different experiments that were conducted throughout this work, and the respective results, will be discussed. Finally, in Chapter 5, a summary of the most important results achieved in this work will be presented as well as possible future work to be conducted.

2

Related Work and Background

Contents

In this Chapter, in-hand manipulation and Representation Learning state of the art works will be discussed, and also an introduction of the main concepts that will be used throughout this work. The first two sections will provide important theoretical background regarding the two main methods used to develop this thesis: RL and Representation Learning. The last section, will present the most important state of the art works for this thesis, each one presenting important information for the objectives defined before: one referring to manipulation using either vision or tactile based methods; and another presenting articles that approach the topic of Representation Learning. At the end, it is also presented a comparative analysis between the presented works, using key features for the development of this thesis.

2.1 Introduction to RL

In this section, it will be given an introductory level explanation of RL and how and why it is useful to help achieve the proposed objectives. RL is an area of ML where there is an agent that interacts with its environment to maximize a certain reward function. The purpose of a RL method is to learn a function that determines what action to take in a given state such that the total expected future reward is maximized. Such a function is called a policy, or in the best case scenario, an optimal policy. Achieving the optimal policy requires exploration and experimentation in a given environment. Figure 2.1 illustrates the relation between the agent and the environment.

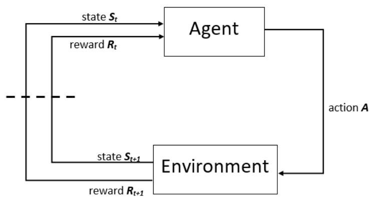


Figure 2.1: Basic operation of RL model

The agent will take a certain action, A, which will have consequences in the environment. The environment will change according to its current state and the action taken, and will create a new state and a reward according to it. The reward R_{t+1} is a measure of how good/bad the action A was, given the current state S_t and the next state S_{t+1}. A state is defined as a set of features that fully describes the environment so that the agent can take this information into account when choosing future actions. The ultimate goal of the agent is to maximize the reward signal, but for that, an adequate reward function must be defined. To illustrate this, let us consider the example of a child playing in a playground. At first, the child does not know what he can do or what is unsafe, so he starts by exploring the playground, randomly. The parents are constantly observing the child to make sure he doesn't hurt himself. After some time, the child learns what he is allowed to do (the good actions) and what he is not allowed to do (the bad actions), based on the feedback his parents gave him. This is how a RL algorithm learns. In this analogy, the child is the agent, the actions represent everything the child can do (play in the sand, go down the slide, etc.), the playground and the parents represent the environment and the reward function is defined by how the parents react to the child's actions. A RL algorithm learns by randomly exploring the environment until, by chance, it takes an action that yields a positive reward. Upon reaching such action, the model will then be more inclined to take that action since it knows that the reward will be higher and so on.

In more detail, most RL problems can be defined as a Markov Decision Process (MDP) defined by the tuple $(S, A, P, r, \rho_0, \gamma, T)$ where S is a set of states, A is a set of actions, $P : S \times A \times S \to \mathbb{R}_{\geq 0}$ is the transition probability distribution (that models the changes between states), $r : S \times A$ is the reward function, $\rho_0 : S \to \mathbb{R}_{\geq 0}$ is the initial state distribution, $\gamma \in [0, 1]$ and T is the horizon. The factor γ is the decaying factor which determines how much an updating step influences the current value of the weights of the networks. The horizon T defines how many steps into the "future" the agents has to take into account.

Most of the RL algorithms optimize a stochastic policy $\pi_\theta : S \times A \to \mathbb{R}_{\geq 0}$. Let $\eta(\pi)$ denote its expected discounted reward for horizon T, [16]:

$$\eta(\pi) = \mathbb{E}_\tau[\sum_{t=0}^{T} \gamma^t (r(s_t, a_t)], \tag{2.1}$$

where $\tau = (s_0, a_0, ...)$ denotes the whole trajectory, $s_0 \sim \rho_0(s_0)$, $a_t \sim \pi(a_t|s_t)$ and $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$. As a practical example, consider the problem of learning to navigate through a maze, like the one shown in Figure 2.2. The goal is to have an agent that is able to go from the START cell to the FINISH one,

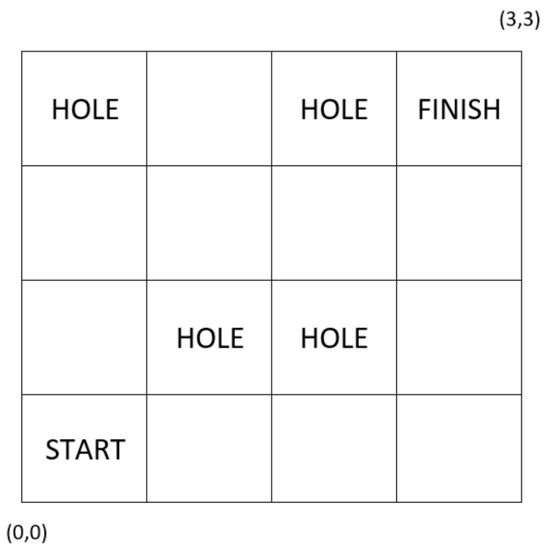


Figure 2.2: Maze that the RL agent needs to solve.

without falling into any of the HOLE cells. As an action, the agent could move ”up”, ”down”, ”left” or ”right” and the state is defined as the coordinate system (x_i, y_i) where $i \in \{0, 1, 2, 3\}$. The START is at (0,0) and the FINISH is at (3,3). It is also necessary to define a reward function that defines how the agent receives feedback from the environment. A more in depth explanation on reward functions will be given later, but for now let us consider that the agent receives -1 points for falling in a hole a +5 points for reaching the goal. Also, every time the agent falls in a hole, the position is reset to the START. The goal of the RL algorithm is to guide the agent in order to reach the goal and achieve the maximum points possible. At first, the algorithm will take random actions since it has no knowledge of the environment. But eventually, since this is a finite space (*i.e* there is a finite number of state-action pairs) the algorithm will learn to find what is the best possible action for every state. A state-action pair is defined as all the actions that the agent can take in a particular state. For example, the state (0,0) has as its possible actions ”up” or ”right” and the algorithm will learn to evaluate both these actions regarding their relation to the goal. This process is repeated for every state and by the end of the training process, the algorithm should have defined what is the best action to take in each state and will be able to reach the goal without falling.

The previous example is what is known as a discrete problem, since the number of states and actions are fixed and finite. But there are some problems where this assumption cannot be made. For example, if one was to develop a model to drive a car, the number of actions (and states) are infinite. In this case, the actions could consist of breaking, accelerating and turning. For each of these actions, there would be a magnitude part to it (*i.e.* the amount of acceleration or break), which could be any real value. As stated above, with a discrete observation space one could build a model that would learn to predict the value of each state-action pair, since these are finite, thus learning to take the best action for every state. But with a continuous observation space, this is not possible since there are an infinite number of state-action pairs. The problem of controlling a robotic hand to manipulate an object, is defined as a continuous problem, as it will be detailed in Chapter 3. In order to tackle these continuous problems, other algorithms were developed such as the REINFORCE [17]. This algorithm estimates the gradient of the expected reward $\nabla_\theta \eta(\pi_\theta)$ using the likelihood ratio trick [16]:

$$\widehat{\nabla_\theta \eta(\pi_\theta)} = \frac{1}{NT} \sum_{i=0}^{N} \sum_{t=0}^{T} \nabla_\theta log\pi(a_t^i | s_t^i; \theta)(R_t^i - b_t^i), \tag{2.2}$$

where $R_t^i = \sum_{t'=t}^{T} \gamma^{(t'-t)} r_t^i$ and b_t^i is a baseline that only depends on the state s_t^i. From this point, an ascent step is taken in the direction of the estimated gradient following the rule:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla_\pi(a_t^i | s_t^i; \theta)}{\pi(a_t^i | s_t^i; \theta)}, \tag{2.3}$$

where G_t is the total discounted return and α is the learning rate. The fact that in (2.3) the updates are inversely proportional to the probability of a particular action being taken, prevents frequently-occuring action from ”winning-out”, even if they do not provide the highest reward.

Other algorithms such as Natural Policy Gradient (NPG) or Relative Entropy Policy Search (REPS) improve upon REINFORCE in different ways. In the case of NPG, by computing an ascent direction that approximately ensures a small change in the policy distribution. In the case of REPS, by limiting the loss of information per iteration to ensure a smooth learning progress. Both NPG and REPS are part of the family of Deep RL algorithms, each of them having optimal results in certain type of environments. The ”Deep” part of the name in Deep RL algorithms just indicates that the way the algorithms learn is by mimicking the way a human brain works, *i.e.* by using neural networks. The work in [18] provides an extensive review of the state of the art Deep RL methods used in continuous control by comparing the performance of each of them in a set of given tasks. Another two very promising state of the art Deep RL algorithms are the Proximal Policy Optimization (PPO) [19] and Soft Actor-Critic (SAC) [20]. Both these algorithms were tested in this thesis, so an introduction of both will be given here.

2.1.1 PPO and SAC

PPO and SAC have been proven to work in different tasks and scenarios as demonstrated in [3] for PPO and in [21] for SAC. Both algorithms are *Actor-Critic* algorithms [22] which is a subset of Deep RL algorithms, that were designed to tackle the problem of having a continuous observation space. This class of algorithms, have two separate models: the *Actor* model which will output the desired action in the continuous space, taking into account the current state; and the *Critic* model which will evaluate how good an action is, given a certain state. The relation between each model and the environment can be seen in Figure 2.3.

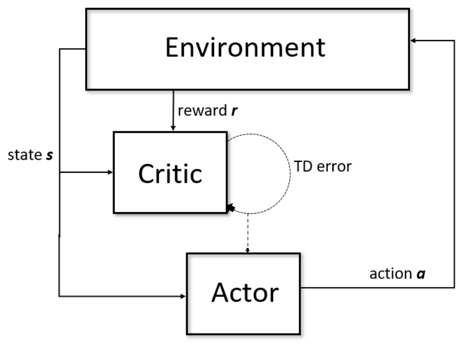


Figure 2.3: General architecture for Actor-Critic algorithms.

The *Actor* will take as input the state and output the action. The *Critic* will take as input the state and the reward obtained from executing the action that the *Actor* computed. These two inputs are then used to evaluate how good that action was, by reporting back the difference between the estimated reward at any given state or time step and the actual reward. This is represented by the ”TD error” in Figure 2.3 where TD stands for Temporal Difference (a class of methods that sample from the environment and perform updates based on current estimates) [23]. Resuming the previous analogy of the child playing in the playground, in this case the *Actor* model would be represented by the child, *i.e.* this model would have to learn to choose the actions yield the best results, and the *Critic* model would be represented by the parents, *i.e.* this model would have to learn to judge the child's action as ”good” or ”bad” actions.

There are two main differences between SAC and PPO and the first is related to the optimization function used in each one. SAC tries to minimize the following optimization function given by (2.4) [21].

$$\pi^* = \arg\max_{\pi} \sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t)] + \alpha\mathcal{H}(\pi(.|s_t))]. \tag{2.4}$$

Simply put, (2.4) optimizes two terms, one relative to the reward and one relative to the entropy of the system, where α controls the importance of the entropy term. This algorithm provides a way to encourage the model to explore more states instead of only visiting the states that give a better reward. In simpler words, the hyperparameter α allows the user to balance the exploration versus exploitation. Exploration means that a policy has a bigger probability of taking random actions, for the possibility of achieving better future rewards. Exploitation is the opposite, the policy is more likely to follow paths that have already been explored and are known to yield a certain reward. The balance between exploration and exploitation is a problem for many Deep RL algorithms. If one decides to only maximize the immediate reward, it removes the possibility of exploring new states that could yield a better future reward. The opposite is also true, if one mainly focuses on exploration, the model will not learn to maximize the reward, thus not learning to complete the task. In many RL methods, this is solved by a simple algorithm known as the Epsilon-Greedy algorithm. The algorithm is described as follows:

$$\text{Action at time } t = \begin{cases} \text{max } Q_t(a) & \text{with probability } 1-\epsilon \\ \text{random } a & \text{with probability } \epsilon \end{cases}, \tag{2.5}$$

where ϵ is an hyperparameter and $Q_t(a)$ represents the value of taking action a. The algorithm in (2.5) allows the user to balance between exploration and exploitation, but this can be further improved with a simple trick. By adding a decay to ϵ, *i.e.* by progressively decreasing the value of ϵ, it is possible to better guide the algorithm to a more optimal solution. At the start, ϵ is at its highest value, since we want the algorithm to explore the most, in order to find as many options as possible. Towards the end, ϵ gets smaller, ensuring that the algorithm will take the actions that yield the best value, since it is assumed that a significant amount of exploration was made before. Thus, it is safer to follow a specific path, knowing that that path will lead to an optimal or sub-optimal solution. The fact that SAC has this mechanism built into the optimization function makes it a good candidate in order to achieve the proposed objectives, as it will be described in Chapter 3.

To understand the optimization function behind PPO, it is first necessary to explain the Trust Region Policy Optimization (TRPO) algorithm [24], since the PPO was built upon it. The loss function of TRPO is given by (2.6), [19].

$$L(\theta) = \hat{\mathbb{E}}\big[r_t(\theta)\hat{A}_t\big], \tag{2.6}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta\,old}(a_t|s_t)}$, which is also known as the probability ratio, *i.e.* the likelihood of the chance that a certain event will occur as a result of a random experiment. $\hat{A}_t$ is an estimator of the advantage function at timestep t. The advantage function is a measure of how much is a certain action a good or bad decision given a certain state. The improvement of PPO relative to TRPO can be described by (2.7), [19].

$$L(\theta) = \hat{\mathbb{E}}\big[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\big], \tag{2.7}$$

where ϵ is a hyperparameter. In (2.7), the expected reward is computed with a min function between two terms. The first term is the term given by the TRPO in (2.6) and the second term is the actual contribution of the PPO algorithm. This second term clips the probability ratio, $r_t(\theta)$, which motivates the policy to be in the interval $[1-\epsilon, 1+\epsilon]$. Then, the min function is applied, which means that the if the change in the probability ratio (2.7) increase, then the change is ignored, otherwise the change is included. Since this is an optimization problem, the objective is to minimize the cost function, in this case given by (2.7) and that is why, if a certain change leads to an increase in the cost function, it is discarded, and otherwise accepted.

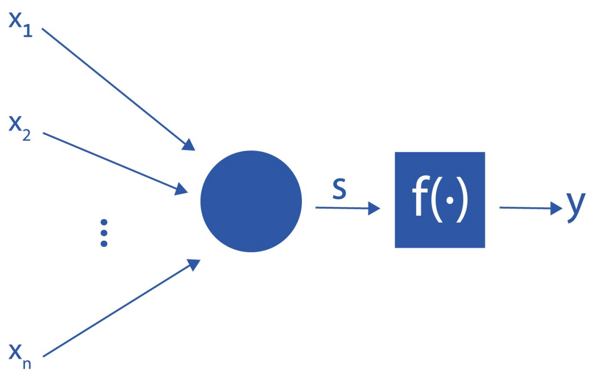
The other difference between PPO is and SAC is that PPO is an on-policy algorithm, while SAC is an off-policy algorithm. The difference between on-policy and off-policy methods is that on-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data.

The reason as to why these two algorithms were chosen will be further described in Chapter 3.

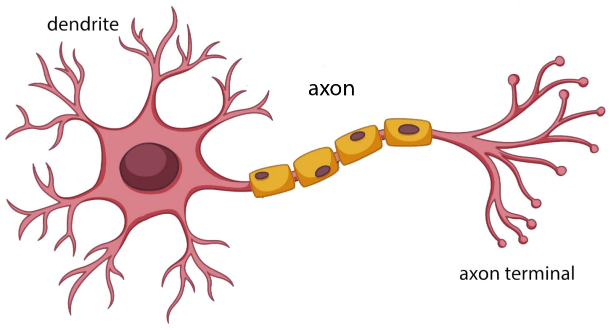
2.2 Representation Learning

Representation Learning is a concept where the objective is to extract useful, unlabeled information from the input data in order to make it easier to use it when building classifiers or other predictors. The goal is to train a model that will learn what are the most important features of a given input, to later use that information. Consider the example of applying Representation Learning to images of dogs and cats. The ultimate goal is to have a model that is capable of classifying the images in either dogs or cats. The standard approach would be to use a Supervised Learning method, a subset of ML, where it is available an input and output set. The input would consist of a set of images of cats and dogs and the output would contain the correspondent labels of those images. With this, it would be straightforward to train a model capable of labeling images. Now consider that, instead of using the input signal as it is, we would use a representation of it, that would only contain the most important features of the input. By using this representation, the model would learn faster and with more precision, since the representation will only contain information relevant for the task of labeling the images (the original images could contain extra information, such as background objects). Usually, this representation is in a latent space, *i.e.* it is represented with a lower dimension than the original one. There are several ways one could achieve a representation of the input, but in this thesis it was only used one: the Autoencoder (AE) [25].

An AE is a type of neural network and to understand how an AE works, one must first understand how a neural network works. As the name implies, a neural network is a set of connected units (artificial neurons) that mimic the process of how information travels in the human brain. A neuron, which is represented in Figure 2.4(b), has 3 main parts: the dentride, which is the part responsible for receiving the input, the axon, which will process the input collected, and finally the axon terminal, which will send the processed information to its connected neurons. In a very similar way, an artificial neuron is composed by 3 main parts: the blue circle in Figure 2.4(a) is responsible for collection the data, to which is then applied some function (also know as the activation function), whose output will be transmitted to other neurons.



(a) Neuron representation in ML.



(b) Representation of a human neuron.

Figure 2.4: On the right, a representation of a human neuron; on the left, a representation of the neuron in ML.

A neural network is composed by layers of units (as the one represented in 2.4(a)). Each layer has a finite number of units and each neural network has a certain number of layers, depending on its purpose. Generally speaking, a neural network, takes an input, passes it through its layers, until an output is produced.

In Supervised Learning, two sets of data are available to train a neural network: the input, X, and the expected/true output, Y. In order to ”teach” a neural network to perform a specific task, such as labeling images, the general idea is that an entry of the input data, x is passed through the network until it produces a certain output $\hat{y}$. Then that output will be compared to the true output y (the corresponding entry from the output data), and finally the network parameters are updated based on the difference between $\hat{y}$ and y. The network parameters are the weights that connect two neurons in consecutive layers, for example, in Figure 2.4(a) the weight of each input x_i where $i \in \{1, 2, ..., n\}$ can start as 1 (every input has the same importance), and as the model is training, the weights will be updated. Taking the simple example of labeling images, the untrained model will most likely output an incorrect and random label at first, but as the training process proceeds, the model will learn to not make the same mistakes, since it has access to the true label of each image, and the accuracy of the model will improve. By using the back propagation algorithm [26], it is possible to propagate the error (difference between $\hat{y}$ and y) from the output layers to the hidden layers, and update the network parameters using the gradient descent method [27].

