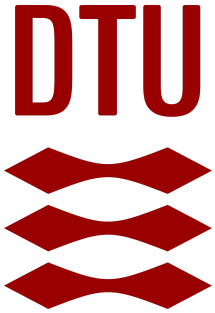
POLITECNICO DI TORINO

MSc Degree in Mechatronic Engineering

Master's Thesis

Closed-loop control of Soft-Robots using Reinforcement Learning



Supervisor
prof. Alessandro Rizzo

Co-Supervisors:
prof. Egidio Falotico
prof. Silvia Tolu

Candidate
Andrea Centurelli
ID: s267132

Academic year 2020 – 2021

Abstract

The field of soft robotics is a relatively new one which has recently developed thanks to the solutions it offers to the limitations of rigid robots: these include the wider possibilities they have in the area of human-robot interaction since their soft and lightweight bodies are intrinsically safe for humans. The biggest effort of the research community has been on the development of innovative materials and designs to build new soft robots but designing controllers that are employable for them is still an open challenge. This is due to the fact that the control and modelling paradigms used on rigid robots are not applicable on soft robots because of their highly nonlinear dynamics which are difficult to model on account of the small degree of stochasticity they often incorporate. These reasons have opened the door to new data-driven controllers based on neural networks which, while being suboptimal for their rigid counterparts, have proven to be effective on soft robots. The main trend right now is to use static controllers that rely on the kinematic models of these manipulators; these have the advantage to be relatively simple to create but their drawbacks are the limitations on the speed and efficiency of the robot. This thesis is an effort to create both open-loop and closed-loop dynamic controllers, with the closed-loop architecture being the main objective, for tracking tasks on a soft robotic manipulator purely based on data. The first method described is based uniquely on data gotten from the robot and its aim is to design an open-loop controller: initially, static data are collected to approximate its inverse kinematic model that in turn is used to generate trajectories in the task space of the robot at a higher frequency; these trajectories will later serve as input for a new network that includes the error caused by the dynamics effects ignored by the kinematics. The second method is instead thought for making a closed-loop controller which makes the tracking of a moving object resistant to changes of the forward model of the robot: a very important feature considering the high variability of the structure of this kind of manipulators which could experience deformations of their pneumatic actuators when put under prolonged actuation stress. Using a state-of-the-art reinforcement learning algorithm (Trust Region Policy Optimization), the controller is trained on an environment which consists of the soft robot's forward dynamic model, previously derived using a Long-short Term Memory recurrent neural network. The two methods proved to be effective: both the open-loop and the closed-loop controllers applied to the real soft robot are able to follow a 3D trajectory with an average error in the cartesian space of below 6mm; the closed-loop controller also proved to be resistant to moderate changes of the forward model of the robot. The work done in this thesis offers quick and easily applicable ways to generate precise tracking using model-free, neural network-based, controllers. The fact that these rely uniquely on data makes the pipelines to obtain them applicable to virtually any soft robot, and the high accuracy with which they achieve dynamic tracking will be paramount for more complex tasks such as grasping and pick-and-place.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Objectives

Soft-robotics is a field that saw a surge in popularity in the past 15 years thanks to their intrinsically safe applications in human-robot collaboration. The differences with rigid robots generate the necessity of developing controllers which are appropriate for these newly thought manipulators; this problem that made way to fairly unconventional control strategies that include the usage of neural networks.

In fact, although their benefits with respect to rigid robots are numerous, one of the most evident downside is the difficulty in modeling both their kinematics and dynamics due to continuous and elastic structures that increase their dimensions drastically.

To overcome the main modelization and control problems of this field, I employed only model-free strategies that include the usage of recurrent neural networks and reinforcement learning, a branch of artificial intelligence that I have been fascinated with and that I studied in the past year. The main goal of these approaches is to achieve performances that are comparable to model-based strategies accompanied by a much easier way to attain them, and the objective of the thesis was to develop a dynamic controller able to fulfill the tracking of a moving object.

I initially focused on doing this with an open-loop controller based on long-short term memory networks, and later closed the loop with the employment of a state of the art reinforcement learning algorithm.

Once I achieved both of these objectives on the first simpler manipulator I used, I focused on finding a way to make the closed-loop controller resistant to disturbances that might incur in the deployment of these robots, and proving that the methods are employable on different soft-manipulators.

1.2 State of the Art

Considering the young age of soft robotics, the research work about it is still relatively scarce. Furthermore, the papers on soft and continuum robots are largely focused on the structural designs of these manipulators, trying to employ new materials to implement properties such as inherent compliance, variable stiffness, and higher dexterity in tough

environments.

If these developments are paramount for their application in industrial, service, and assistive scenarios, an even more important role will be played by the development of kinematic and dynamic controllers that guarantee accurate and reliable control.

This is an extremely challenging task due to the non-linearity of the materials that soft manipulators are build with, which generate phenomena like bending, extension, contraction, torsion, and buckling of the bodies, and give rise to essentially infinite degree-of-freedom motions.

The work on controls for soft and continuum robots can be divided in two ways: based on whether they rely or not on individuating the parameters of the models (*model-based* or *model-free*), and on the steady-state assumption, or lack thereof (*kinematic* or *dynamic*).

- *Model-based kinematic controllers* presently are the most used and investigated ones.

  The most common way to model the kinematics of such robots is through constant curvature (CC) [2] [3]. This is due to the low computational complexity of this strategy that allows to extrapolate a very good approximation of the manipulators' model. The limitations of the CC approach arise when the robot's body is non-symmetric and when torsional effects arise.
  More complex modeling approaches have been explored, such as :

  - piecewise constant curvature (PCC) models [4], an evolution of CC for multi-sectional manipulators
  - beam theory [5], which include large-deflections dynamics and axial extensibility to achieve accurate setpoint tracking
  - Cosserat rod theory [6] [7], an advanced mathematical approach to the modelization of complex continuum and nonlinear bodies.

  Even though these methods are theoretically more sound and should guarantee a better accuracy, they're oftentimes computationally burdening. Other downsides are that they're platform specific and, considering that most of them require feedback from the configuration space, they demand lot of sensory data (especially in the case of pneumatic actuation)

- *Model-free kinematic controller*. The main advantage over their *model-based* counterpart is that they don't require to define the parameters that map the task space to the operating space of choice (joint, configuration, or actuation space). This task can in fact be burdening and sometimes mathematically challenging for certain non-linear, nonuniform, and low-budget robot. It is on these types of soft manipulators that the *model-free* solution fares equally well or often better. The downside is that it is impossible to establish convergence of the controller or even a stability analysis due to the lack of a clearly parameterized model.
  For the approximation of the model, more recent research tend to favor neural networks (such as in [8] and [9]) where the authors successfully employ a neural network to approximate either the global or the local differential inverse kinematics.

- *Model-based dynamic controllers*: moving to a non-steady-state approach, the challenges posed by the high dimensionality of these robots become even more evident.

Dynamic controller are still in their nascent stage, and for this reason the research on this topic is still sparse. Most of the work done until now stems from the same constant curvature methods used for the kinematic counterparts of these controllers.

Notably, a PD torque controller was applied in [10], while a sliding mode controller was employed in [11].

- *Model-free dynamic controllers.* Lastly, the most unexplored type of controller in the list: these methods have clear advantages, such as being a relatively simpler path to develop a dynamic model and being virtually platform-independent, but also clear disadvantages, like the training time and the impossibility of reaching the stability that a classic controller guarantees.

  Very little has been done in this area, two of the most impressive ones are by the same author: in the first one [12] sequential quadratic programming is used to produce a learning based-open loop dynamic controller, and in the second one [13] a NARX (nonlinear autoregressive exogenous model) is used to learn from a dataset of trajectories produced with the same method as in the first paper. Considering that the optimization does not allow the loop to be closed because of the computational expensiveness, training a model (or a network) to do the same task is a valuable workaround to the problem.

Among the most valuable references I underline the work done in [9], in which the author uses a two-module version of the first manipulator I used (described in section 2.1), and very similar to the second manipulator described in section 2.2, although controlled through cable actuations instead of pneumatic ones as did in this thesis. In this paper the author develops a learning-based closed-loop kinematic controller that achieves a position error of about $10mm \pm 5mm$.

A second paper developed using the same platform used in my experiments, although only one of the two modules was pneumatically actuated while the proximal one was left passive, is [12]. The open-loop dynamic controller presented here achieves a tracking error of about 50mm for a circle and 21mm for an infinity symbol, which are the same benchmark I used to test the goodness of the controller developed by me.

Lastly, the work that is more closely related to the one presented in this thesis is [13], in which a closed-loop dynamic controller is learned using a technique that merges aspects of supervised and unsupervised learning. Although this paper served as source of inspiration for the research I did, the results obtained regard exclusively tasks of point-reaching, which are hard to compare with the point-to-point tracking task that I explored.

For a more detailed outline of the state of the art of this field, I recommend this valuable survey ([1]).

1.3 Organization of the Thesis

This is a brief overview of the chapters' content:
Chapter 1 explains the thesis' objectives and summarization of the methodologies used.
Chapter 2 presents the soft robots used during the thesis development, with a detailed explanation of the fabrication and control mechanism.