The example above was just one example of an application of neural networks, but they can be used in many different types of applications, ranging from classifying images, such as the AlexNet [28], which is capable of labeling images from 1000 different labels, or to remove noise from images, as is one of the main applications of AEs. An AE is a specific type of neural networks, since its architecture *i.e.* number of layers and number of units per layer, follow a very specific pattern. An example of such architecture is depicted in Figure 2.5.

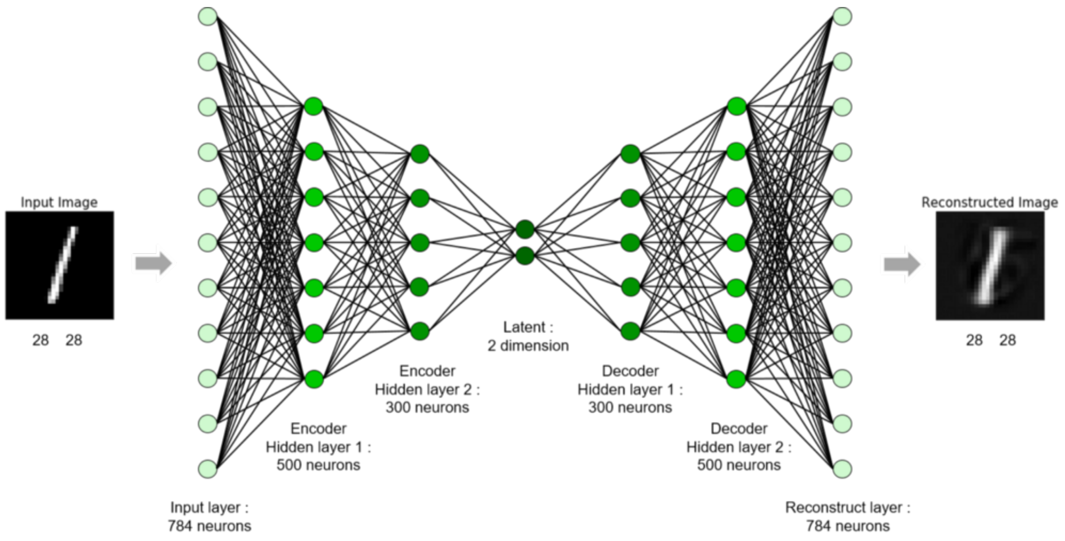

Input Image
28 28
Input layer : 784 neurons
Encoder Hidden layer 1 : 500 neurons
Encoder Hidden layer 2 : 300 neurons
Latent : 2 dimension
Decoder Hidden layer 1 : 300 neurons
Decoder Hidden layer 2 : 500 neurons
Reconstruct layer : 784 neurons
Reconstructed Image
28 28

Figure 2.5: Architecture of an example autoencoder. Image taken from [1].

As it can be seen in Figure 2.5, the AE is a simple neural network but with the particularity that the middle layer, also known as the bottleneck, has the fewest number of units. The objective of the AE is to learn a lower dimension representation of the input, *i.e.*. From Figure 2.5, it is possible to observe that the size of the input (first) layer is 784, while the number of units in the bottleneck is 2. This process of lowering the number of units in each layer, to then increase it back up when moving from the bottleneck to the output layer, will allow the model to learn how to represent the input, using less dimensions. To further explore this and to understand how an AE works, it is useful to separate it into two different networks: the encoder and decoder networks. The encoder part comprises everything from the input layer to the bottleneck. As the name states, the encoder network is responsible for encoding the input into a lower dimension representation. The encoder function is given by:

$$\phi : X \to \mathcal{F}, \tag{2.8}$$

where X is the original data and $\mathcal{F}$ is the latent space (the lower dimension representation). Essentially, the encoder function maps the input space to this latent space by progressively decreasing the number of units in each layer. Applying the same mechanism to the decoder network, the decoder function can be written as

$$\psi : \mathcal{F} \to Y, \tag{2.9}$$

where Y is the output space. The decoder network has the opposite purpose compared to the encoder, it decodes the latent space representation to the output space. So, by combining (2.8) and (2.9), the objective of the AE is to learn the non-linear transformations ϕ, ψ such that the output data is as close as possible to the input data. In other words:

$$\phi, \psi = \underset{\phi,\psi}{\arg\min} ||X - Y||^2. \tag{2.10}$$

Usually with AEs, the output space is the same as the input space, *i.e.* $X \equiv Y$, so that the model can learn a representation of the input and from that representation, learn how to reconstruct the input, with as little error as possible, but there are some cases, like the denoising AE, where this is not observed.

One may ask why is it relevant to have a model whose output is the same as its input, and in fact that is not the main feature of the AE. Its main purpose, is to learn a latent representation of the input. By forcing the input to decrease its dimension, the AE is essentially learning to represent the same input, but with less dimensions, only maintaining the most important features necessary to reconstruct the output. The fact that the output data is the same as the input data, is just necessary for the model to learn the representation. For example, the AE could be used to reduce the size of images by learning to represent them with less pixels, or it could be used, as indicated before to denoise images. Let's further explore the denoising AE. Consider the two images represented in Figure 2.6.

In Figure 2.6(a), it is shown an image with 28x28 pixels (each pixel has a value between 0 and 1) of a handwritten digit, in this case the number 7. The image on Figure 2.6(b) represents the same digit but with noise added to it (gaussian noise with 0.4 mean and 0.2 standard deviation). In order to design a denoising AE, one could define the input X as a set of images like the one in Figure 2.6(b), and

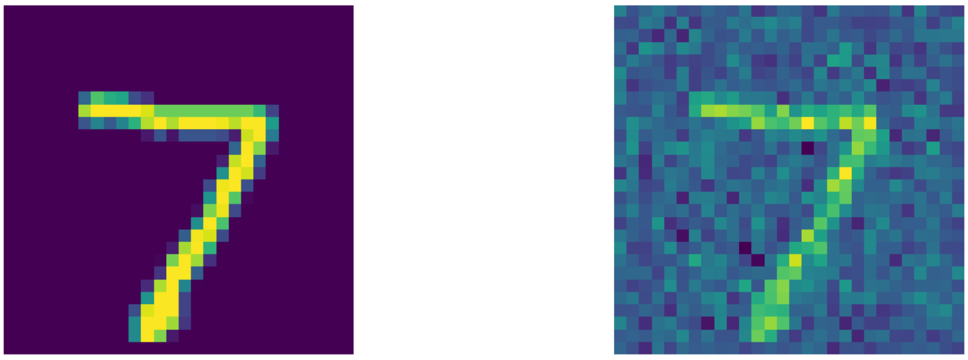

(a) Image of a handwritten digit from the MNIST dataset. **(b)** The same image as on the left but with added noise.

Figure 2.6: On the left an example image from the MNIST dataset [2]; on the right, the same image but with added noise

the output Y as a set of images like the one in Figure 2.6(a). Then, by applying Supervised Learning methods, the AE would learn to generate the output images, from the input images, *i.e.* to remove the noise from the input images.

The ability of an AE to condense information and to only map the most important features of the input, allows for models to be trained in a different way. More specifically, it can be of use for problems where the input has a high dimension, since it allows us to reduce that dimension, which is the case of the problem presented in this thesis. For this reason, the AE was chosen as a staring point for learning to represent the input, as it will be described in Chapter 3.

2.3 Related Work

2.3.1 In-hand Manipulation

One of the most recent works in this field is [3], in which the authors developed a model capable of manipulating a cube to a specified pose, using vision, as shown in Figure 2.7. Since there was a specific task to be performed, a RL method was used, and in order to maximize the reward function, they used a Long Short-Term Memory (LSTM) [29] network which was trained with PPO [19]. The training of the model was first conducted in a simulated environment and then transposed to the real environment. A key point in this work that allowed for the model to have a good performance on the real environment, were the randomizations (e.g. physical properties of the object, added noise) introduced in the training period.

Although the work presented in [3] achieved state of the art results using vision, these randomizations lead to an increase on the training time of 33 times the time it took to train a model without randomizations (1.5 hours to 50 hours in real time). So, in order to achieve similar results, but with a lower training time, tactile data could be used [30].

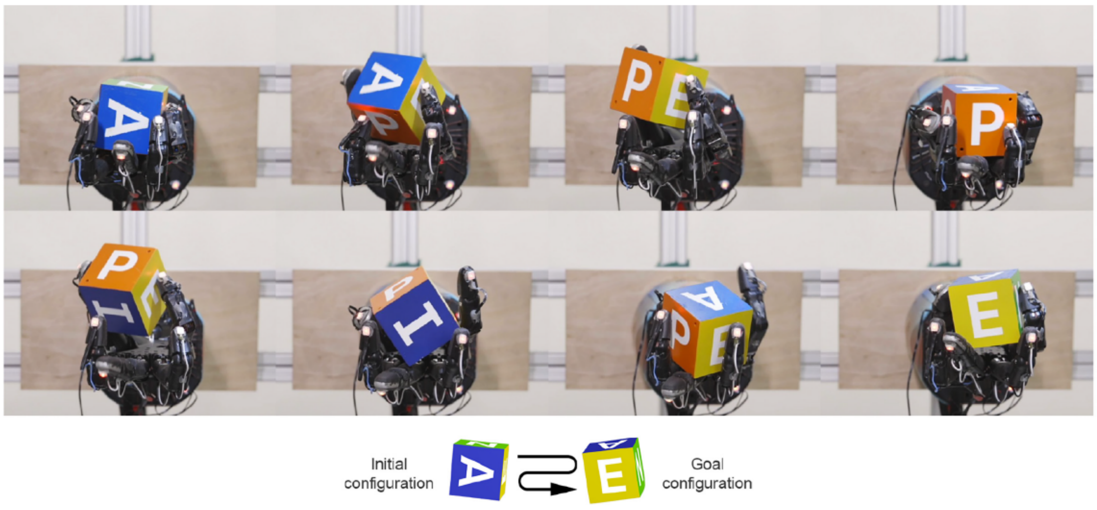


Figure 2.7: Example sequence of the real robot manipulating the object to the desired pose. Image taken from [3].

There are some works which focus on simpler tasks such as [31], while using tactile data. The objective in that work was to move several cylindrical objects from one point to another using a two fingered hand. By using an algorithm previously proposed from the same authors, Non-parametric Relative Entropy Policy Search (NPREPS) [32], the final model was able to manipulate novel objects with minimal updates to the policy. Y. Chebotar *et al.* [4] also add to this aspect in the sense that the model is able to adapt to the changing environment. In that work, the model learned was trained with Dynamic Motor Primitives (DMP) [33] and REPS [34] algorithms, in order to perform a scraping action, while using a robotic arm grabbing a spatula. The robot used and the experiment carried out are represented in Figure 2.8. Although it was able to overcome the changes in the environment, the method used to obtain tactile data is not feasible for standard, human-like robotic hands. In this case, the tactile information was obtained from two tactile matrices of 16 cm x 16 cm attached to the robot, while in this thesis, it is intended to use smaller tactile sensors.

There are also other works that include human demonstrations in the training process, in order to reduce the training time and also, to achieve a more human-like manipulation. A. Rajeswaran *et al.* [5] focused on having a robotic hand that is capable of performing every-day tasks such as opening a door, manipulating a tool, etc. The authors propose a new approach, the Demo Augmented Policy Gradient (DAPG) algorithm, in order to reduce training time. The results showed that this new method was up to 30 times faster than learning from scratch, in some tasks. Figure 2.9 shows the 4 tasks

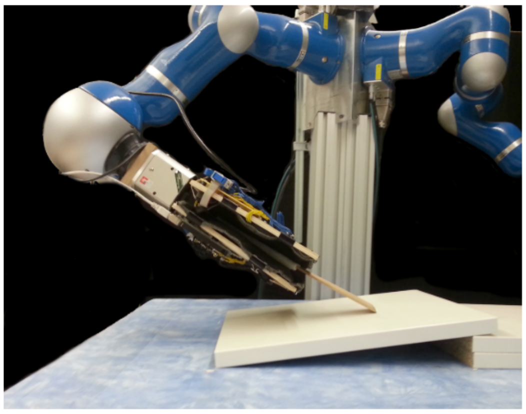


Figure 2.8: Robot performing a gentle scraping task using tactile feedback. Image taken from [4].

trained in that work. Although using human demonstrations shows promising results, it also adds another constraint since additional hardware is required to record the demonstrations, such as a cyberglove.

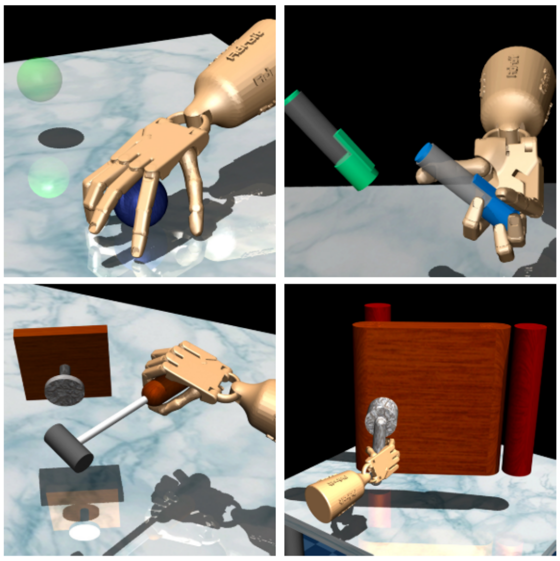


Figure 2.9: Learned tasks in a simulated environment. Top-left: object relocation; top-right: in-hand manipulation; bottom-left: tool use; bottom-right: door opening. Image taken from [5].

Another approach in achieving in-hand manipulation, is to be able to predict tactile slip [6]. While using Supervised Learning methods, namely Support Vector Machine (SVM) [35] and Decision Forest (DF) [36], the authors developed a method capable of detecting if the object is slipping, and in case it is, to apply the necessary forces to counteract. The robot used and the experiment carried out are represented in Figure 2.10. Although is was successful in a real environment application, the hardware used was a single sensor and the object was not being manipulated, which is not the goal of this thesis.

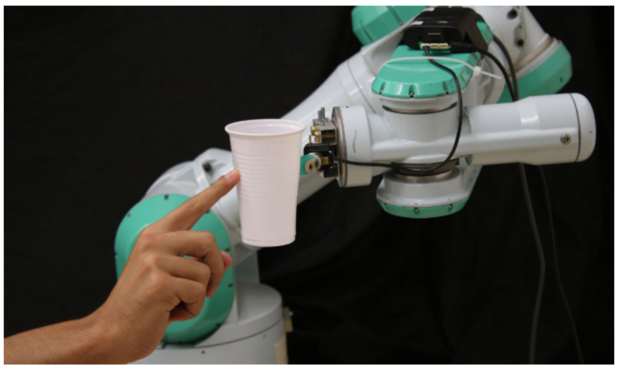


Figure 2.10: A human-robot grip stabilization experiment where a human and a robot collaborate in order to preserve a stable grip on a deformable plastic cup. Image taken from [6].

2.3.2 Representation Learning

Representation Learning became a topic of interest since it provides a way to characterize data in a more efficient way [13]. There are multiple approaches to tackle this problem and using an AE is a common approach. Z. Zhu *et al.* [37] use this method, initializing the AE with a Deep Belief Network (DBN) [38], to compute a 3D shape of an object from multiple 2D images of that same object.

A more manipulation-oriented work was presented in [39]. By developing a RL algorithm that leverages representations learned from an AE, they were able to model a 5 DOF robot to manipulate a pole to a given orientation, while using visual data.

Although not directly related to Representation Learning, Z. Yi *et al.* [40] present a way to represent an unknown object. Using a Gaussian Process, they are able to efficiently explore an unknown object surface in order to reconstruct the full 3D surface.

M. C. Gemici *et al.* [41] propose a learning algorithm that takes as input tactile sensory signals of a robot obtained while performing actions on the objects (which were food in this cause), to then extract useful features from these signals and map them to physical properties of the objects. They also perform unsupervised and clustering based on Diritchlet Process (DP) to represent the physical properties compactly, from the previous mapping.

Z. Wang [42] uses Representation Learning to infer important object properties, such as its pose, from tactile data. The learned representation is used to train a control policy which is then transferred to a real robot. They used a LSTM to learn the tactile representation and then apply supervised learning to map the desired pose to the representation.

Another approach is using Predictive State Representation (PSR) [43]. The work presented in [44] extends PSRs to perform planning in-hand manipulation tasks, using tactile data.

2.3.3 Literature Analysis

In Table 2.1, some of the works presented above, are compared in terms of some of the key features identified with the work to be developed, such as: the number of DOF in the robot used; if the training/testing time was done with the help of visual or tactile information; what types and methods were used to learn the model; and if the final model was tested on a real environment or not.

Table 2.1: Literature analysis of the main state of the art works.

Work / Date	DOF	Vision / Tactile sensors	Type of learning	Tested on a real environment
[3] - 2020	24 (4 passive)	Vision	RL (with PPO algorithm)	YES
[31] - 2015	4	Tactile	RL (with NPREPS algorithm)	YES
[4] - 2014	7	Tactile	DMPs + RL (with REPS algorithm)	YES
[5] - 2017	24	Tactile	RL (with DAPG algorithm)	NO
[6] - 2018	1	Tactile	Supervised Learning (with SVM and DF)	YES
[37] - 2016	N/A	Vision	Deep Learning (with DBN + AE)	YES
[39] - 2016	5	Both	RL (with AE)	YES
[42] - 2018	5	Tactile	Supervised Learning + RL (with LSTM)	YES
[41] - 2014	7	Tactile	Supervised Learning + DP	YES
[44] - 2015	2	Tactile	PSR	YES

In conclusion, in-hand manipulation has been a topic of interest for the past few years and there are many possibilities for future directions. Although the works presented in the above sections are successful in what was desired for each work, most of them face some difficulties/limitations. For example O. M. Andrychowicz *et al.* [3] relied on intensive simulation, data and model augmentation to perform a single task and since it uses vision, the model won't most likely adapt to similar objects, but with different weights or friction coefficients. A. Rajeswaran *et al.* [5] relied on human demonstrations to speed up the training process and to achieve the desired behaviour (thus needing extra hardware), while only testing the model in simulation. Y. Chebotar *et al.* [4] were able to adapt to the changing environment and to test the model on a real robot, but the tactile sensors used are not feasible for a real-life application since they are too big (16 cm x 16 cm). F. Veiga *et al.* [6] managed to achieve grip stabilization on different objects but a single sensor was used to do so, and in order to train the model, the training data had to be labeled before hand.

3 Methodologies

Contents

In the previous Chapters, it was discussed how tactile feedback can be used to train a model capable of manipulating objects. Additionally, its benefits over vision-based approaches where highlighted. In this Chapter, it will be shown how such information can be used in order to accomplish the proposed objectives. Recapping the objectives of this thesis, one goal is to develop a model capable of in-hand manipulation. And a second goal is to verify if, by using a model that uses the latent representation of the input, instead of the actual input, the performance and learning process of new tasks will be improved. So, in the first section, an overview of the overall architecture will be given. The following section will then describe what computational tools are going to be used. The next section will approach the problem of the high number of base dimensions on the action vector. The next two sections are going to describe in more detail what methods and models are going to be used in order to solve the proposed goals. Finally, the last section is going to give an overview of the experiments conducted.

3.1 Architecture Overview

In this section, an overview of the main features and architecture proposed for this thesis will be given. As mentioned before, an important objective is to develop a model capable of in-hand manipulation tasks, in a simulated environment. Both the environment and the tasks are going to be described in Section 3.2 and Section 3.4, respectively. The first approach is to use the aforementioned Deep RL methods in their standard form, to train a model capable of learning a specific task. The general architecture of these methods is presented in Figure 3.1.

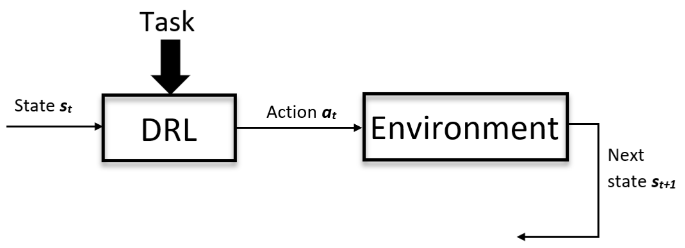


Figure 3.1: General operation of a Deep RL method.

This model would take as input a state, which will consist of tactile feedback from the touch sensors, pose (position and orientation) and velocity of the object, joint amplitudes and velocities, and will output a certain action. This action will control the movement of the joints in a humanoid hand. The given action will cause a certain change in the environment, thus creating a new state, which will once again be fed into the Deep RL model, and so on. Although this approach by itself can lead to good results, it also leads to a few problems, which may influence the learning process and overall performance of the model. Those problems are:

- **the high number of DOF:** as mentioned in Chapter 1, the human hand has about 27 DOF [14]. Although the simulated model that was used to develop this thesis had a base of 20 DOF instead of 27, this high number of DOF could make the learning process relatively slow, possibly reaching the point of not learning at all. This is due to the fact that Deep RL methods, initially tend to explore the action space in order to access the relations between actions and futures states but with a high number of DOF for the action, this process becomes much harder.

- **nonexistent generalization for different objects:** it is intended with this work that the developed model is capable of generalizing a certain task to several objects, different from the one it was trained with. This poses a problem because most Deep RL algorithms tend to focus the training process on improving the performance in a particular environment. For instance, if the model was trained to manipulate a cube and, after training, this object was switched to a pen, the model would have difficulties in adapting to the new object, since they have different shapes and properties. Even if the new object was introduced during the training phase, it would be difficult for the algorithm to learn how to generalize the knowledge in order to successfully manipulate both objects.

- **nonexistent generalization for different tasks:** similarly to the aforementioned issue, most Deep RL methods, when used in their purest form, fail to generalize to multiple tasks. This is directly related to how these methods learn. As explained in Chapter 2, RL methods learn by maximizing a given reward function. There will be a more in depth explanation about this topic in Section 3.4, but this reward function needs to be directly related to the task to be learned. If the objective is to have a controller that can perform multiple tasks, it would imply that there would have to be a different Deep RL model for each task, or if only one was used, it would need to be updated every time the task was changed.

The problem of the high dimensional action space can be solved by reducing the number of actions the model can take, while maintaining the ability to manipulate objects (using synergies). This will be addressed in Section 3.3. Regarding the remaining problems, their solution is not so trivial, therefore, another approach must be considered. The main concept of this new approach is to have the Deep RL methods learn over a reduced feature set that only contains the relevant information for manipulating an object. The overall architecture for this new approach is presented in Figure 3.2.

The difference between the proposed architecture and the one presented in Figure 3.1 is that now the input (state) is first passed through another model, the Forward Model, which will produce two outputs. The most important one here is the learned representation $\mathcal{F}_t$, which will comprise only the most important features of the state that are relevant for the task at hand. This Forward Model will have an AE-like architecture and it should be trained with a rich dataset (multiple objects and different tasks).

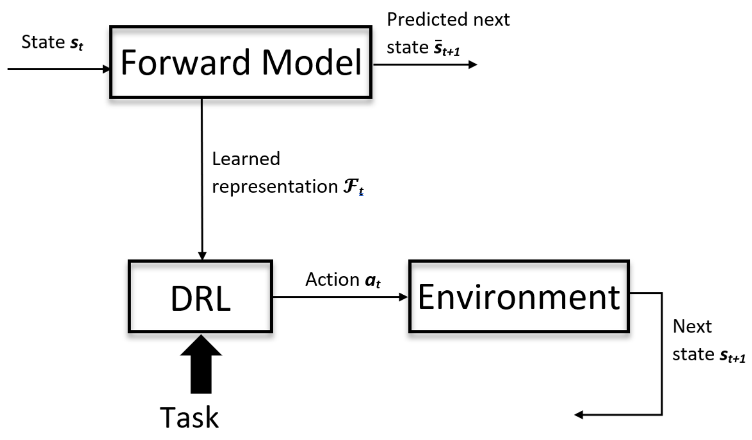

State s_t
Forward Model
Predicted next state $\bar{s}_{t+1}$
Learned representation $\mathcal{F}_t$
DRL
Action a_t
Environment
Next state s_{t+1}
Task

Figure 3.2: Updated architecture intended to generalize the inputs given to the model.

Its input will be the state and the output will be a prediction of the next state given a certain action. Since this prediction requires an action but that action is only available after the Deep RL model computes it, the predicted state will not be used for training of the Deep RL algorithm, but only for training of the Forward Model (more on this topic in Section 3.5). The training set for this model should comprise data obtained while manipulating different objects and with different tasks. This way, the Forward Model will be able to represent the state, regardless of the type of object or task. Moreover, the Forward Model will be able to ”filter” the information contained in the state vector and feed that pre-processed information to the Deep RL model. This new input could allow for:

- **a faster training process:** since the Forward Model follows an AE-like structure and the learned representation will be at the bottleneck, this signal will have a considerably lower dimension space, making it easier for the Deep RL algorithm to learn.

- **robustness to different objects:** the data collected to train the Forward Model was obtained from manipulating different objects. Therefore, no matter which object is being currently manipulated, the Forward Model will still be able to provide a representation that is invariant to the object.

- **possible generalization to different tasks:** for the same reason as above, the Forward Model will be able to generalize to different tasks, making the input of the Deep RL model (the signal $\mathcal{F}_t$), invariant of the task.

The following sections will provide more detail on the simulator and learning frameworks used and also on how the Controller and Forward Model were obtained.

3.2 Computational Tools

As mentioned before, this work will be fully developed and tested in a simulated environment, and for that, there are two main software applications that are needed: a learning framework and a simulator. This section will provide an extensive review on each software, in order to conclude what were the chosen softwares and why.

3.2.1 Simulator

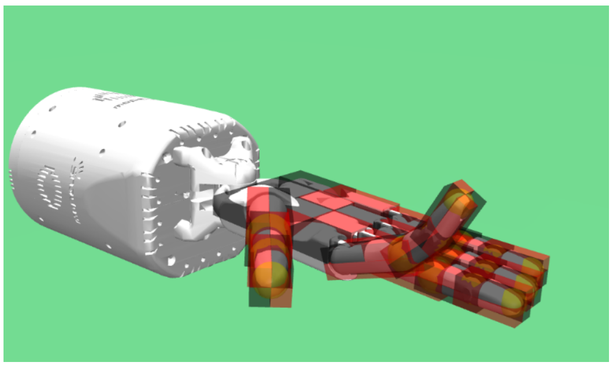
Deep Learning (DL) methods usually need a large amount of both data and time to learn a model, so a good approach is to train a certain model in a simulator first, where it is possible to train faster and there is no risk to damage a physical robot. After the model is trained, theoretically, it is possible to transfer it to the real robot to further test the model, although that will not be part of this work. So, in order to have a way to develop the methods and test them, a simulator is required. There are many options when it comes to simulating robotic environments and some articles (such as [45] and [8]) already made a revision on the most commonly used simulators. In [45], the authors compared the simulators Bullet, Havok, MuJoCo [46], ODE, and PhysX based on their performance on four different scenarios, including grasp stability and, when the engine is pushed to the limit, by evaluating the maximum time step at which the engine still works. They concluded that there is no particular engine that is better than the rest in an uniform way, each engine is good in particular tasks and environments. However, it was also observed that MuJoCo was both the fastest and the most accurate on constrained systems relevant to robotics, and was capable of stable grasping at a much larger time step.

In [8], it is presented a user feedback-based review of the most commonly used engines. Table 3.1 presents a table adapted from [8] showing the user satisfaction ratings for each engine (not all), in the different categories.

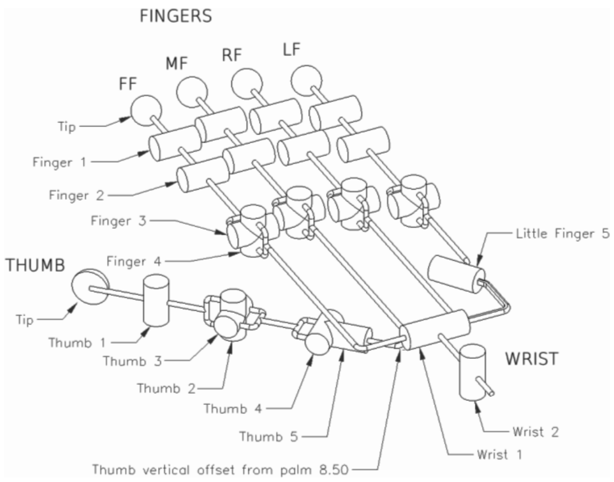
Table 3.1: Ratings for the level of user satisfaction of the most diffused tools. Table adapted from [8].

Engine	Documentation	Support	Installation	Tutorials	Advanced use	API	Global
ODE	3.80 ± 0.63	3.40 ± 1.07	4.10 ± 1.28	3.20 ± 1.13	3.90 ± 1.37	3.40 ± 1.26	3.59 ± 1.15
Bullet	3.37 ± 1.06	3.62 ± 0.91	4.75 ± 0.46	4.00 ± 0.76	3.75 ± 0.71	3.87 ± 0.83	3.96 ± 0.78
V-Rep	4.28 ± 0.76	4.43 ± 0.79	4.71 ± 0.76	4.14 ± 0.90	4.28 ± 0.76	4.14 ± 1.07	4.25 ± 0.80
MuJoCo	2.33 ± 1.15	1.67 ± 0.58	4.33 ± 1.15	3.33 ± 1.15	4.67 ± 0.57	5.00 ± 0.00	3.62 ± 0.66
XDE	1.40 ± 0.55	2.80 ± 1.09	3.60 ± 0.55	2.80 ± 1.09	3.40 ± 1.10	3.00 ± 0.00	2.83 ± 1.07

As observed from Table 3.1, the V-Rep engine [47] was the one rated highest in most categories while XDE was the engine rated lowest in most categories in terms of user satisfaction. From [45], MuJoCo is the most suitable engine for the work to be developed in this thesis but from Table 3.1, V-Rep appears to be the user's preferred engine. However, OpenAI created an interface called OpenAI Gym [48] that allows to easily implement RL methods in their environments and disposes of premade models to be used in the MuJoCo engine. In particular, the OpenAI Gym framework includes a simulated model of the Shadow Dexterous Hand [49] shown in Figure 3.3(a) which is a viable model to use to train the models proposed. For this reason, the MuJoCo engine was chosen.



(a) Simulated model of the Shadow Hand.



(b) Actuators, represented by the cylinders, present in the Shadow Hand. Image taken from [50].

Figure 3.3: On the left, the simulated model of the Shadow Hand, provided by the OpenAI Gym framework; on the left, the location of all the actuator in the Shadow Hand.

As it can be seen from Figure 3.3(b), the hand has 24 DOF with 4 being passive while 20 are actuators (*i.e.* the motors that trigger movement in each individual joint). Each actuator is represented by a cylinder, where its axis (the axis along its biggest dimension) indicates how that actuator moves. In Figure 3.3(b) there are 24 cylinders in total, each representing a different actuator, but as mentioned before, the model of the Shadow Hand only possesses 20 active actuators. The remaining 4 are passive actuators, which means that they only move when their ”parent” actuator moves. They are indicated by the arrow labeled ”Finger 1” in Figure 3.3(b) and they represent the distal interphalangeal joint on a human hand. This is very similar to how a human hand moves, it is very hard to solely bend the middle of the finger without bending its tip.

Another key feature of the simulated model of the Shadow Hand, is that the hand has 92 built-in tactile sensors, which makes this model very suitable for the proposed goals. It is possible to observe the embedded tactile sensors, which are represented by the black and red rectangles and also by the yellow spheres in Figure 3.3(a). These simulated tactile sensors provide readings of pressure applied to the sensor and will be used to provide feedback to the Deep RL algorithm.

It is also important to define how a certain action will control the model. The action is a vector of values where each component will control a particular actuator from the ones shown in Figure 3.3(b).

An action can be defined as such:

$$A = [a_0, a_1, \ldots, a_n], \tag{3.1}$$

where $n + 1$ is the dimension of the action vector, which by default is the total number of actuators (20) and where $a_i \in [-1, 1]$.

The MuJoCo simulator also allows for the user to choose the sampling rate of the simulation. The simulated model of the Shadow Hand already has a predefined time step of 0.04 seconds, which means that every 0.04 seconds, the simulation takes a step, *i.e.* the environment is updated given a certain action. Another important factor when choosing the right simulator is what kind of data it is possible to extract from the simulator. It was already shown that with MuJoCo, it is possible to obtain touch sensor data from the hand, which is a key component of this work. Additionally, there are other information that are going to be used to train the model, which are available within the simulator. A summary of all the information is provided below:

- touch sensors values (92 values)
- object's position (3 values)
- object's linear velocity (3 values)
- object's orientation (4 values)
- object's angular velocity (4 values)
- joint amplitudes (20 values)
- joint velocities (20 values)

The touch sensor values are the ones previously presented in Figure 3.3(a) where each sensor detects a measure of pressure (it is also possible to make the sensors only show boolean values, if the object is touching the sensor outputs 1, else it outputs 0 but that was not explored in this work). The object position is measured in meters, the orientation is measured in radians and the velocities are measured in meters per iteration (linear) and radians per iteration (angular). Lastly, the information about the state of the hand is also included, namely the amplitude of each joint, in radians, and the angular velocity of each joint, again in radians per iteration. With this information, one is able to fully define the state of the environment and it is with this information that the model will try to learn the optimal policy. This is one of the reasons why Deep RL methods take a long time to learn, because the model needs to learn how each particular value is influenced by each action.

Finally, the OpenAI Gym framework also made available 3 different objects, presented in Figure 3.4 which will be important when testing the ability of the model to generalize to new objects. These objects

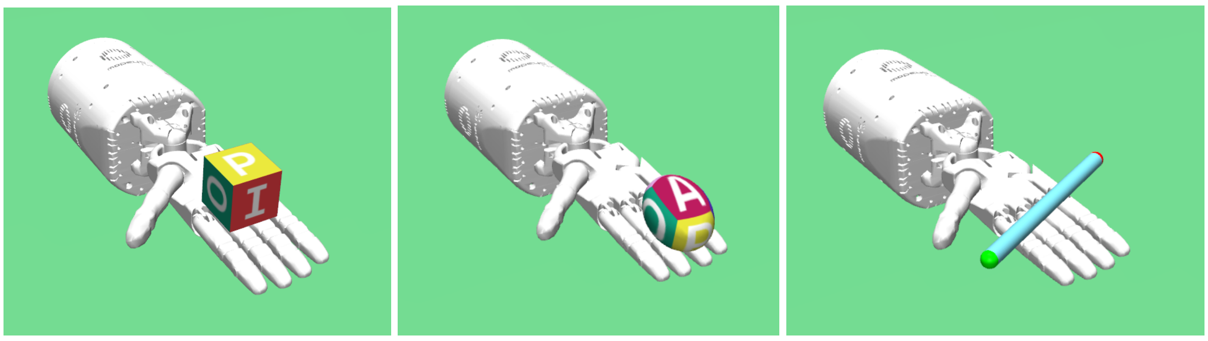

(a) Cube **(b)** Egg **(c)** Pen

Figure 3.4: Objects to be tested on the different models.

will be useful when testing the trained model given different objects and to also gather more diverse data for the Forward Model.

3.2.2 Learning Framework

As established in Chapters 1 and 2, RL and especially Deep RL methods, such as the PPO, have proven to be useful in achieving models capable of precise in-hand manipulation. Luckily, most state of the art algorithms have online implementations that everyone can use to train their custom models. For that, a learning framework that offers the necessary tools to train a custom model (and more), is necessary. Table 3.2 presents a comparison of the main learning frameworks used in state of the art works. This comparison takes into account key categories for this work.

Table 3.2: Learning frameworks comparison. Table adapted from [9].

Software	Open source	Interface	CUDA support	Automatic differentiation	Actively developed
Caffe [51]	YES	Python, MATLAB, C++	YES	YES	NO
Keras [52]	YES	Python, R	YES	YES	YES
MATLAB + Deep Learning Toolbox	NO	MATLAB	Train with Parallel Computing Toolbox and generate CUDA code with GPU Coder	YES	YES
PyTorch [53]	YES	Python, C++	YES	YES	YES
TensorFlow [54]	YES	Python (Keras), C/C++, Java, Go, JavaScript, R, Julia, Swift	YES	YES	YES
Theano [55]	YES	Python (Keras)	YES	YES	NO

As it is possible to observe from Table 3.2, there are several choices, which have little to no difference in between them (in terms of the features presented in Table 3.2). Two of the most important features are, if the framework has automatic differentiation, and if it has CUDA (Compute Unified Device Architecture) support. Automatic differentiation is necessary since it enables the user to freely build a neural network, knowing that the software will automatically compute the derivatives necessary to be used by the back propagation algorithm. This is important since it is a key component of training a model and without automatic differentiation, one would have to implement the algorithm from scratch. CUDA support is also important since it allows for a faster training process, making use of the computer's GPU. In Deep RL algorithms, this is useful since usually the training process of a model takes more time than normal ML algorithms due to the larger training data. As seen in Table 3.2, all of the learning frameworks fulfil the two main requisites, but the chosen framework was TensorFlow [54]. This choice is due to the fact that this framework has extensive online documentation and examples, which make its usage simpler to understand. PyTorch [53] also has extensive documentation but there was previous experience with TensorFlow, so that was the chosen framework.