Chapter 3 offers an overview of machine learning fundamentals and introduces the networks used. These are Artificial Neural Networks (ANN), Recurrent Neural Networks (RNN), and the more advanced versions of them which are applied to both the methodologies presented: Long-Short Term Memory (LSTM) networks.
Chapter 4 prepares the reader to the main fulcrum of the thesis, a novel artificial intelligence paradigm called Reinforcement Learning (RL). After an initial outline that presents the principal components of classical RL, including also the specific vernacular used, this chapter moves on to the more current branch of RL called Deep Reinforcement Learning (DRL). A few words are spent to establish the differences it has with RL, and then the method used in the thesis, the Trust Region Policy Optimization (TRPO) algorithm, is analyzed in great detail.
Chapter 5 is about the first of the two methodologies developed by me. It is aimed at the production of a model-free, neural network-based, open-loop controller. This chapter was developed in the first months of work, and improved in the later ones, to provide a way to allow accurate control of soft robots, no matter what the actual platform used is. A thorough description of the process is provided, in chronological order, as well as the results obtained.
Chapter 6 is instead about the second methodology. This one has the goal of creating a closed-loop dynamic controller, again model-free, trained using TRPO, the deep reinforcement learning algorithm explained in chapter 4. Once more, the process is outlined in chronological order and the results are provided at the end.
Chapter 7 concludes the thesis with my considerations over the methods I developed and their future potential.

Chapter 2

Soft-robots

The branch of soft robotics started developing in 2009; the main drive of its pioneering stages was the observation of how biological beings move in complex and unpredictable environments thanks to structural paradigms that differ substantially from the ones used in the conventional rigid robotics. In fact, if rigid robots are extremely capable of performing fast and precise tasks thanks to their metallic composition and their electromagnetic components, they inherently provoke issues when handling unexpected environmental changes or interactions with humans. The scope of soft robots is to solve these congenital faults of their rigid counterparts not with complex control systems but with the morphology and material properties of their bodies.

Considering the differences between rigid and soft robots, I find it necessary to redefine, where possible, the operating spaces for continuum as did in [1]:

Table 2.1: Definition of the operating spaces

Operating Space	Definition	Pneumatic	Tendon-driven
Actuator space	$\tau \in \mathbb{R}^k$	Pressure applied to the chambers	Motor positon or torques
Joint space	$\varsigma \in \mathbb{R}^l$	Hard to define possibly the volume in the chambers	Cable length or tension
Configuration space	$\zeta \in \mathbb{R}^m$	The independent physiscal parameters that define the configuration of the manipulator	
Task space	$x \in \mathbb{R}^n$	Position and/or pose of the end-effector	

During the development of this thesis I used two distinct soft-robots to demonstrate the adaptability of the methods presented to virtually all robotic platforms. Said soft-robots have intrinsic structural similarities but their actuation-space and task-space have different dimensions.

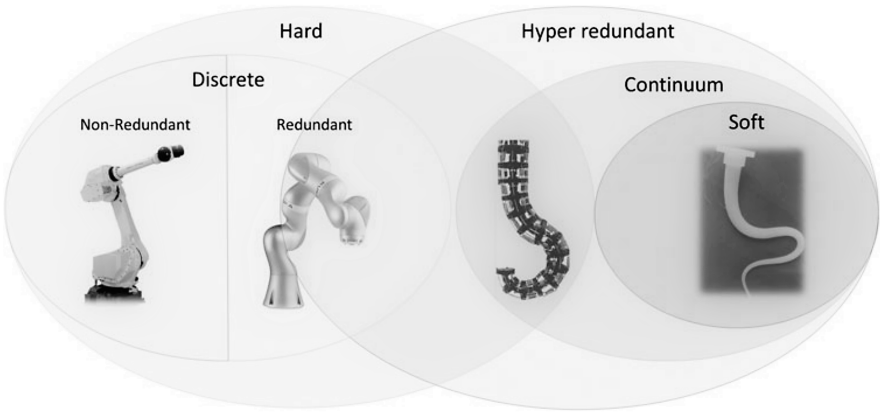


Figure 2.1: Difference between rigid-linked robots and hyperredundant robots, which enclose continuum robots and soft robots. Figure curtesy of [1]

2.1 I-Support

The first soft robotic manipulator is made of only one module, composed by three pairs of McKibben-based actuators and three cables which are alternately displaced at an angle of 60° along a circle with a radius of 3 cm, and kept together with a set of perforated plastic rings. The total length of the manipulator is $\sim$ 20 cm while the total weight is 160 g. Considering that the cable and pneumatic actuators are decoupled, in all the experiments conducted I used uniquely the pneumatic ones.

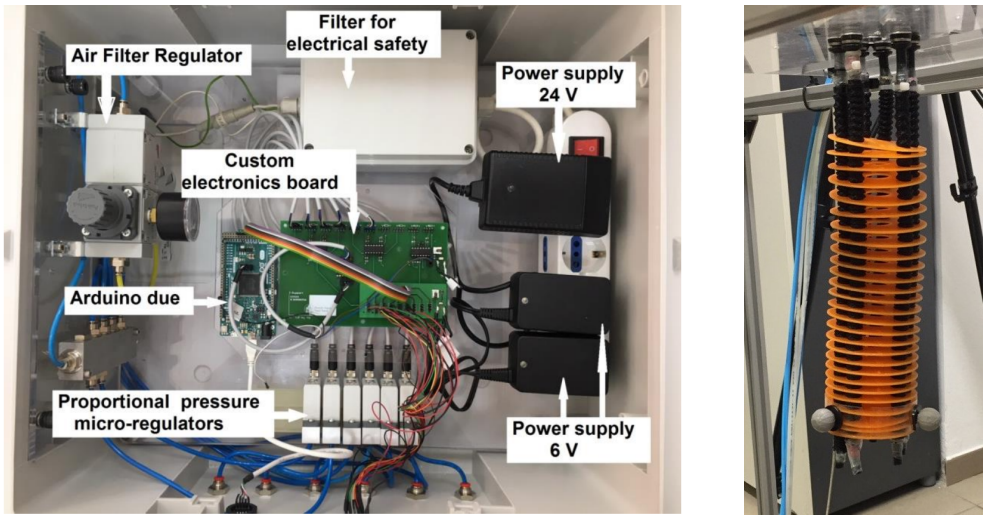
The pneumatic chambers are McKibben actuators, a type of *pneumatic artificial muscles* (PAMs), each composed of:

- An external chamber, consisting of a polyester braided sheath with a diameter of 3 mm. Said sheath is deformed through the application of a mechanical compression to a bellow shape, which is then memorized via the application of cyclical uniform heat
- An internal chamber, consisting of a latex balloon. The internal chamber is fit inside the external one and then sealed using Parafilm (a semi-transparent flexible film)

The insertion of the internal chamber in the external one allows it to expand only longitudinally and not radially, producing an elongation of the actuator whenever a pressure is applied to it.

The robot is actuated through an actuation box (as seen in Fig.2.2a) made up by three parts:

- The pneumatic actuators: constituted by 3 proportional pressure micro-regulators which map a voltage in the range of 0-10V to a pressure in the range of 0-3 bar, and 1 air filter regulator to set the pressure input at 4bar.
- The control unit: composed by an Arduino Due and a custom electronics board that includes a DAC and an amplifier to manage the pressure micro-regulators.



(a) Maximum elongation (b) Maximum bending

Figure 2.2: Actuation results

- The power supply: 24V for the micro regulators, and 6V for the custom electronic board

To input the actuations to the manipulator, the Arduino is interfaced with MATLAB through a serial communication. The inputs are given from MATLAB in the form of digits from 0 to 255, which are then converted to Volts (from 0-5V) to be fed to the pressure regulators that hold a range of 0-1.5bar.

2.2 AM-I-Support

This evolution of the previously presented I-Support [14] is composed by two identical modules, called proximal and distal, connected to each other through nuts and bolts to allow complex movements and a larger task-space. Like in its previous version, each AM-I-Support module is actuated only through its three pneumatic chambers, even if the manipulator allows more degrees of actuation through cables pulled by three DC motors.

This second robot is fabricated following a *Design for Additive Manufacturing* (DfAM) approach: the materials used to 3D print the manipulators are two:

- A thermoplastic polyurethane with Shore A hardness equal to 80 (TPU 80A LF), for the pneumatic actuator chambers and the rings
- Polylactic acid for the terminals

The choice of TPU 80A LF allows large elongations and deformations thanks to its tensile module of 17 MPa and elongation at break of 471%, but also thanks to the bellow-like shape of the chambers

Each module has a resting length L_0 of about 20 cm and an activation range that goes from 0 to 4 bar. When actuating all the chambers at once with 4 bar, the module reaches