3.2.3 Deep RL algorithms

It is also necessary to choose the Deep RL algorithm that is going to be used to train the models. As indicated before in Chapter 2, there are several Deep RL learning algorithms, each of them having optimal performance in different circumstances. The work in [18] provides an extensive review on state of the art Deep RL methods, but since there are many different algorithms and it would take too much time to test all of them, two candidates were chosen: the PPO and SAC algorithms. There are different reasons as to why they were chosen to be used in this work. The PPO was chosen since it proved to work well in a very similar setup to the one that will be used in this work. As detailed in [3], the authors obtained good results with the PPO algorithm when training a model to achieve in-hand manipulation in the form of rotating a cube to achieve multiple different orientations. The SAC algorithm was chosen due to its ability to potentiate the exploration of different states [20].This is where SAC potentially outperforms PPO. As detailed in Section 2.1, the way SAC achieves the optimal policy, (2.4), is by including an exploration term that, with hyperparameter tuning, allows the same algorithm to benefit exploration over exploitation, or the other way around. In this case, since the problem at hand has a large state space, and one of the goals is to have a model capable of exploring the object, without any other particular objective, having the option to favor exploration over exploitation makes SAC a good algorithm to tackle this problem.

3.3 Action dimension

An important factor when implementing a RL model is to define what and how many actions the agent can take. With simpler problems, the dimension of the action vector might have a smaller influence in the learning process, but in this case the default action dimension is 20, which is a relatively large number. This means that such a number will have to be reduced, to promote faster training. For the case of the

Shadow Hand, the algorithm would have to choose 20 different values every time instant, making the process of learning the influence of each actuator, much slower. This may not seem much for a normal controller (without using RL) but with Deep RL it is important to remember that in the early stages of training, the algorithm is mostly performing random actions, and it will be hard for the model to learn about the system when there are 20 variables to test every iteration (which are not even independent in the case of the human hand).

One approach to solve this problem, or at least to aid the algorithm when searching for the best policy, is to reduce the number of active actuators. This would effectively make the model learn faster (since there are technically less parameters to check), but it would also mean that the hand would lose DOF, thus losing dexterity when manipulating. So, instead of completely removing a few actuators from the hand, intra-synergies were added (the prefix ”intra” means that the synergies are within a finger, instead of between fingers). These synergies will allow for some joints to move according to others, much like when someone bends a finger, they bend all three joints (the metacarpophalangeal, proximal interphalangeal and distal interphalangeal joints) at the same time. With these synergies, it is possible to reduce the number of actions needed for the model, with minimal cost to the dexterity of the hand. Two approaches were considered when choosing the right synergies for the model and they will be detailed in the subsections below.

3.3.1 *Ad hoc* Synergies

The *ad hoc* synergies were obtained by trial-and-error, while testing for the ability of the hand to move an object. As mentioned before, the objective was to reduce the number of actions while making sure the hand could still perform a good majority of the simpler movements and even some more complex ones. The first change that we can do is remove the 2 DOF that allow the wrist to move up/down and from side to side (indicated by ”Wrist 1” and ”Wrist 2” in Figure 3.3(b)). These 2 DOF will not be necessary since the goal is to only evaluate the ability to manipulate an object by only using the fingers. Another change was to remove the ability for the fingers (except the thumb) to perform the adduction and abduction movements (represented by the label ”Finger 4” in Figure 3.3(b), which allows the model to spread or join the fingers). Although this removes some freedom when manipulating, the advantage of having less 4 DOF out-weights the benefit of being able to move those 4 joints. Similarly to the distal interphalangeal joints (indicated by the ”Finger 1” label), the proximal interphalangeal joint (indicated by the ”Finger 2” label) were also transformed into passive actuators, following the rule in (3.2).

$$\theta_2 = \alpha \cdot \theta_1, \tag{3.2}$$

Table 3.3: Updates made to the DOF of the Shadow Hand. In this table, ”F” stands for ”Finger, ”T” for ”Thumb” and ”W” for ”Wrist”. The actuators that are referred in Table 3.3 are the ones presented in Figure 3.3(b).

	Actuators											
	F1	F2	F3	F4	Little F5	T1	T2	T3	T4	T5	W1	W2
Used	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	No	No
Parent joint	F2	F4	None		F4 (of the little finger)	T2	None		None	None		
α	1	1	N.A.		0.5	0.7	N.A.		N.A	N.A		

where θ_1 represents the amplitude of the joints indicated by ”Finger 1” and θ_2 represents the amplitude of the joints indicated by ”Finger 2”. The parameter α controls the influence of θ_1 over θ_2 but in this case, α was set to 1. Although with this change it now means that the fingers cannot be fully stretched (unless when resting), it was assumed that this would not have a considerable negative impact on the dexterity of the hand, in fact this is similar to what humans do when manipulate objects. At this point 10 DOF were removed, with 10 remaining. Another change made was to the thumb. As seen in Figure 3.3(b), the thumb has in total 5 DOF, being the most dexterous finger in the hand. Although the thumb is an important finger when manipulating objects, some changes were made to the actuators. Namely, the actuator labeled ”Thumb 3” in Figure 3.3(b) was removed. This actuator allowed the thumb to move laterally at approximately the middle of the finger. With a small range of motion of about 22º, it was opted to remove this actuator, since its contribution was smaller than other actuators. The other change made to the thumb was to make the actuator labelled ”Thumb 1” depend on the actuator ”Thumb 2” by adding the same relation as in (3.2) but this time with $\theta_{t1} = \alpha \cdot \theta_{t2}$ where θ_{t1} represents the amplitude of the joint indicated by ”Thumb 1” and θ_{t2} represents the amplitude of the joint indicated by ”Thumb 2”. In this case, α was set to 0.7. The final change was related to the actuator labelled ”Little Finger 5”, which allows the model to mimic the movement of closing the palm of the hand (by touching with the tip of the thumb on the tip of the little finger). This is another important actuator so it was decided to maintain it but make it dependent on the actuator ”Finger 3” of the little finger, which means that whenever the little finger bends, so does the ”Little Finger 5” actuator (in this case with $\alpha = 0.5$), closing the hand.

To recap, all the changes made to the base model are presented in Table 3.3. In total, with these synergies, we are left with a 7 DOF model that should be able to perform most manipulation tasks, as will be demonstrated in Chapter 4.

3.3.2 Human Synergies

Unlike the aforementioned synergies, these synergies were obtained by running experiments on human candidates. The authors of [7] asked volunteers to perform certain grasps with different objects, who were tele-operating a robot hand (the ShadowHand was one of the hands tested). The data for every different grasp was obtained and the synergies between the fingers were computed using the Principal

Component Analysis method. Eight total grasps were tested with them being: tripod, palmar pinch, lateral, writing tripod, parallel extension, adduction grip, tip pinch, and lateral tripod. All of these grasps are illustrated in Figure 3.5.

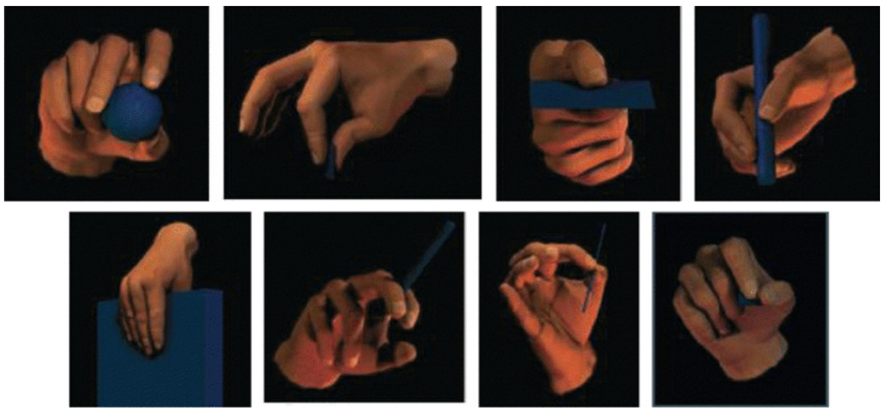


Figure 3.5: The precision grasp types selected for the experiments: tripod, palmar pinch, lateral, writing tripod, parallel extension, adduction grip, lateral tripod. Image taken from [7]

Two different types of synergies were obtained: the first type was a synergy for each grasp type tested; the second type was a set of synergies computed from the mean of all the previous synergies (from now on denoted as global synergies). The main result of [7] is that with 6 dimensions (6 different global synergies), it is possible to reconstruct, with a squared error lower than 10%, all the precision grasps considered in this work. The difference from the synergies presented previously, is that these were obtained from real human-object interactions, whilst the former were obtained by trial and error, so theoretically, the human synergies should yield a better result and a more human motion of the hand. Another key advantage of using these synergies is that it is possible to control the number of synergies used. Contrary to the previous synergies, where the action space was fixed to 7 dimensions, here the action space can vary between 0 and 21, which is the total number of components of the global synergies. As mentioned above, the first six components are enough to achieve most grasps but if more precision is required, more components can be added to further fine tune the movements.

Although these global synergies, in theory, will promote a more human-like manipulation, the majority of the experiments was conducted using the synergies detailed in Section 3.3.1, since the global synergies were only obtained on a later stage of the work.

3.4 Controller

The Controller is one of the most important parts of this work, since it will be responsible for actually manipulating the object. This Controller should take into consideration the state of the object and act on it, in order to achieve the desired result. As described in Chapter 1, there are two possible ways to define a controller. The first one is have a ”hard-coded” controller that, for every finger position and object state combination, would know how to move each finger to achieve the desired object pose. While this approach may be feasible for simpler problems, it would not be suitable for the problem addressed in this work, since there are too many variables to take into account, and defining a controller that would act on all of them would be very time consuming. The other approach is by using learning methods. Learning methods, namely Deep RL methods, are useful when the controller has a specific task to fulfil and the action-state space is too large to fully explore it. This last approach shows promise since other related works have shown to have good results with Deep RL methods in the field of hand manipulation. Before diving in more detail on the Controller, its main architecture is shown in Figure 3.6.

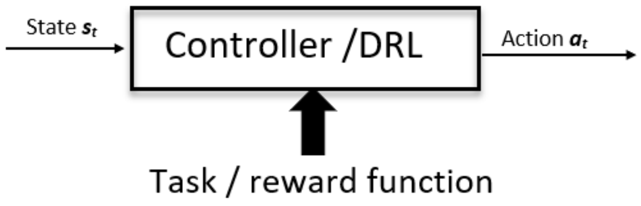


Figure 3.6: Architecture of the Controller

The Controller will be the Deep RL agent that was represented in Figure 3.1. In order to further define the Controller, it is necessary to identify the input, the output, and the tasks, *i.e.* the state, the actions and the reward functions respectively. A brief introduction to the actions was given in Sections 3.2 and 3.3 but there are other aspects that were changed regarding the actions.

The first one is related to how each action will impact the environment. By default the simulator is working with absolute actions, *i.e.* $\theta_t = \beta \cdot a_t$ where θ_t represent the amplitudes of each joint, a_t is the current action and β is a scaling vector that scales the range of the action ([-1, 1]) to the range of motion of each actuator. When taken to the extreme, this type of control can lead to problems to real robots since there are no boundaries relative to the difference of amplitudes from one instant to the next, causing the robot to execute actions that may damage it. To prevent this, relative control was implemented, *i.e.* $\Delta\theta = \beta \cdot a_t$ where $\Delta\theta = \theta_t - \theta_{t-1}$. This alone does not solve the problem, it is also necessary to limit the range of values that each action may take, from [-1, 1] to [-0.1, 0.1]. With this modification, the model is not able to go from one end of the range of motion of a finger to the other end, in one iteration.

It was also necessary to define what the state is. As mentioned before, the state will serve as input to the Controller, and it is what will help the Deep RL algorithm to evaluate the changes in the environment. Therefore, it is necessary that the state contains all the information that is required to fully describe a certain instant of the environment. This information is already available from the simulator, as described in Section 3.2, so the state can be defined as:

$$s_t = (j_t, j_{t-1}, y_t, \dot{y}_t, \theta_t, \dot{\theta}_t), \tag{3.3}$$

where j represents the values from the touch sensors (92 values); j_{t-1} represents the values from the touch sensors from the previous iteration (92 values); y represents the pose of the object, namely the position and orientation (in quaternions) (3 + 4 values), $\dot{y}$ represents the linear and angular velocities of the object (3 + 4 values), θ represents the amplitudes of each joint (20 values) and $\dot{\theta}$ represents the angular velocities of the joints (20 values). Many of these signals were already provided within the simulator but other were added to further detail the state, those being the $j_{t-1}, \dot{y}$ and $\dot{\theta}$ signals. All of these components lead to a state dimension of 238.

Lastly, it is also necessary to define a set of reward functions that will aid the Controller in learning specific tasks. Three main tasks were considered:

- ”random” babbling of the object
- rotate the object a certain amount, in a given axis
- lift the object of the palm of the hand

For a more visual intuition, a set of videos with the trained models performing tasks 1 and 3 is available here (the reason as to why task 2 was not included in the videos will be detailed in Chapter 4). The last two tasks are more goal-oriented and the first one is a more exploration-oriented task. The first task has the objective to explore the object, while constantly manipulating it, *i.e.* explore different states. This is important in order to collect data for training the Forward Model. The second and third tasks have the purpose of evaluating the controller in dexterous manipulation movements. These were the tasks that were implemented and tested since they cover three different objectives, with three different types of in-hand manipulation strategies. In the next subsections, the reward function for each task will be presented but first, the concept of reward shaping will be introduced.

Defining a reward function is a key factor when designing a RL model. It is the reward function that will guide the algorithm towards achieving the desired behaviour. If this function is not well suited for the problem at hand, the model will not learn properly. There are a few key factors to consider when defining a reward function, the more important ones of those being:

- the desired task
- the rewarding strategy: dense versus sparse reward

The desired task is the most important aspect when defining a reward function. As mentioned above the objective of having a reward function is so as to guide the RL algorithm to perform the desired task.

With humans, the better a task is defined, the less questions the person performing the task will have, and the faster the task will be concluded. The same happens with RL algorithms, the more explicit the reward function is, the better the model.

Directly related to the learning process of the algorithm, dense and sparse reward functions play different roles. The difference between dense and sparse rewards is related to the total number of possibilities that the agent has to receive a reward. As the names imply, a dense reward function will typically reward the agent at every time instant, whilst the sparse reward function will only reward the agent upon the achievement of the goal. Although having a dense reward function may make the model learn faster, since there is more feedback given to the model, this is not how humans would learn a task. A person performing a task, writing a document for example, does not receive feedback for every work written, but instead, upon completion of said document.

With these topics in mind it is possible to define a reward function for the tasks presented in Section 3.4, which will be detailed in the subsections bellow.

3.4.1 Task 1: Task-free object manipulation

The motivation behind this task is related to how babies would learn to manipulate an object. Babies often pick up an object without any particular reason and play with it with no particular objective. The objective of this task is not directly related to evaluating the performance of the Controller, but instead to gather data for the Forward Model. As explained before, it is important that the Forward Model is trained with rich and diverse manipulation data, so if the Controller is performing a task where the objective is to explore as much of the object as possible, the data collected will be as diverse as possible. Let's say for example that the task of lifting the object (will be detailed below) was the only one selected to generate data. This would imply that the Controller would mainly focus on getting better at the task which would mean that the Controller would try to perfect a technique (a grasp) in order to obtain a better reward. If this was the case, then the generated data would only contain information about that specific task, which is knowledge that is most likely not transferable to other tasks, such as the task of rotating the object. So, by having a task where the objective is to explore the object, it is possible to maximize the number of different states that the Deep RL algorithm explores, thus maximizing the data diversity. Although this seems a simple and pointless task for a human to do, it is hard to train a model to perform such tasks. This is related to how Deep RL models learn. As explained above, the most important aspect when choosing a reward function is to have a clear and defined goal, in order transfer this goal to the actual reward function. In the case of this particular task, there is no obvious goal, contrary to the tasks of rotating or lifting the object. Without a specific goal in mind, it will be hard to define an adequate reward function, thus hindering the learning process of the algorithm.

In order to overcome this problem, a workaround was implemented. In a certain way, exploring the object, can be seen as ”inspecting” every side of the object. With vision, this would translate to viewing every side of the object, but since the goal of this thesis is to only use tactile feedback, another method must be adopted. A way to mimic vision would be to keep track of all the orientations that the object was already in. So, by making use of the simulated environment (which allows the direct access to the object orientation), it is possible to motivate the algorithm to achieve orientations that have yet to be achieved. The reward function for this task is then defined as:

Algorithm 3.1: Reward function for the random babbling task.

```
Initialize dictionary with possible orientations $D1$
Initialize dictionary with key orientations $D2$
while running do
    $r_i = 0 \leftarrow$ reward for iteration $i$
    if object falls then
        $r_i$ = -10
        return $r_i$
    else
        $q_i \leftarrow$ current orientation
        foreach $q \in D2$ do
            if $q_i \approx q$ then
                $r_i = r_i + D2[q]$
                $D2[q] = 0$
        if $D1[q_i] == 0$ then
            $r_i = r_i + 1$
            $D1[q_i] = 1$
    return $r_i$
```

In sum, the reward function presented in Algorithm 3.1 will evaluate, at every time instant, if the current orientation was already visited or not. If yes, a reward of +1 is given, else nothing happens. There is also another way to reward the model which is if a certain key orientation is achieved. The dictionary $D2$ contains a set of key orientations that will award a given amount of reward to the model if said orientation is achieved. These key orientations are defined in Table 3.4.

Table 3.4: Key orientations and respective reward

	Reward
+90º	+3
-90º	+3
+/-180º	+6

The first column of Table 3.4 represents the change, in degrees, in each euler axis of the orientation, so in total there are nine key orientations. Although this provides an overall guide for the controller to follow, it is not the main way the algorithm will received rewards. It is instead by using the dictionary $D1$, which is initialized at the start of the experiment and contains most possible orientations. It is important to note that the dictionary $D1$ does not contain every single possible orientation, as that would be an infinite number. Instead, each orientation has 4 values (since they are expressed in quaternions) and each value was discretized into 11 bins, *i.e.*, the possible orientations are defined as $q = (x, y, z, w)$ where

$x, y, z, w \in \{-1, -0.8, -0.6, -0.4. -0.2, 0, 0.2, 0.4, 0.6, 0.8, 1\}$. Additionally, the algorithm is penalized with a reward of -10 every time the object is dropped. Since this negative reward has a bigger influence (the absolute value of the reward for dropping the object is greater than the reward for achieving new rotations), the model will first focus on not dropping the object and then focus on manipulating. With the reward function presented in Algorithm 3.1, one is able to manipulate the object by directly exploring different orientations.

3.4.2 Task 2: Rotating the object

Opposed to the previous task, this one has a well defined objective. The goal is to rotate the object a certain amount, around a given axis. Two different axis were tested, the *x* and *z* axis. The reference frame can be seen in Figure 3.7. In this experiment the object starts in the same orientation every time

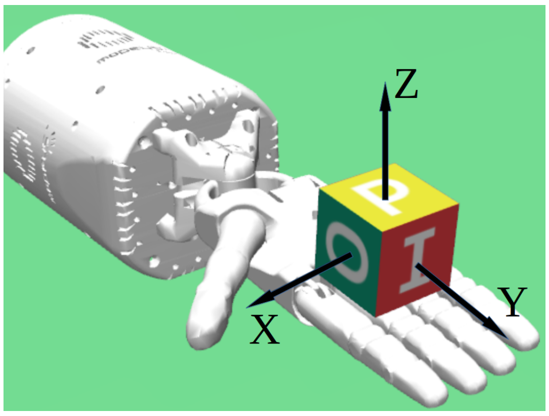


Figure 3.7: Reference frame used.

and the goal orientation is chosen by:

$$\theta_g = x \text{ where } x \in |x - \theta_c| > \frac{\pi}{6},$$

(3.4)

where θ_g is the goal orientation and θ_c is the current orientation. By forcing the goal orientation to be at least $\pm$ 30º than the current orientation, it is possible to eliminate cases where the goal orientation is achieved by accident.

Regarding the reward function, there are two possible reward functions that are standard in tasks like these, one being a sparse reward function and the other a dense reward function. The sparse one would only reward the algorithm if and when the goal orientation is achieved. On the other hand, the dense reward function rewards the algorithm on every time step, based on the difference between the current and goal orientations. Instead of using one of these reward functions, a mixed between them was adopted. Similarly to the reward function presented in Algorithm 3.1, the method is both rewarded for achieving some key orientations and for achieving the goal orientation. So the reward function for this task is given by:

Algorithm 3.2: Reward function for the task of rotating an object.

```
$q_g$ = sample_goal()
Initialize dictionary with key orientations $D2$
while running do
    $r_i = 0 \leftarrow$ reward for iteration $i$
    if object falls then
        $r_i$ = -10
        return $r_i$
    else
        $q_i \leftarrow$ current orientation
        foreach $q \in D2$ do
            if $q_i \approx q$ then
                $r_i = r_i + 1$
                $D2[q] = 0$
        if $q_g \approx q_i$ then
            $r_i = r_i + 20$
            $q_g$ = sample_goal()
    return $r_i$
```

The reward function presented in Algorithm 3.2 is similar to the one in Algorithm 3.1 since it will also reward the model for achieving key orientations. The difference here is that these key orientations are computed so as to serve as intermediary orientations between the current and goal orientations. They are obtained by computing the orientations that would lead to the goal orientation in the most straightforward way. Since we are only dealing with rotation in one axis these rotations are simple to compute. For example if the goal is to reach the orientation of (0º,0º,90º) (in euler angles) and the current orientation is (0º,0º,0º), these key orientations could be (0º,0º,30º) and (0º,0º,60º). But since the goal orientation is randomly generated, these key orientations need to be computed accordingly. Similarly to the previous reward function, each component of the key orientations is discretized into 11 bins.

3.4.3 Task 3: Lifting an object

This final task intends to explore the ability of the model to achieve precise grasping. The goal is to lift an object off the palm of the hand, ideally using the tip of the fingers. The reward function for this task is relatively simple and its given by:

$$r = is\_on\_palm() + (h_s - h_i) * 10, \tag{3.5}$$

where h_s is a constant height value, h_i is the current height of the object and the function $is\_on\_palm()$ will reward $\pm$ 1 depending if the object is touching the palm of the hand or not (-1 for being in contact with the palm and +1 otherwise). By penalizing the model for having the object touching the palm, the model will be encouraged to lift the object. It is possible to detect if the object is touching the palm of the hand by verifying the status of the touch sensors located in the palm. If any of them has a value different than 0 (meaning that some pressure is being applied) then it is considered that the object is touching the palm. The sensors that are considered to be part of the palm are depicted in Figure 3.8. The parameter h_s is defined as a constant, bigger than h_i could ever be. This way the algorithm will be constantly trying to minimize this difference. Additionally, the scalling factor of 10 is applied so that the "height" factor of the reward function has the same impact as the "palm" factor.

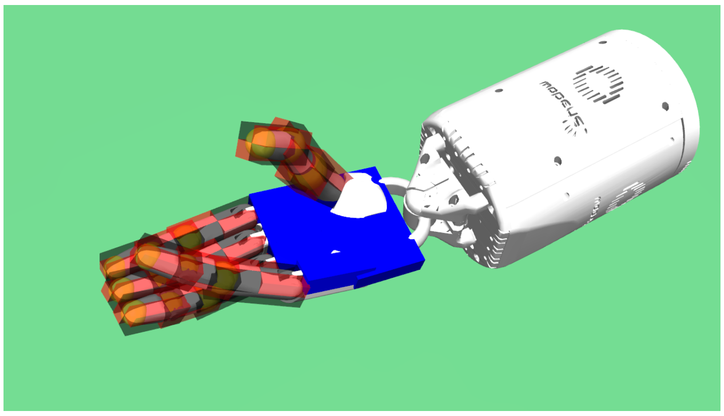


Figure 3.8: Touch sensors (in blue) used to decide whether the object is touching the palm of the hand or not.

3.5 Forward Model

The Forward Model is the main contribution of this work. Its goal is to provide the Controller with valuable information to be used when manipulating an object. This information should be obtained by training the Forward Model with the manipulation dataset, to extract the most useful features from the state vector. In the robotics area there is a common concept of having an Inverse Model, which has the objective to compute the set of actions a_t that would lead from state s_t to state s_{t+1}, so:

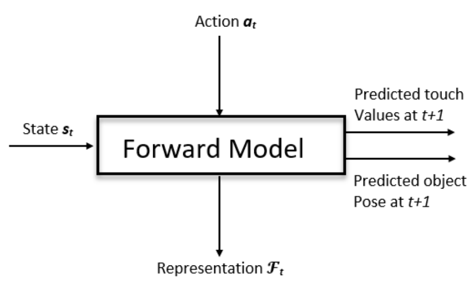
$$a_t = F_I(s_{t+1}, s_t), \tag{3.6}$$

where F_I is the Inverse Model. This equation is more useful with planning tasks, where there is a well defined start and final state and we want to know how to go from the start to the final. However, in cases where the dynamics of the robot are unknown, it may be more useful to learn how an action will impact the robot, given a certain state. This notion is also known as a Forward Model. The Forward Model will compute the state s_{t+1} based on the state s_t and action a_t, *i.e.*:

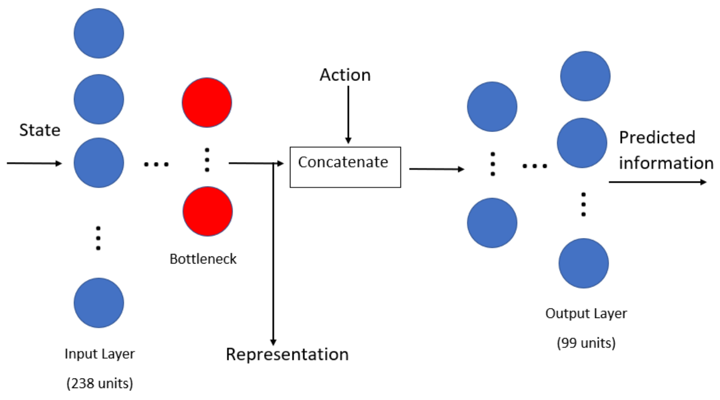
$$s_{t+1} = F_F(s_t, a_t), \tag{3.7}$$

where F_F is the Forward Model. The idea behind it, is to have a model that is able to learn the dynamics of a robot, in order to predict the next state, given a certain action. For instance, Wolpert *et al.* [56] propose this very exact model, that would predict the next state of a robotic arm, given the current state and a certain action. It is also important to note that, in order to obtain the Inverse Model, it is necessary to know in advance the dynamics of the robot, while this is not the case for the Forward Model. Moreover, learning an Inverse Model can lead to undesired solutions, such as multiple solutions for the same start and end state configuration.

Following this concept, the Forward Model will then be responsible for computing the next state, without knowing before hand how the robot (the hand) works. In a nutshell, the Forward Model will learn the dynamics of the robot. In the case of this work, this model will follow a specific architecture, similar to an AE, which was introduced in Chapter 2, since it will have an encoder and decoder network. The encoder part will be responsible for encoding the state vector, into a latent representation, which in turn will be decoded to the output vector (the next state). The difference from the Forward Model to the traditional AE, is that the output vector will not be equal to the input vector. This way, it is possible to train the Forward Model to learn to predict the next state (given an action), and to compute a representation of the current state, at the same time. Figure 3.9 depicts the architecture for the Forward Model.


Action a_t
State s_t
Forward Model
Predicted touch Values at $t+1$
Predicted object Pose at $t+1$
Representation $\mathcal{F}_t$

(a) Base architecture for the Forward Model.



(b) Detailed architecture for the Forward Model.

Figure 3.9: Architecture of the Forward Model. Note that in this case the action is used as input for the Forward Model but this is just when training the model.

There are five main elements in Figure 3.9: two input signals (the state and action) and three output signals (the latent representation, the predicted touch values and the predicted object pose for the next time instant). As indicated before, the Forward Model will follow an architecture similar to an AE. The main other difference from the standard AE, is that there is a signal that is only considered halfway through the model. In this case, the action vector is only taken into account for the decoder network. As seen in Figure 3.9(b), the representation is obtained by running the decoder part of the model with the state as input. Then, this representation is concatenated with the action and the concatenated vector is passed through the decoder network to predict the next state. This means that the learned representation only takes into account the current state of the model. The two main reasons for this are first, by only encoding the state (instead of the state-action pair), we are guiding the Forward Model to only summarize the important information about the state, which is the desired. Secondly, the option to only consider the action for the decoder is necessary given the Controller architecture. The Controller needs a state to output an action, however, if the action was required to compute the representation of the state, this would lead to a vicious cycle. The action is only included in this architecture when training the model. After training, the part of the Forward Model used is just the encoder part,

The most important aspect of this architecture is the latent representation, which is obtained at the bottleneck layer, represented in red in Figure 3.9(b). In the traditional AE, this layer has a lower number of units than the input or output layer and the reason for it is so that the input vector is forced to be represented in a lower dimension space. This way, only the most important features of the input will be encoded on the learned representation. This is an important aspect of the Forward Model since one of the goals of this work is to evaluate if using a lower dimension representation of the state instead of using the full state, leads to faster learning. More importantly, the Forward Model is forced to keep only the most important features of the input, allowing the representation to be mostly consisted of rich information that the Controller can use, instead of having a larger input vector that might have redundant information.

The number of layers and units per layer for the encoder and decoder parts of the Forward Model were obtained by testing several combinations and the results will be presented in Chapter 4. The way this model was trained was by using Supervised Learning, *i.e.* a dataset of input and output vectors was obtained beforehand, and then used to train the Forward Model. In order to make the Controller perform object-agnostic manipulation, the Forward Model must learn a representation of the input that is invariant to the object, to its weight, and even to the task used to collect the data. To achieve this, ideally there would be a dataset of manipulation with various objects and various types of grasps in order to provide the necessary data diversity, but such dataset is not available, so it was necessary to create it as it will be described in Chapter 4. This is where the task of having the Controller ”randomly” manipulating the object comes into play. By having the Controller manipulating the object with, ideally, different types of grasps and with different objects, it is possible to maximize the diversity of generated data.

The training process for this model will be further discussed in Chapter 4, but an important aspect is to have an adequate loss function. This function is what will dictate how the model will learn to perform a certain task, by evaluating the difference between the predicted output and the true output. Most common problems can be solved with a loss function such as the Mean Squared Error (MSE) or the Mean Absolute Error (MAE), but since the input data of this problem is very specific, a custom loss function was necessary. There are two main reasons for this. The first is related to the input data that comes from the touch sensors. As detailed before, this part of the signal consists of 92 values, each corresponding to a different touch sensor. The problem is that most sensors are not activated during manipulation. On average, about 7 sensors are triggered during a given time instant. This means that the part of the input that derives from the touch sensors will be mostly zeros. The second problem is related to the final prediction. As it can be understood from the different tasks, the pose of the object, namely the orientation, plays a key role in all of them. So, it is important that the prediction of the next pose is as accurate as possible. These conditions lead to the following loss function:

$$L = L_{pos} + \beta * L_{rot} + L_{sensors}. \tag{3.8}$$

In (3.8), there are 3 different losses, each associated with a specific part of the output, L_{pos} is related to the difference between the predicted object position and the true object position, L_{rot} is related difference in predicted and true object orientation and the same for $L_{sensors}$, considering the true and predicted values of the touch sensors. The factor β acts as a scalling term, in order to give more importance to certain losses. In this case, since the prediction of the orientation is of substantial importance, β was set to 10. The loss L_{pos} is relatively simple and it is given by the MSE metric. The remaining two losses are more custom. Regarding L_{rot}, the loss is given by the norm of the MAE between the true and predicted orientations. However, since we are dealing with quaternions, and the quaternion q originates the same orientation as the quaternion $-q$, a small modification was made.

$$L_{rot} = \min(\text{MAE}(q - \hat{q}), \text{MAE}(q + \hat{q})) \tag{3.9}$$

where q is the true quaternion and $\hat{q}$ is the predicted quaternion. This way, it is possible to teach the model to minimize the difference between the two quaternions. Lastly, regarding $L_{sensors}$, ideally, the goal would be to minimize the predicted touch sensors compared to the true values, but considering the type of data we have (where the data points for each entry are mostly zeros), standard loss functions cannot be applied. If we were to use MSE for example, the model would simply learn to bring the mean of the predicted values close to zero (since the mean of the true values is also close to zero), thus losing the important information, which is what sensors are active in a given instant. On the other hand, if we only focus the loss function on the error of the sensors that are actually active, the remaining sensor values would have no constraint and could lead to very incorrect predictions. In order to bypass these problems, the following loss function was implemented:

$$L_{sensors} = \begin{cases} \beta/2 \cdot (\hat{y} - y)^2, \text{ if } (y > 0 \wedge \hat{y} < 1e^{-3}) \\ 0, \text{ else} \end{cases} + \begin{cases} \beta/10 \cdot \hat{y}^2, \text{ if } (y == 0 \wedge \hat{y} > 1e^{-1}) \\ 0, \text{ else} \end{cases} , \quad (3.10)$$

where y are the true touch sensors values, $\hat{y}$ are the predicted ones, and β is the same as the one in (3.8). This function computes a vector (with the dimension equal to y and $\hat{y}$) where each value indicates the error for the correspondent sensor. If the true sensor is active but the predicted one is not, then that value is given by the first term of the loss in (3.10). On the other hand, if the true sensor is not active but the predicted one is, then the value is given by the second term of the equation. An important note here is that the importance of the first term is five times higher than the second term. This is because, as mentioned above, at a given time instant, there are about 7 active sensors (out of 92) and in order to give them the same importance (at least) than the non-active sensors, the scakking factor was added. In the end, we take the norm of the loss vector and feed that value to the optimizer.

3.6 Experimental Plan

So far, it has been defined what methods are going to be used to train both the Controller and the Forward Model, and also what tasks are going to be used to test both models. In this section, all the experiments that were conducted in order to evaluate the performance of each model are going to be discussed. The ultimate goal is to test if by having a Forward Model guiding the Controller when learning to perform a manipulation task, the results (in terms of reward) are better than by only having a Controller. In order to test this theory, the experiments presented in Table 3.5 were conducted. The corresponding results will be presented and analysed in Chapter 4.

In total, six experiments were conducted (in the order by which they appear in Table 3.5). Experiment 1 was prior to all experiments since it was necessary to first define what Deep RL algorithm was going to be used to train the Controller. As described before, the goal of this thesis is to evaluate the performance of a model that was trained with tactile feedback, the goal is not to test several Deep RL methods to see which performs better. Then, Experiment 2 was conducted in order to test if the reward functions presented in the previous section were adequate for the model to learn the proper tasks. In Experiment 3, the Controller will learn a task with multiple objects at the same time. The model will start learning with one object, and after some time the object will change and so on. The goal here is to evaluate if the Controller is able to adapt to the sudden change in the object. As mentioned before, Deep RL methods do not learn how to extract information, they just learn how to perform a task, so the trained model will most likely be overfitted. By changing the object, we are changing the environment, thus invalidating

Table 3.5: Set of experiments conducted throughout this thesis.

	Experiment number	Objects used	Objective	Using the Forward Model?	Tasks used
PPO vs SAC	1	Cube	Evaluate the performance of both algorithms	No	1
Training Controller with cube	2	Cube	Evaluate the performance of the Controller in the different tasks	No	1,2,3
Training general Controller / Collect data	3	All	Evaluate the performance of the Controller in all three tasks	No	1,2,3
Train and test Forward Model	4	All	Training and testing the Forward Model	Yes	-
Controller versus Forward Model	5	All	Evaluate the performance of the Controller versus the performance of the Forward Model	Yes	1,2,3
Ad hoc synergies versus human synergies	6	Cube	Evaluating the performance of the two types of synergies	No	1

most of the previously learned knowledge. Moreover, Experiment 3 also intended to evaluate the ability of the Controller to ”save” information. The order by which each object is used is fixed (and will be presented in Chapter 4) and the same object is used at least two times (after other objects have been used). So, ideally a model would ”remember” that it had already seen that object, and the training process would be faster. Finally, this experiment is also important in order to collect data for the next experiment: training and testing the Forward Model. Experiment 4 is a different experiment than the rest since here, it is only shown how the Forward Model was obtained.