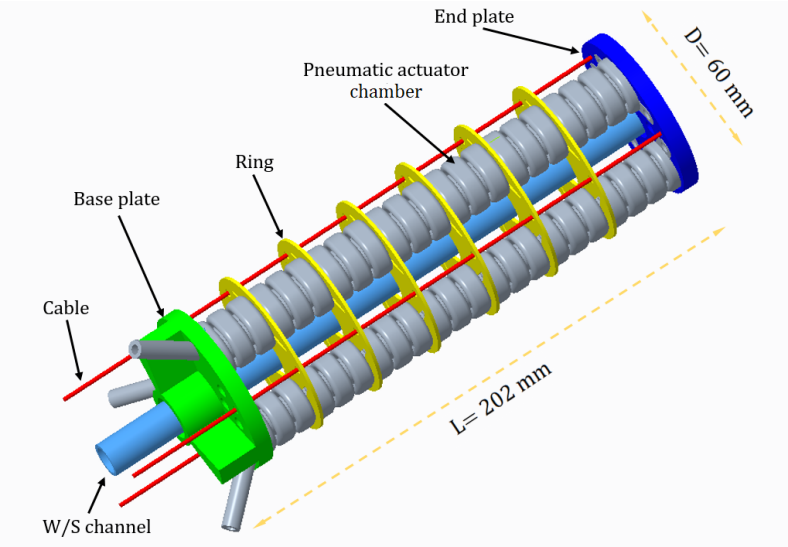
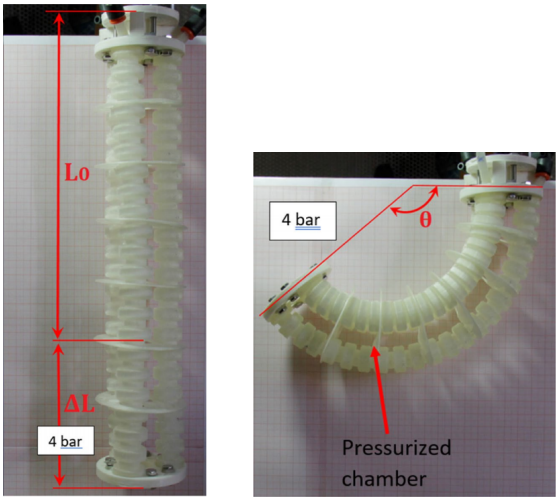


Figure 2.3: AM-I-Support 3D design (Figure courtesy of [14])



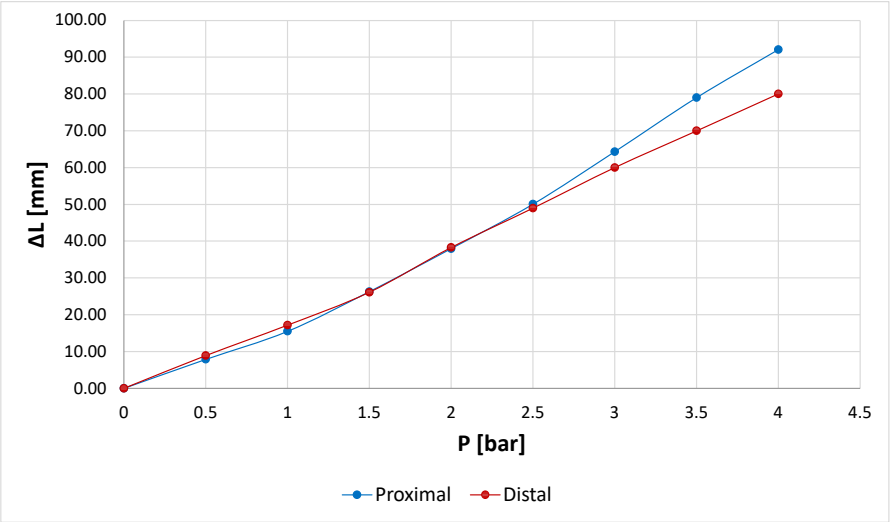
(a) Maximum elongation

(b) Maximum bending

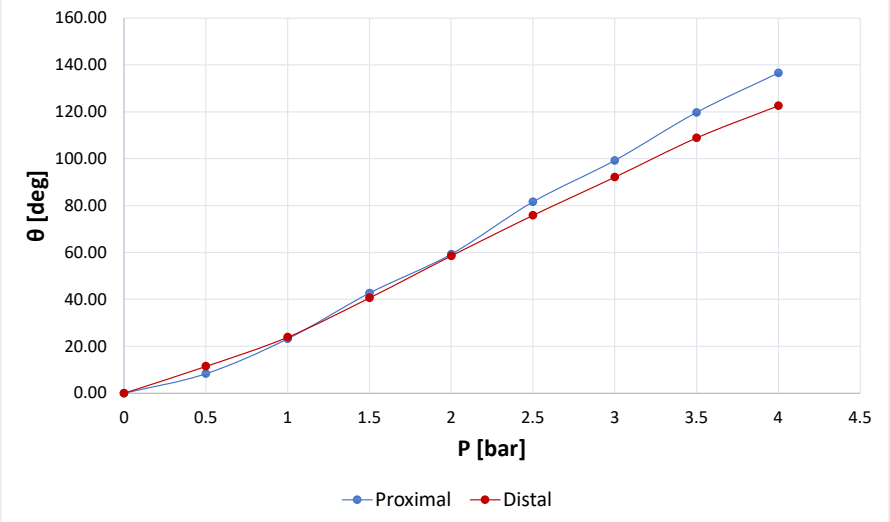
Figure 2.4: Actuation results (Figure courtesy of [14])

its maximum elongation $\Delta L_{max} \approx 47\% L_0$ as seen in Fig. 2.4a, while actuating only one chamber at 4 bar achieves maximum bending, with an angle $\theta_{max} \approx 137°$ as seen in Fig. 2.4b.

Even considering that the fabrication method employed for this manipulator is thought for repeatability, the two modules employed have slightly different values of elongation ΔL and bending angle θ for the same values of pressure input in the pneumatic chambers, as can be seen in Figures 2.5.



(a) Maximum elongation



(b) Maximum bending

Figure 2.5: Elongation and bending values for the two modules (Figure courtesy of [14])

Chapter 3

Neural Networks

In this chapter I will describe the neural networks that I used in the methods presented in the thesis. These are Artificial Neural Networks (ANN), and Long-short Term Memory (LSTM) Recurrent Neural Networks (RNN).

Most of the formulas in this chapter are taken from two free Deep Learning books: [15] by Ian Goodfellow and Yoshua Bengio, and [16] by Michael Nielsen.

3.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computing systems that take inspiration in their structure from the brains of biological brains.

Brief History of Neural Networks

Their birth can be traced back to 1943 when neurophysiologist Warren McCulloch and mathematician Walter Pitts created a simple electrical circuit to implement a computational model based on threshold logic [17].

Their work was then expanded on by psychologist Frank Rosenblatt, which was the first to come up with the idea of the Perceptron back in 1958 [18], while trying to quantify the decision making processes of flies.

The Perceptron was applied to make the first networks that solved real world problems, provoking a surge in popularity of this new technology. That lasted until 1969 when Marvin Minsky published his book "Perceptrons" [19] in which he concluded that Rosenblatt's approach couldn't be employed in large scale neural networks because of the limited computing power that computers had back then, and that perceptrons were unable to compute the exclusive-or (xor) logical operation, making them useful only to learn how to separate linearly separable classes, but not non-linearly separated ones (in fact the first usages of NNs were used for classifications, not regressions).

Minsky's work brought many institutions to deny funds for Artificial Intelligence (AI) research, causing what's known as "the AI winter", a period that lasted about 15 years and in which the progress in this field slowed down dramatically.

The next huge step in machine learning was taken in the 1980s when backpropagation, a method that alongside gradient descent makes the backbone of the NNs' learning, was brought back to light after digging in the past decades' literature [20].

After the basic structure of ANNs had been laid down, the work starting from the 1990s, higly regarded as the golden era of machine learning discoveries, until the present day brought neural networks to be used in an immeasurable variety of fields.

Structure of Artificial Neural Networks

Every ANN is composed by neurons which are vertically stacked to create the so called *layers*. With reference to Figure 3.1 he layers can be of three types:

- Input layer: it's the first layer of the network, it directly connects with the input signals.
- Hidden layers: these are layers of neurons which are connected neither with inputs nor outputs, but only with neurons of other layers. There can be as many hidden layers as the user wants.
- Output layer: the last layer of the network, it connects with the last hidden layer and its output is the final output of the network.