Experiment 5 is the most important experiment of this work. As mentioned before, the ultimate goal is to test if the Forward Model will improve the performance of the model compared to just using the Controller. In this experiment, all three tasks will be explored and in each task, all three objects will be used, same as with Experiment 3. The difference here is that the Forward Model will also be used. Ideally, by using the Forward Model, the difficulties presented before regarding the expected inability of the Controller to quickly adapt to new objects and to ”remember” information, will be reduced.

Last but not least, in Experiment 6, both synergies described in Section 3.3 will be tested with just the cube and by just using Task 1. In this experiment, it was also evaluated the impact on using the full actuators of the human hand versus using the synergies, and serves as a starting point for future work.

4

Experiments and Results

Contents

So far, all the methodologies that were used in order to achieve the proposed goals have been described. It was presented a method for evaluating the ability of the Controller to perform certain tasks, and a method for evaluating if by using a Forward Model, the performance of the Controller will increase or not. In this Chapter, the set of experiments that was presented in Table 3.5 are going to be further detailed, and its results discussed.

4.1 PPO versus SAC

As mentioned in Chapter 3, there are two main Deep RL algorithms that show promise in solving the proposed objectives, the PPO and SAC algorithms. A preliminary experiment was conducted in order to decide which algorithm had a better performance and should be used in the following experiments of this thesis. In order to test both algorithms, two different tasks were considered. The first one was the task of exploring the object, defined in Section 3.4 and the second task was one just used for this experiment. The goal with the latter task was to keep the object from falling off, while the angle of the wrist if randomly changed according to the current angle, to mimic the movement of a shaking hand. The reward function for this task is relatively simple, the model gets penalized (with -20) if the object is dropped. Moreover, if that happens, the episode (sequence of states from an initial state to a terminal one) terminates, hence it is also important to evaluate the duration of the episodes for this task. The

Table 4.1: PPO versus SAC experiment parameters

		PPO	SAC
Environment parameters			
	State	$(j_t, j_{t-1}, y_t, \hat{y}_t, \theta_t, \hat{\theta}_t)$	
	Actions	described in Table 3.3	
	number of iterations	300000	
	episode duration	100	
Algorithm parameters			
	number of layers	2	
	units per layer	256	
	entropy coefficient	0.1	-
	replay buffer size	-	10000

parameters shown in Table 4.1 are common to both tasks and the object used in both tasks was the Cube. To make the learning process more challenging for the task of stabilizing the cube, the angle of the wrist was set to a random value at the start of each episode. The number of iterations of this task was also reduced from 300000 to 100000 because after 100000 iterations the average reward for this task was mostly constant.

To evaluate the performance of each algorithm, a set of tests were conducted. After training both policies, they were tested over 100 episodes, and the reward and episode duration was stored. This process was repeated 4 times and the results in Table 4.2, show the average of those 4 times. As it

Table 4.2: Results obtained for experiment 1: PPO vs SAC

	Explore		Stabilize	
	PPO	SAC	PPO	SAC
Avg. reward	7.58±4.21	1.94±2.35	-9.6±5.6	-11.1±5.76
Avg. episode duration	94.85±9.62	93.15±11.7	59.04±28.3	57.42±31.3

can be observed from Table 4.2, PPO outperforms SAC in both average reward and average episode duration, in both tasks. Regarding the task of exploring the object, the important factor is the average reward and PPO achieved almost 4 times the reward that SAC achieved, although the standard deviation was bigger for the PPO.

For the task of stabilizing the object, the important factor is the duration of the episodes. The average iterations per episode is relatively similar in both algorithms. With these results is it possible to conclude that PPO outperforms SAC, hence it will be the algorithm used in the following experiments.

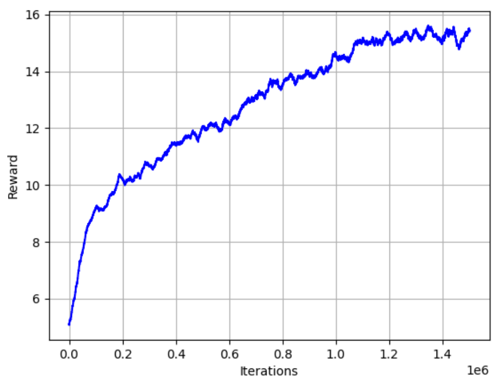
4.2 Training Controller with the cube

The goal of this experiment is to evaluate the performance of the Controller regarding the three different tasks detailed in Chapter 3, using the Cube. Before presenting the results on a test run, it will be first presented the learning process for each of the different tasks. The parameters that were changed throughout the different tasks are shown in Table 4.3.

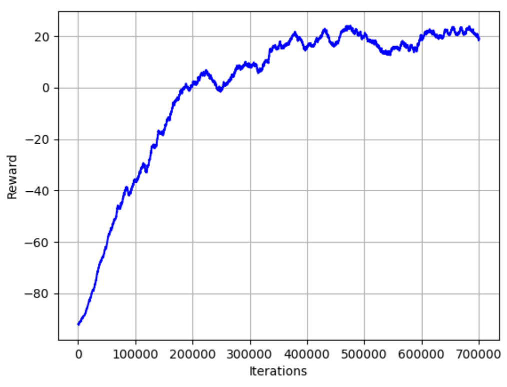
Table 4.3: Different parameters used for the different tasks.

	Task 1	Task 2	Task 3
Randomize initial position	NO	NO	YES
Randomize initial orientation	YES	NO	YES
Total iterations	1.5e6	8e5	7e5
Iterations per episode	100	1000	100
Use target orientation as observation	NO	YES	NO

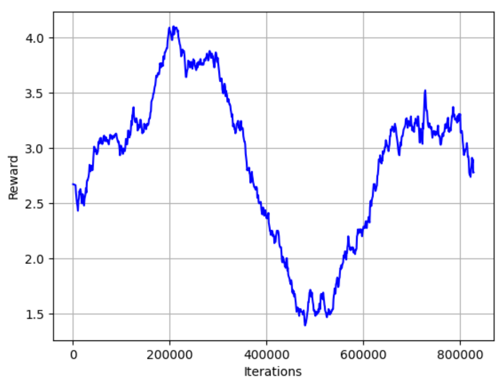
These parameters were chosen by observing how each of them impacted the overall reward. Ideally, the initial position and orientation of the object were both randomized at the start of each episode, but this will make the learning process slower, specially with more difficult tasks such as Task 2. When the randomization was on, the initial position was randomized around a predefined starting value, where each coordinate was sampled from a normal distribution with 0 mean and 0.005 standard deviation (in meters). Note that the position is an array with 3 components, *x*, *y* and *z*, that describes the position of the object in world space. For the rotation, if it was randomized, a random value between $-\pi$ and π was chosen for each axis at the start of every episode. Another important factor is the total number of iterations. The reason why each task was trained with a different number of iterations (and with a different number of iterations per episode for Task 2) will be discussed below. The other difference is the addition of the target orientation as part of the observation for Task 2. This will facilitate training since the model now has access to the current goal, instead of searching for it blindly. It is also important to note that each experiment was conducted 4 times (with different seeds each time) to minimize errors due to the natural variability of Deep RL methods. The results for the training process of each task are shown in Figure 4.1.



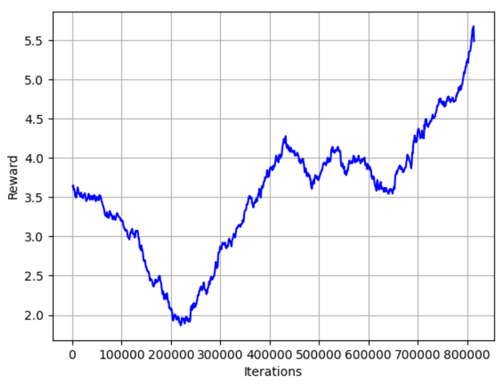
(a) Average reward during training for Task 1: task-free object manipulation.



(b) Average reward during training for Task 3: lifting the object.



(c) Average reward during training for Task 2: rotating the object (with axis Z as the rotation axis).



(d) Average reward during training for Task 2: rotating the object (with axis X as the rotation axis).

Figure 4.1: Reward during training for all three tasks for Experiment 2: training the Controller with the Cube.

It is now possible to justify the number of iterations for each task. For Tasks 1 and 3 (Figures 4.1(a) and 4.1(b) respectively), the average reward stabilized towards the end of training, indicating that further training would most likely not lead to any major improvements in terms of reward. This is not the case for Task 2. Although having trained the Controller with 800000 iterations, and using 1000 iterations per episode instead of 100, the progress in the reward is not very visible and if one were to train the model with more iterations, the results would be very similar. Even with 2000000 iterations the average reward would still not increase. Since in this case it is unclear if the model learned to complete the task, this result will be further analysed when discussing the test results for each task. The reason as to why 1000 iterations per episode were considered in this task instead of 100, is that it was difficult for the model to reach the end orientation with only 100 iterations.

In order to evaluate if indeed the Controller learned to perform the current task, a set of tests were conducted, where each task had a specific key metric:

- Task 1: how many different orientations were achieved
- Task 2: how many goal orientations were achieved
- Task 3: percentage of iterations where the object was elevated

Each test was conducted over 1000 episodes for Tasks 1 and 3, and 500 episodes for Task 2. The results were averaged among all episodes and then averaged again between the 4 different models (the different seeds) for each task. The results are shown in Table 4.4:

Table 4.4: Results for each key metric for Experiment 2: training Controller with the Cube. The row ”Max” indicates the maximum value of the current metric obtained in a single episode.

	Task 1 - Explore		Task 2 - Rotate		Task 3 - Lift	
	Trained	Untrained	Trained	Untrained	Trained	Untrained
Key metric	8.61±5.22	4.6±2.4	0.23±0.45	0.22±0.46	64%±28%	6%±11%
Max	25	15	3	2	100%	80%

In the table above, it is presented the results obtained for each task by using a trained model and an untrained one. For Tasks 1 and 3 it is clear that the performance of the trained model is greater than the untrained one. Regarding Task 1, where the key metric is how many different orientations were achieved, the average reward of the trained model is almost twice as much compared to the untrained model. The row ”Max” refers to, in this case, the maximum number of different orientations in a single episode. Since the episode has 100 iterations, 25 different orientations means that every 4 moves, a different orientation was achieved.

Regarding Task 3, the difference between the trained and untrained model is even greater. The goal here was to evaluate the ratio between the iterations where the object was in contact with the palm, and when it was not. It is also possible to observe that for the untrained model, the maximum was of 80%, which is a relatively big number considering the model was not trained. But this value can be treated as an outlier since the average reward is far lower (and with a low standard deviation).

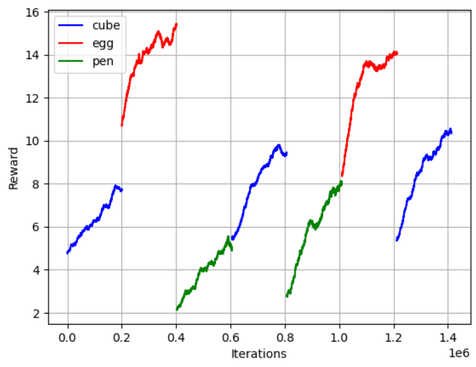
Lastly, Task 2 does not share the same conclusions as Tasks 1 and 3. The results shown in Table 4.4 are relative to the model where the rotation axis was the Z axis, but the results were the same for the X axis. The metric here is how many goal orientations were achieved in a single episode. The difference between the trained and untrained model is minimal, indicating that the model could not properly learn the task. This can also be observed in Figures 4.1(c) and 4.1(d). The average reward constantly varies during the training process, and if one was to compute the absolute average for both, the reward would remain mostly at the same value. It was also observed that training the model for more iterations (2000000) would not solve the problem. There can be multiple reasons for this, namely the reward function itself or the hyperparameters of the algorithm. Since using the axis Z or X yields the same results, from now on, whenever the task of rotating an object is used, it will only be shown the results where the rotation axis is the Z axis.

A set of videos with the trained models performing tasks 1 and 3 is available here.

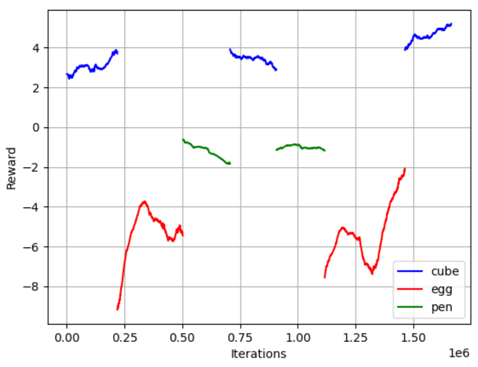
4.3 Training general Controller / Collect data

From the previous experiment, it was possible to prove that the Controller is able to learn a single given task. But a Controller that can only perform one task with one object, is not the desirable, given that the goal is to obtain a more general Controller. So, in order to test if the Controller is able to generalize to different objects, another experiment was conducted. This time, instead of just using a single object during training, the object is changed mid training, forcing the model to adapt to the new object. The training will start off with one object, then change to another and so on, going through the objects depicted in Figure 3.4. The order in which each object appear is fixed and is as follows: Cube $\rightarrow$ Egg $\rightarrow$ Pen $\rightarrow$ Cube $\rightarrow$ Pen $\rightarrow$ Egg $\rightarrow$ Cube. With this order, it is possible to test every possible transition between two different objects. The parameters used for each task are the same as in Table 4.3, except for the total number of iterations. For this experiment, a different model was trained for each object with 200000 iterations. As before, each task was trained 4 different times, each time with a different seed, to minimize outliers. Following the same principle as in the previous experiment, first we will analyse the training process for each task, and after the test results.

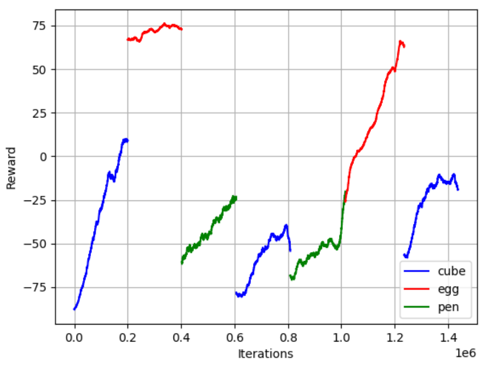
The results after training for each task are shown in Figure 4.2. It is important to note that each line in Figure 4.2 represents a different model, and different colors represent different objects. Additionally, each model's starting point, is the previous model's ending point. So this can be seen as one big individual model that was divided into 7 different models (one for each object used in the aforementioned order). Each of these lines will also be referred to as training periods. After training, the most noticeable observation for Task 1 (Figure 4.2(a)) is that with the Egg (red lines) the average reward is greater than the ones with the Cube and Pen. This can be justified by analysing the shapes of each object. Since for this task, the reward is obtained by generating never before seen orientations, the easier this is achieved, the more probable it is to have a high reward. Comparing the Egg to the Cube and Pen, it is clear that the



(a) Average reward during training for Task 1: task-free object manipulation.



(b) Average reward during training for Task 2: rotating the object.



(c) Average reward during training for Task 3: lifting the object.

Figure 4.2: Reward during training for all three tasks for Experiment 3: train general Controller.

Egg is a more unstable object, making it easier to ”roll” on the palm, allowing the Controller to achieve greater rewards, with less effort. Contrary to the Egg, the Pen is an hard object to grasp, hence its average reward is the lowest of all three objects. The other main conclusion is that it is possible to confirm that the Controller has the ability to remember that it has used that object previously, or at least to reuse knowledge obtained from training with other objects. Take the example of the Cube, the three blue lines. It is possible to observe in Figure 4.2(a) that the reward at the end of each training period increases from the first time to the second, and increases again from the second time to the third. This result is even more visible with the pen. This conclusion will be further explored in the testing phase.

Moving to Task 2, which is represented in Figure 4.2(b), there are also interesting conclusions to point out. The first one is that in this task, the observation that was presented before regarding the ability of the Controller to reuse knowledge is also present. By observing the training periods of the Cube, it is possible to confirm that the final reward, after the third time, is greater than the one from the first time. Moreover, the starting reward for each training period for the Cube, is greater (or at least equal) to the ending reward from the previous period. On another note, the plot in Figure 4.2(b) has a detail that is not present in the other two figures, which is that the Egg is the object for which the reward is the lowest. Once again, this is related to the shape of the object itself. The fact that the Egg is more unstable than the other objects, which worked in favor for Task 1, now has the opposite effect. Since the objective with this task is to achieve a very specific orientation, the instability of the Egg makes it harder to precisely manipulate it to achieve the desired goal. Moreover, it can be observed that the average reward during training is mostly negative, which means that the Egg was dropped multiple times during training.

Finally, regarding Task 3, Figure 4.2(c), the main observation is that the Egg has the highest reward of all 3 objects. Once again, it is possible to confirm the ability of the Controller to reuse knowledge from previous objects. Taking a closer look at the training periods for the Egg, the learning curve for the second period is very different from the one of the first period. The reason for this is that the Cube has a more similar shape compared to the Egg, than the Pen has. So, the Controller is able to reuse the knowledge obtained from training with the Cube, but not so much the knowledge obtained from the Pen. This also works the other way around. If we look at the second and third training periods of the Cube, it possible to observe that the starting reward right after training with the Egg is bigger than the one after training with the Pen (third and second training periods, respectively). It is also possible to observe that the second time the Pen appears, the reward starts out lower, but ends up at the same reward as the one from the first time. The same happens for the Egg. This is because, the reward from the previous model was already low in the first place, hindering the beginning of the next training period.

An important observation to point out is that, for a considerable amount of transitions between objects, there is a decrease in reward. This means, that although the Controller has the ability to remember information, it still takes some time to adapt to the new object. This is most visible for Task 2. Ideally, the Forward Model will mitigate this effect, since it will provide an object-agnostic representation of the state.

Same as with the previous experiment, the test results for all three tasks are going to be discussed. Once again, each test was conducted with 1000 iterations for Tasks 1 and 3 and 500 for Task 2. The key metrics used for each task are the same as the ones is the previous experiment. The difference here is that, instead of just using the final model, the tests were conducted for every intermediary model. So, for every task and for every object used in that task, a model was tested. The results for each key metric are presented in Table 4.5 .

It is important to note that each row in Table 4.5 represents a different training period. The conclusions that were demonstrated above can also be proved by observing the results in Table 4.5. For example, regarding the ability of the Controller to remember that a certain object was already used. If

Table 4.5: Results obtained for Experiment 3: train general Controller. The order of the rows is the order in which each object was used and the number on the name of the object is referring to the number of times that object was used before.

Row	Object	Task 1 - Explore	Task 2 - Rotate	Task 3 - Lift
1	**Cube 1**	8.53±4.08	0.22±0.45	0.56±0.31
2	**Egg 1**	15.80±4.51	0.01±0.08	0.96±0.05
3	**Pen 1**	6,48±3.90	0.00±0.01	0.39±0.38
4	**Cube 2**	10.08±4.20	0.21±0.42	0.27±0.20
5	**Pen 2**	8.93±5.54	0.00±0.00	0.31±0.30
6	**Egg 2**	14.92±4.65	0.02±0.08	0.83±0.11
7	**Cube 3**	10.93±4.56	0.22±0,44	0.44±0.25

we look at Task 1, more specifically to the blue rows, we see that the average value for the metric is increasing from one training period to another. It is also possible to prove that the knowledge obtained from the last object, can be useful knowledge for the next training period. However this does not happen for the Egg. The reason for this is related to the order in which the objects were trained with. Looking at row 6 for Task 1, it was expected that the Egg had an increase in reward compared to row 2, but since the previous object that the model had trained with was the Pen, the reusable knowledge is lower, since the Pen and Egg have very different shapes. Another interesting observation is if we look at Task 3, rows 1, 4 and 7, we see that the average value in row 7 is bigger than the average value in row 4 (which in turn is smaller than the average reward in row 1). This goes against what was previously observed in Task 1, but the reason for this is, once again, related to the order in which each object was trained. The training period from row 1 had no previous model since it is the first one, row 4 had as its previous object the Pen, but in row 7, the previous object was the Egg. Since the Egg has a more similar shape to the Cube than the Pen does, the model is able to reuse that knowledge and adapt faster to the new object. This is also the reason as to why there is a drop in the average value from row 1 to row 4, because the previous object for the model in row 4 was the Pen. On the other hand, it is now safe to assume that the Controller was not able to properly learn Task 2. The average value for the Egg and Pen is very close to 0 and for the Cube it is slightly higher, but not enough.

As mentioned in Chapter 3, this experiment also had the purpose of collecting data in order to later train the Forward Model. The reason as to why Task 1 was chosen to collect the data is that this task is the one that provides more general manipulation information since there is no particular goal. The collected data contain information regarding the manipulation of all 3 objects and how these files are used is going to be discussed in the next Section.

4.4 Train and test Forward Model

The Forward Model was designed to help the Controller in making more informed decisions regarding the manipulation of a set of objects. The objective is to have the Forward Model filter the raw observation obtained from the environment and output a new input to the Controller, that will only consist of relevant information for manipulating all objects. An ideal Controller would be able to manipulate any object. Thus, the input that the Forward Model produces must be object-free, *i.e.* the Forward Model should be able to understand the important properties of the current object, and produce an output accordingly. In this section it will be described how the Forward Model was obtained. In order to achieve the final model, two characteristics need to be considered: the architecture and the data.

The general architecture of the Forward Model was previously discussed in Section 3.5, but now it will be presented the detailed architecture, *i.e.* including the number of layers and units per layer. This is depicted in Figure 4.3.

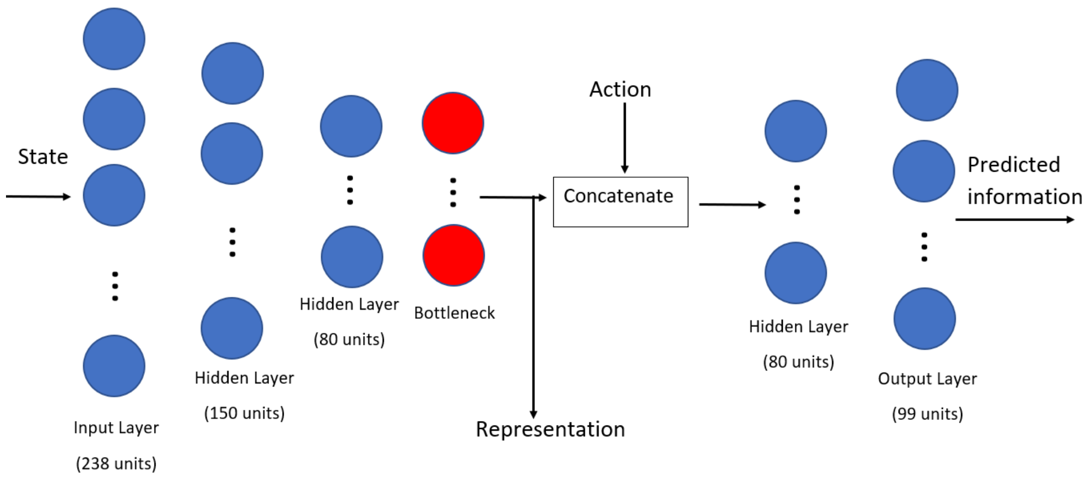


Figure 4.3: Detailed architecture, with number of layers and units per layer of the Forward Model used.

. All the layers are fully connected, with the ReLU activation function, except for the bottleneck layer, which used the *tanh* activation function. The is because the output of this layer will be concatenated with the action, and in order to make sure that each value of concatenated vector is approximately in the same range of values, the absolute values of the latent representation are forced to be in the range [-1,1].

In order to have a general Forward Model, the data needs to be versatile and vast so that the model is able to predict the next state accurately, even with never before seen inputs. As described before, the data was obtained by collecting information on every iteration throughout training of the Controller, using Task 1. Supervised Learning was used to train the Forward Model and the Adam optimizer [57] was used. Its parameters were kept at their default values:

- learning rate (α) = 0.001. Refers to the amount that the weights are updated during training.
- exponential decay rates for the moment estimates ($\beta_1 = 0.9$ and $\beta_2 = 0.999$). Control the exponential decay rates of the moving averages, which are estimates of the 1st moment (the mean) and the 2nd raw moment (the uncentered variance) of the gradient.

Regarding the actual training of the Forward Model, the full data set (around 1300000 data entries) was split into train, validation and test data, where the test data is 20% of the full data set (260000 data entries) and the validation data is 30% of the remaining data (312000 data entries). Before splitting the data, all entries were shuffled, so that the validation and test data contained data entries from different points of the training periods. Moreover, early stopping was implemented, with a patience (the number of epochs to wait before early stop if no progress on the validation set) of 15 epochs, to prevent the model from overfitting. In order to obtain the best Forward Model, different models were trained and tested in the test dataset where two different parameters were changed: the batch size and the dimension of the latent representation (number of units of the bottleneck layer). The results shown in Table 4.6 represent the 4 different evaluation metrics of the Forward Model:

- Table 4.6(a) represents the norm of the absolute difference between the true and predicted touch sensor values.
- Table 4.6(b) was obtained with the same metric as in Table 4.6(a) but only for the cases where the true sensors were active.
- Table 4.6(c) shows the average relative angle between the true and predicted orientations, in degrees.
- Table 4.6(d) represents the MSE between the true and predicted object position (in millimeters).

There are two main observations from the results presented in Table 4.6. The first is that the model is able to learn to predict the next object orientation given a state and an action with an error of about 17º. As mentioned above this is important since most tasks rely on the orientation to learn. The second observation is that there seems to be no difference in using a representation of 5 units versus using one of 40 units. Theoretically, the representation with 40 units would be able to more accurately predict the information since it has more ”space” to code the information into, but this was not verified. A possible reason is that the model was not able to learn how to predict the values regarding the touch sensors. By observing Tables 4.6(a) and 4.6(b), it is possible to observe that they represent the biggest loss between all the losses. That could be due to the fact that the absolute values themselves are higher but it is also possible to observe that they have a relatively high standard deviation. This implies that the model is neither capable of predicting the pressure on the various touch sensors, and of predicting when and which sensors are active. This suggests that the model is not capable of consistently predicting the

Table 4.6: Results obtained for the different instances of the Forward Model, varying the batch size and number of units for the bottleneck layer.

(a) MAE between the true and predicted touch sensor values.

		Batch size	
		128	**256**
Dim	**5**	6.05±7.08	6.06±7.07
	10	6.06±7.07	6.05±7.08
	20	6.05±7.08	6.06±7.07
	40	6.06±7.07	6.06±7.07

(b) MAE between the true and predicted touch sensor values (only for the cases where the true sensors where active).

		Batch size	
		128	**256**
Dim	**5**	5.97±7.14	5.96±7.13
	10	5.96±7.13	5.97±7.14
	20	5.97±7.14	5.96±7.13
	40	5.96±7.14	5.96±7.13

(c) Average relative angle between true and predicted quaternions, in degrees.

		Batch size	
		128	**256**
Dim	**5**	17.2±10.2	17.4±10.1
	10	17.2±10.1	16.9±10.1
	20	17.3±10.3	17.1±10.1
	40	17.1±9.9	16.8±10.0

(d) MSE between true and predicted object position, in millimiters.

		Batch size	
		128	**256**
Dim	**5**	15±9	14±8
	10	15±9	14±9
	20	15±9	14±9
	40	15±9	14±8

touch values, which could have hindered the learning process for the remaining losses, thus explaining the similar results between all models.

Nonetheless, from this point on, a specific model needed to be chosen, so it was opted to use the model that was trained with batch size 256 and had 40 units for the representation. This was the chosen model, since as stated before, a bigger representation dimension is often related to more accurate representations. Figure 4.4 shows the training and validation loss for this particular model. Another reason as to why there is little to no difference from using a representation with 5 or 40 units, could be due to the fact that there are 92 touch values to predict whereas for the pose, there are only 7. The Forward Model might have not been able to find any relation between the touch sensors and object position and action. If we observe Figure 4.4, the loss drops over the course of training, indicating that the Forward Model is learning. But we also observe that the loss stagnates at around 1.6, indicating that the model in fact learned to some degree but then it was not able to lower the loss even more, possibly due to the errors in the touch sensors predictions.

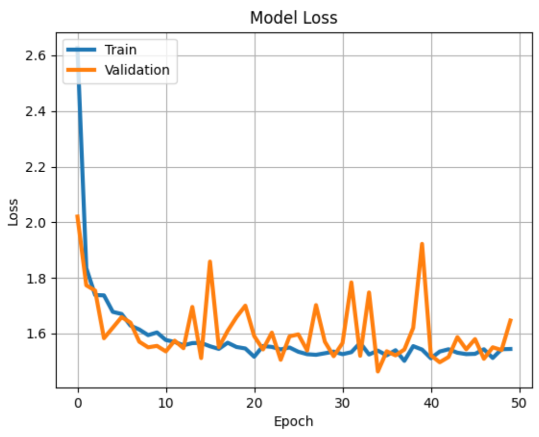


Figure 4.4: Training and validation loss for the Forward Model trained with a batch size of 256 and with 40 units on the bottleneck layer.

4.5 Controller versus Forward Model

Now that the both the Forward Model and the Controller have been tested separately, it is time to evaluate if using the representation obtained from the Forward Model to serve as input for the Controller, will indeed help with the learning process. The setup for this experiment is the same as the one in Section 4.3, but now we are testing the models obtained by using the Forward Model versus the ones that do not use it. As before, first it will be analysed the training process for each model and then tests will be conducted to further analyse the differences between them. The results for the training process are depicted in Figure 4.5.

As before, the blue lines in Figure 4.5 represent the models using the Cube, red lines for the Egg and green lines for the Pen. The common observation between almost every training period is that the models using the raw Controller tend to have a higher average reward. This is mostly visible in Tasks 1 and 3 (Figures 4.5(a) and 4.5(c) respectively). However, this does not necessarily imply that the Forward Model failed to fulfil its purpose. It is normal that the raw Controller outperforms the models using the Forward Model, because in this case the Controller is only training with one object at a time, while the Forward Model was designed to be object-agnostic. The other observation is that the training periods for Task 2, Figure 4.5(b) are quite different from the ones of Task 1 and 3. With Task 2, the difference in performance for both models is less visible, and in some cases, the model using the Forward Model even outperforms the one without. The reason for this is that, as discussed before, even the raw Controller has difficulties learning this task. So the models using the Forward Model are outperforming a model that is not as stable as the ones of the other tasks. Nonetheless, it is possible to observe that in fact using the Forward Model in Task 2, led to a better model, at least with the Pen.

In the previous observations, the models using the Forward Model were being compared to a model specialized in learning a particular task for a given object. This is an unfair comparison, since the purpose of the Forward Model is not to be better than the raw Controller in these cases. The main goal of the Forward Model is to provide more stable object manipulation, when using previously unseen objects. In order to test this, each trained model was tested with each object, for all 3 tasks, using the key metric that was previously defined for each task. The results are presented in Tables 4.7, 4.8 and 4.9. In each table, for each object, two models were tested: the model using the Forward Model (represented by ”FM + C”) and the model using the raw Controller (represented by ”C”)

Table 4.7: Results for the different models tested with all objects for Task 1: task-free object manipulation. The colored cells represent the cases where the object tested is the same as the object that the model was trained with.

Models \ Object	**Cube**		**Egg**		**Pen**	
	FM + C	**C**	**FM + C**	**C**	**FM + C**	**C**
1 - Cube	5.43±2.72	7.49±3.75	9.22±3.11	13.47±3.52	3.71±1.81	3.23±1.61
2 - Egg	4.48±2.58	6.35±3.96	10.93±3.96	16.34±4.43	3.49±1.73	3.21±1.54
3 - Pen	4.98±2.71	5.46±2.73	9.12±3.18	9.88±3.28	4.01±2.27	5.09±2.79
4 - Cube	5.30±2.75	9.17±3.80	9.33±3.10	13.34±4.04	3.78±1.94	3.74±1.76
5 - Pen	5.14±2.74	6.64±3.01	9.19±3.47	11.19±3.56	4.03±2.19	8.40±4.91
6 - Egg	4.59±2.50	4.49±2.74	10.53±3.98	14.79±4.69	3.55±1.84	2.98±2.37
7 - Cube	5.50±2.76	10.45±4.78	8.51±2.64	14.82±4.22	3.87±2.03	5.11±3.54

Let us analyse each table independently, starting with Table 4.7, which regards Task 1. From the overall results, there are two main observations. The first, which was also observed in the Figures 4.5(a), 4.5(b) and 4.5(c), is that the models using the raw Controller are able to achieve a higher average reward for almost every combination of model-object than the models using the Forward Model. However, it is also possible to observe that the models using the Forward Model consistently achieve a lower standard deviation. This leads to the conclusion that although the average performance is lower, the reproducibility and reliability of the model is better when using the Forward Model, since the model is able to achieve the same reward more consistently. Another important observation concerns the models with which the Pen is tested. It is possible to confirm that for almost every model, the models where the Forward Model was used have a higher reward than the ones where it was not used. This increase in performance is once again related to the ability of the model using the Forward Model to be more versatile. This is an important result since the Pen is a different object than the Cube or the Egg, which means that if other objects with unique shapes were tested, the models using the Forward Model would most likely perform better. However, this is not true for the cases the model being tested was also trained with the Pen (lines 3 and 5) but the reason for this was explained before.

Regarding Task 2 (Table 4.8) the results are not conclusive. As shown before, the model had trouble learning the required task, whether it was using the Forward Model or not. The only object that has any relevant information is the Cube but it is not possible to draw any conclusion from this data.

Table 4.8: Results for the different models tested with all objects for Task 2: rotating the object. The colored cells represent the cases where the object tested is the same as the object that the model was trained with.

Models \ Object	Cube		Egg		Pen	
	FM + C	**C**	**FM + C**	**C**	**FM + C**	**C**
1 - Cube	0.16±0.37	0.25±0.48	0.01±0.07	0.02±0.14	0.00±0.00	0.00±0.00
2 - Egg	0.16±0.37	0.18±0.39	0.01±0.07	0.00±0.00	0.00±0.00	0.00±0.00
3 - Pen	0.15±0.38	0.14±0.35	0.00±0.00	0.04±0.2	0.00±0.00	0.00±0.00
4 - Cube	0.24±0.45	0.12±0.33	0.01±0.07	0.02±0.1	0.00±0.00	0.00±0.00
5 - Pen	0.15±0.36	0.21±0.45	0.00±0.00	0.01±0.07	0.00±0.00	0.00±0.00
6 - Egg	0.14±0.35	0.22±0.42	0.04±0.20	0.00±0.00	0.00±0.00	0.01±0.07
7 - Cube	0.15±0.36	0.21±0.41	0.01±0.07	0.04±0.19	0.00±0.00	0.00±0.00

Table 4.9: Results for the different models tested with all objects for Task 3: lifting the object. The colored cells represent the cases where the object tested is the same as the object that the model was trained with.

Models \ Object	Cube		Egg		Pen	
	FM + C	**C**	**FM + C**	**C**	**FM + C**	**C**
1 - Cube	0.08±0.13	0.56±0.32	0.5±0.33	0.92±0.12	0.24±0.3	0.17±0.26
2 - Egg	0.23±0.31	0.25±0.33	0.92±0.13	0.96±0.05	0.32±0.38	0.21±0.3
3 - Pen	0.07±0.17	0.06±0.13	0.49±0.33	0.55±0.3	0.25±0.33	0.35±0.39
4 - Cube	0.09±0.15	0.29±0.19	0.47±0.20	0.45±0.14	0.16±0.23	0.17±0.23
5 - Pen	0.07±0.12	0.06±0.09	0.41±0.25	0.3±0.19	0.19±0.26	0.32±0.3
6 - Egg	0.13±0.24	0.1±0.19	0.84±0.18	0.7±0.17	0.27±0.34	0.26±0.33
7 - Cube	0.12±0.17	0.3±0.2	0.53±0.32	0.62±0.15	0.2±0.26	0.21±0.3

Moving on to Task 3 (Table 4.9), it is once again possible to prove the observations that were made regarding Task 1. It is possible to verify that the models using the Forward Model have, on average, a lower standard deviation, and that the models not using the Forward Model, have, on average, a higher reward. Once again, if we look at the column were the Pen was tested, it is possible to observe that the models using the Forward Model have a higher reward, except for the cases discussed above. Moreover, it is possible to observe that whenever the last object trained with was the Pen (rows 3 and 5) the performance drops for both models. This is most likely due to the unique shape of the Pen, that was little to no similarities with the Cube and Egg, making the control strategy less effective. It is also possible to observe that the performance of the Forward Model with the Egg, when the model was trained with the Egg, is as good as with the Controller. This could be due to the fact that Task 3 is a relatively simple task to learn, specially with the Egg, as seen in Figure 4.5(c). Another interesting observation here is that the performance with the Cube of the models using the Forward model is relatively low. The reason for this is unknown but it could once again be due to the fact that the Forward Model had trouble predicting the touch sensor values, which may have hindered the overall performance.

To conclude this experiment, a last table was obtained. Table 4.10 is the summary of the 3 tables above, where each value is the average reward for a given Task, using each of the objects. For instance, the first value is the average of the rewards of all the models using the Forward Model, that were tested with the Cube and using Task 1. It is once again clear that although the models using the Forward Model

Table 4.10: Average rewards for all the models for the different tasks and objects.

	Cube		**Egg**		**Pen**	
	FM + C	**C**	**FM + C**	**C**	**FM + C**	**C**
Task 1	5.06±0.4	7.15±2.08	9.55±0.86	13.4±2.23	3.78±0.21	4.54±1.92
Task 2	0.16±0.03	0.19±0.05	0.01±0.01	0.02±0.02	0.00±0.00	0.00±0.00
Task 3	0.11±0.06	0.23±0.18	0.59±0.2	0.64±0.24	0.23±0.05	0.24±0.07

have a lower reward among all tasks, the standard deviation has the opposite effect.

4.6 *Ad hoc* synergies versus human synergies

The previous experiments were all conducted using the *ad hoc* synergies. However, as described in Section 3.3, there is the option to use human synergies. This experiment is only here to serve as a starting point to improve upon the current setup. The goal with this experiment is to evaluate whether using the hand synergies, over using the full DOF, leads to improvements on the training time and on the performance. To test this theory, 3 different models were tested using Task 1: one with the 18 DOF (the Shadow Hand has 20, here we are removing the two DOF that control the wrist); another using the *ad hoc* synergies; and the final one using the human synergies. Each model was trained under the same conditions as the ones in Section 4.2, while varying the actions each could take. As before, each different model was trained 4 times with 4 different seeds, to minimize variance errors. Figure 4.6 depicts the training process for each of the different models and

Table 4.11: Average number of iterations per episode during training and final average reward for the 3 different models.

	18 actions	Human synergies	*Ad hoc* synergies
Average number of iterations per episode	97.55±1.25	99.81±0.31	99.92±0.11
End reward	12.34	14.96	15.76

From Figure 4.6, it is visible that both models that use synergies have an higher average reward during training then the model using the 18 DOF. Moreover, the model using the *ad hoc* synergies seems to have achieved a higher reward after training than the other two models. As it will be shown from the results in Table 4.12, the apparent better performance of the model using the *ad hoc* synergies was not verified when running a test on all models. This test is the same as the one Section 4.2, *i.e.* count the number of different orientation achieved during an average of 1000 episodes.

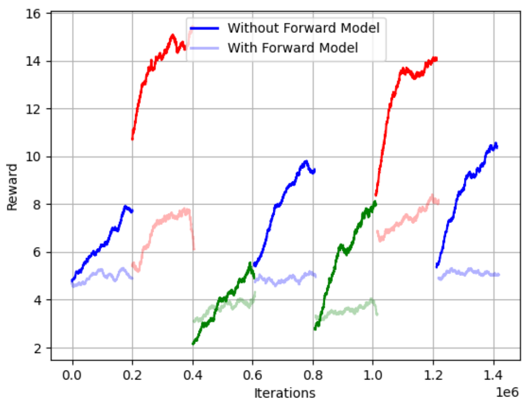
As it can be observed from Table 4.12, the worst model is now the one with the *ad hoc* synergies and the best one is the one using the 18 DOF. It is unclear why the results shown in Figure 4.6 are

Table 4.12: Test results for each of the 3 different models.

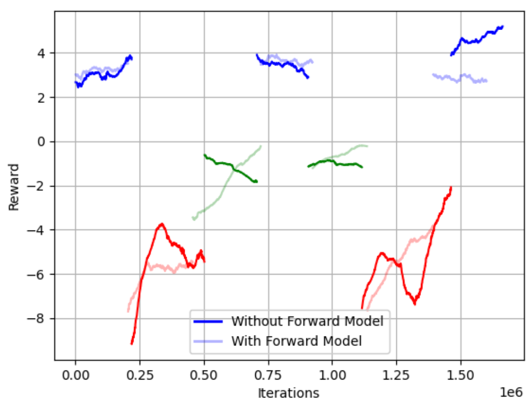
	18 actions	Human synergies	*Ad hoc* synergies
Average different orientations achieved	13.62±3.98	13.14±4.54	8.61±5.22
Maximum number of different orientations in a single episode	28	29	25

not concordant with the ones in Table 4.12. However, a possible reason can be that the model with 18 DOF has more variance, leading to more sparse results. Analysing Table 4.11, it is possible to observe that the model with highest standard deviation in the number of iterations per episode, is the one with 18 DOF. Another possible reason as to why the results are not concordant, is because in fact we are removing dexterity of the hand, specially for the model with the *ad hoc* synergies, so it is normal that the reward is lower. The reason as to why it was so low, could be due to the randomness of the experiment itself.

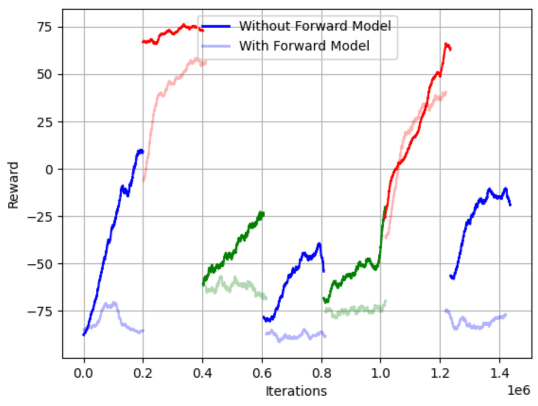
Nonetheless, it is possible to observe that the performance of the model using the human synergies is similar to the performance of the model using the 18 DOF. This result shows that using more DOF does not equal better performance and that using a smaller number of actions, while making using of natural hand synergies leads to as good results as those obtained with all DOF. The reason as to why these human synergies were not used in the experiments above was because this result was only obtained on a later stage of the work, after most of the aforementioned experiments.



(a) Average reward during training for Task 1: task-free object manipulation.



(b) Average reward during training for Task 2: rotating the object.



(c) Average reward during training for Task 3: lifting the object.

Figure 4.5: Reward during training for all three tasks for Experiment 5: Controller versus Forward Model. Solid lines represent the models without the Forward Model and transparent lines represent the models with the Forward Model.

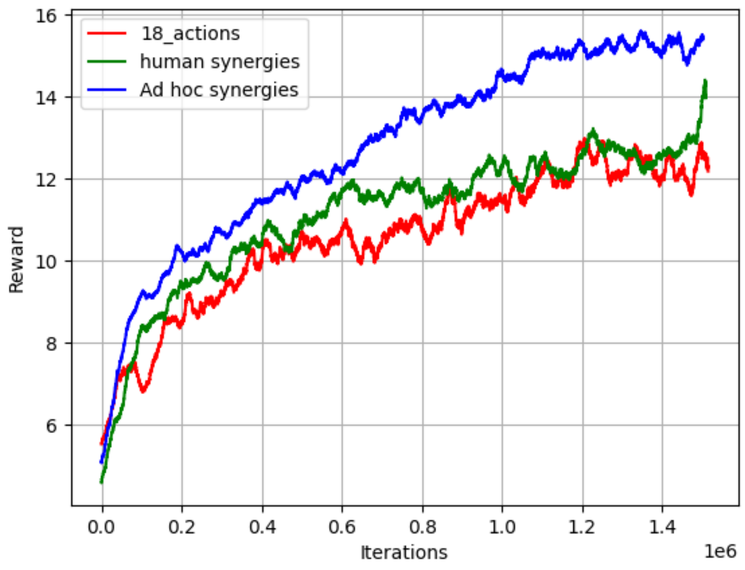


Figure 4.6: Average reward during training for the different models, using Task 1.

5 Conclusion

Contents

There were two main goals for this thesis. The first was related to developing a model, the Controller, capable of using tactile feedback to manipulate a set of objects, based on a set of different tasks. The second goal was to understand if using a representation of the state instead of the actual state would lead to an increase in performance and/or a faster learning/adaptation when training with never before seen objects. The idea behind using a representation is that a certain model, the Forward Model, would be able to encode only the relevant information for the manipulation, disregarding what object was currently being used. To evaluate if both goals were achieved, a set of 3 different tasks and 3 different objects were prepared to test the ability of both the Forward Model and the Controller to adapt to new situations.

Regarding the first objective, the main conclusion is that the Controller was able to learn the proposed tasks, to some degree, except Task 2. Regarding Task 1, the Controller learned but the maximum number of orientations achieved in a single episode were roughly half of the total possible orientations, indicating that the model could have performed better. With Task 3, it is safe to assume that the model learned correctly, but with Task 2, the conclusion is the opposite, the model did not learn properly.

Regarding the second goal, a main experiment (Section 4.5) was conducted. This experiment evaluated the performance of two different models, one using the raw Controller, *i.e.* using the raw input signal, and another using the Forward Model, *i.e.* using the compressed representation of the input. The first conclusion from this experiment is that the model using the Controller is able to perform better (reward-wise) than the model using the Forward Model. This can be explained by the fact that the Controller specializes in performing a particular task, given a certain object, *i.e.* it is overfitting, while the Forward model specializes in having a more general control that is applicable for all objects. Despite having a lower average reward, the models using the Forward Model have, on average, a smaller standard deviation. This leads to the conclusion that the Forward Model is able to provide a more reproducible and reliable control. In other words, the control with the Forward Model is sacrificing performance for reliability. Another important conclusion is regarding the specific case where the Pen is used. This is ultimate test for the Forward Model, because the Pen is the most distinct object from the three. This implies that the control strategy and finger movement for the objects Cube and Egg, is not entirely translatable for the Pen, contrary to what happens between the Cube and Egg, since they have similar shapes. The fact that the performance of the Forward Model when using the Pen is, most of the times, is equal or even greater that the performance with the Controller, leads to the fact that the Forward Model could provide a better result when using object with unique shapes. Moreover, it leads to conclude that the Forward Model was indeed able to generalize to different objects to some degree, achieving the proposed objective of having a more general controller to different objects. However, as indicated before, this generalization comes with the cost of sacrificing performance.

It is important to note that using Deep RL methods to solve a given problem, although providing state of the art results in various fields, are not a very reliable method. Specially when it comes to complex problems such as the one presented here. The work in [58] concludes that a small variance in the hyperparameters or in the architecture of the networks can lead to significantly different results. Moreover, the hand/object interaction is a non-trivial relation, considering the high dimension of the state vector and actions. Also, this type of manipulation is unstable, meaning that a small change in the action can quickly lead to an undesirable state. However, it was shown that is it possible to manipulate a set of object, based on tactile feedback, using Deep RL methods.

5.1 Future work

Although some results were already obtained with this setup, there are possible future improvements that could lead to better results. Starting from the last experiment discussed, if the human synergies were used from the start, the models could have learned better. The reason why they were not used in this work was because this result was only obtained on a later stage. Preliminary results indicate that, by using synergies, namely the human synergies, we are able to reduced in about $^2/_3$ the number of action, while maintaining the same performance as if the full DOF were used. Thus, a good starting point on improving the setup would be to consider the human synergies instead of the *ad hoc* ones.

Another improvement, which may be of greater importance than the one above, is to design a better Forward Model. As it was demonstrated, this Forward Model had some troubles with predicting the next state, specially regarding the touch sensors information. This may have lead to undesired results and a possible solution would be to further adapt the loss function or the architecture of the model itself. Moreover, both PPO and SAC algorithms (which were used to train the models) are Actor-Critc algorithms, meaning that they train 2 networks simultaneously: one to choose the best action (actor) and another to evaluate the expected reward from that action (critic). A possible improvement could be to integrate the Forward Model with the Deep RL algorithm itself, more specifically, with the critic network. This would provide the Deep RL algorithm a better approximation to future states.

Lastly, another future improvement could be related to the actual setup of the experiments, *i.e.* the objects and tasks used. Regarding the objects, the Forward Model could have been even more general if other objects were added or even if the weight of the current object was changed, which was not tested in this thesis. Regarding the tasks, the reward functions could be improved, specially regarding tasks 1 and 2. With Task 1, it was still possible to verify that the models, although learned to perform the task to some degree, could have learned more. but not with Task 2.

Bibliography

[1] *Auto-encoder in biology*, 2019. [Online]. Available: https://mc.ai/auto-encoder-in-biology/

[2] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," 2010.

[3] O. M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray *et al.*, "Learning dexterous in-hand manipulation," *The International Journal of Robotics Research*, vol. 39, no. 1, pp. 3–20, 2020.

[4] Y. Chebotar, O. Kroemer, and J. Peters, "Learning robot tactile sensing for object manipulation," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 3368–3375.

[5] A. Rajeswaran, V. Kumar, A. Gupta, G. Vezzani, J. Schulman, E. Todorov, and S. Levine, "Learning complex dexterous manipulation with deep reinforcement learning and demonstrations," *arXiv preprint arXiv:1709.10087*, 2017.

[6] F. Veiga, J. Peters, and T. Hermans, "Grip stabilization of novel objects using slip prediction," *IEEE transactions on haptics*, vol. 11, no. 4, pp. 531–542, 2018.

[7] A. Bernardino, M. Henriques, N. Hendrich, and J. Zhang, "Precision grasp synergies for dexterous robotic hands," in *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2013, pp. 62–67.

[8] S. Ivaldi, V. Padois, and F. Nori, "Tools for dynamics simulation of robots: a survey based on user feedback," *arXiv preprint arXiv:1402.7050*, 2014.

[9] *Comparison of deep-learning software*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_deep-learning_software

[10] *Handle—Boston Dynamics*, 2019. [Online]. Available: https://www.bostondynamics.com/handle

[11] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Permenter, T. Koolen, P. Marion, and R. Tedrake, "Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot," *Autonomous robots*, vol. 40, no. 3, pp. 429–455, 2016.

[12] R. Szeliski, *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[13] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[14] G. ElKoura and K. Singh, "Handrix: animating the human hand," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 2003, pp. 110–119.

[15] A. Billard and D. Kragic, "Trends and challenges in robot manipulation," *Science*, vol. 364, no. 6446, p. eaat8414, 2019.

[16] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *International Conference on Machine Learning*, 2016, pp. 1329–1338.

[17] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[18] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *International Conference on Machine Learning*, 2016, pp. 1329–1338.

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[20] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1861–1870.

[21] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel *et al.*, "Soft actor-critic algorithms and applications," *arXiv preprint arXiv:1812.05905*, 2018.

[22] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in neural information processing systems*. Citeseer, 2000, pp. 1008–1014.

[23] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[24] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

[25] A. Ng *et al.*, "Sparse autoencoder," *CS294A Lecture notes*, vol. 72, no. 2011, pp. 1–19, 2011.

[26] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[27] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[29] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[30] M. P. De La Bellacasa, "Touching technologies, touching visions. the reclaiming of sensorial experience and the politics of speculative thinking," *Subjectivity*, vol. 28, no. 1, pp. 297–315, 2009.

[31] H. Van Hoof, T. Hermans, G. Neumann, and J. Peters, "Learning robot in-hand manipulation with tactile features," in *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*. IEEE, 2015, pp. 121–127.

[32] H. Van Hoof, J. Peters, and G. Neumann, "Learning of non-parametric control policies with high-dimensional state features," in *Artificial Intelligence and Statistics*, 2015, pp. 995–1003.

[33] A. J. Ijspeert, J. Nakanishi, and S. Schaal, "Movement imitation with nonlinear dynamical systems in humanoid robots," in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)*, vol. 2. IEEE, 2002, pp. 1398–1403.

[34] J. Peters, K. Mulling, and Y. Altun, "Relative entropy policy search," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[35] C. J. Burges, "A tutorial on support vector machines for pattern recognition," *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[36] A. Criminisi, J. Shotton, and E. Konukoglu, "Decision forests for classification, regression, density estimation, manifold learning and semi-supervised learning," *Microsoft Research Cambridge, Tech. Rep. MSRTR-2011-114*, vol. 5, no. 6, p. 12, 2011.

[37] Z. Zhu, X. Wang, S. Bai, C. Yao, and X. Bai, "Deep learning representation using autoencoder for 3d shape retrieval," *Neurocomputing*, vol. 204, pp. 41–50, 2016.

[38] G. E. Hinton, "Deep belief networks," *Scholarpedia*, vol. 4, no. 5, p. 5947, 2009.

[39] H. Van Hoof, N. Chen, M. Karl, P. van der Smagt, and J. Peters, “Stable reinforcement learning with autoencoders for tactile and visual data,” in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 3928–3934.

[40] Z. Yi, R. Calandra, F. Veiga, H. van Hoof, T. Hermans, Y. Zhang, and J. Peters, “Active tactile object exploration with gaussian processes,” in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 4925–4930.

[41] M. C. Gemici and A. Saxena, “Learning haptic representation for manipulating deformable food objects,” in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 638–645.

[42] Z. Wang, “Representation learning for tactile manipulation,” 2018.

[43] B. Boots, S. M. Siddiqi, and G. J. Gordon, “Closing the learning-planning loop with predictive state representations,” *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 954–966, 2011.

[44] J. A. Stork, C. H. Ek, Y. Bekiroglu, and D. Kragic, “Learning predictive state representation for in-hand manipulation,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 3207–3214.

[45] T. Erez, Y. Tassa, and E. Todorov, “Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx,” in *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2015, pp. 4397–4404.

[46] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

[47] E. Rohmer, S. P. Singh, and M. Freese, “V-rep: A versatile and scalable robot simulation framework,” in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2013, pp. 1321–1326.

[48] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *arXiv preprint arXiv:1606.01540*, 2016.

[49] *ShadowRobot Dexterous Hand*, 2005. [Online]. Available: https://www.shadowrobot.com/products/dexterous-hand/

[50] G. Cotugno, K. Althoefer, and T. Nanayakkara, “The role of the thumb: Study of finger motion in grasping and reachability space in human and robotic hands,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, no. 7, pp. 1061–1070, 2016.

[51] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[52] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[53] N. Ketkar, "Introduction to pytorch," in *Deep learning with python*. Springer, 2017, pp. 195–208.

[54] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[55] T. T. D. Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv preprint arXiv:1605.02688*, 2016.

[56] D. M. Wolpert, Z. Ghahramani, and M. I. Jordan, "An internal model for sensorimotor integration," *Science*, vol. 269, no. 5232, pp. 1880–1882, 1995.

[57] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[58] R. Islam, P. Henderson, M. Gomrokchi, and D. Precup, "Reproducibility of benchmarked deep reinforcement learning tasks for continuous control," *arXiv preprint arXiv:1708.04133*, 2017.