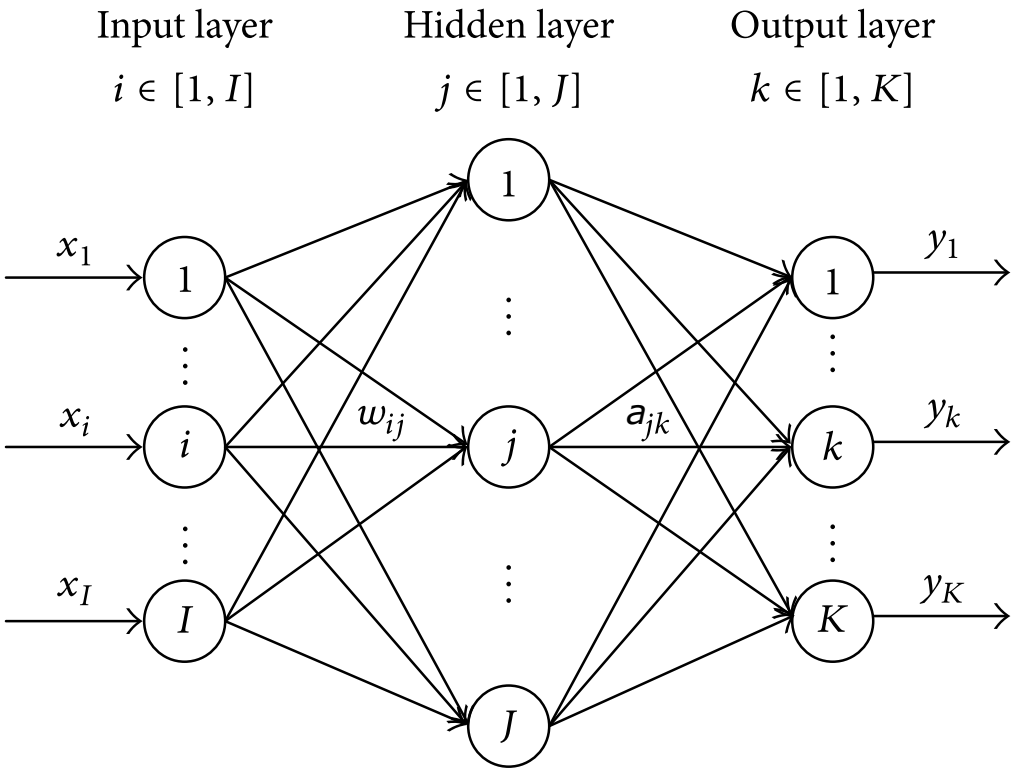


Figure 3.1: Artificial Neural Network scheme

Every j^{th} neuron of every l layer has outputs a_j^l, generally called *activations*, and inputs z_j^l, might they be the input signals provided by the user or the activations coming from previous neurons.

Each neuron of each layer is connected with all the neurons of the next layer (except the output layer), and each one of these connections can be imagined to carry a weight w_{jk}^l that should be read as "the weight that the output of neuron k of the $(l-1)^{th}$ layer has on neuron j's input, with neuron j belonging to the l^{th} layer". In fact, every neuron is essentially comprised of an *activation function* σ which in the original perceptron was a step function with discrete output 0 or 1 and later took more complex (see the next subsection). The activation of each neuron is thus the output of its activation function:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

where the sum is over all neurons k in the previous layer $(l-1)^{th}$. Every neuron has a bias b_j^l, which alongside the weights form the so called *parameters* of the network.

Activation Functions

The activation functions for the neurons of a neural network can be many; the first one historically was the step (or binary) function:

$$\sigma(z) = f(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

with z being the generic input to any neuron. This function has two big drawbacks: it can only produce a binary output, meaning that the neuron will be either *off* or *fired up*, and the function is not differentiable in 0. The motivation for this last drawback will be clearer in the next subsection about backpropagation.

Considering the abundance of activation functions, I'll only report the 4 I used or quoted during the thesis:

- Sigmoid: this is the function that historically resolved the problems posed by the step function. Its output varies smoothly with the input x provided, and saturates at 0 and +1 for values of $x \to -\inf$ and $x \to +\inf$ respectively.

$$\sigma(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$

- Hyperbolic tangent: it is very similar to the sigmoid function in shape, the differences in performance between the two are not always easy to determine.

$$\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

  The reason why *tanh* is practically better is explained in the backpropagation subsection

- Rectified Linear Unit (ReLU): one of the most widely used functions in machine learning. Its strength rely in its simplicity, in fact it turns out that silencing the neurons for values that are unimportant to the model is an excellent way to improve the rejection of outliers. Furthermore, it is much more computationally advantageous to use ReLU with respect to a *sigmoid* or *tanh* function, both for calculating its output and for obtaining its derivative.

$$\sigma(x) = f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} = \max(0, x)$$

- Linear: this is the simplest activation function. It is used uniquely for the output layer as it would be ineffective for the hidden ones:

$$\sigma(x) = f(x) = ax$$

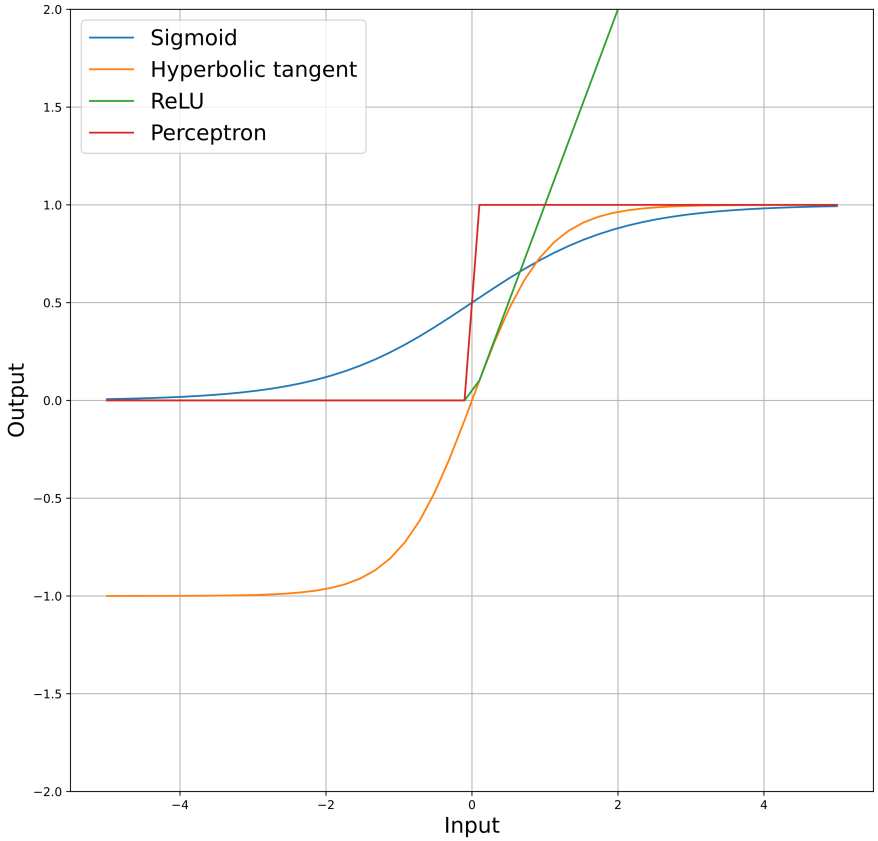


Figure 3.2: Activation functions

Gradient Descent and Backpropagation

The first step towards the optimization of a network's parameters (weights w_{ij} and biases b_j) is to define a *cost function*. A cost function can be any function that is differentiable to the first order, but the most widely used (and the one used in this thesis), is the Mean Squared Error:

$$C(w,b) = \frac{1}{2n}\sum_x \|y(x) - a\|^2$$

where the 2 at the denominator is introduced only for mathematical convenience. In this cost function $y(x)$ is the desired output of the networks (the ground truth or labels), whereas a is the actual output of the network. The better the network is, the closer the estimated values will be to the desired ones, an thus the smaller the cost function will be.

This lays the foundation for a minimization problem: in fact, the cost function depends on the weights and biases of the network. This means that the update of the parameters should follow the direction of steepest descent, given by the partial derivative of the cost function for that given parameter:

$$w_j \to w'_j = w_j - \eta\frac{\partial C}{\partial w_j}$$
$$b_j \to b'_j = b_j - \eta\frac{\partial C}{\partial b_j}$$

where η is defined as *learning rate*, and is responsible for the speed of descent towards the minima of the cost function.

To get to the value of the partial derivatives $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$ it's necessary to follow these steps (taken from chapter 2 of [16]):

1. Calculate the error for the output layer:

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'\left(z^L_j\right)$$

   this equation is made by the partial derivative $\frac{\partial C}{\partial a^L_j}$, which measures the rate of change of the cost function w.r.t. the j^{th} output activation, and the derivative $\sigma'\left(z^L_j\right)$, which in turn measures the rate of change of the activation function at z^L_j.

   In matrix form it becomes:

$$\delta^L = \nabla_a C \odot \sigma'\left(z^L\right)$$

2. Calculate the layer's error δ^l as a function of the next layer's error δ^{l+1}:

$$\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(z^l\right)$$

   where w^{l+1} contain all the weights of the $(l+1)^{th}$ layer.

   This equation is extremely important because it allows to propagate the error backward through the network, which is the scope of backpropagation.

3. Calculate the rate of change of the cost with respect to the network's biases:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

4. Calculate the rate of change of the cost with respect to the network's weights:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$$

By repeating step 2 for all the L layers of the network, and for each of them update the layers' parameters using step 3 and 4, we get a backpropagation pass.

Having the values of $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$ does not mean that the weights and biases are updated at every pass; in fact, the technique used here is not properly gradient descent but *stochastic gradient descent*, which essentially means that the parameters are updated once every m backward passes:

$$w^l \to w^l - \frac{\eta}{m}\sum_x \delta^{x,l}\left(a^{x,l-1}\right)^T$$
$$b^l \to b^l - \frac{\eta}{m}\sum_x \delta^{x,l}$$

Feature Scaling

A common practice is to normalize the inputs, or *features*, of the neural network using a technique called *feature scaling* (or more commonly *normalization*).

Even if the network is theoretically able to approximate any function by virtue of the universal approximation theorem [21], having data in the input belonging to significantly different ranges, might lead to an inefficient training or, in some cases, failure to localize the global optima.

The reasons behind normalization are practical: it prevents the algorithm from getting stuck in local optima and makes the training faster.

The three most prevalent methods to do so are:

- *Rescaling* also called min-max normalization, scales the data between 0 and 1:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

  where x' is the normalized value and x the original one. The range of the normalized data can also be between two value sa and b chosen by the user:

$$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$$

- *Mean normalization*:

$$x' = \frac{x - \bar{x}}{\max(x) - \min(x)}$$

  with $\bar{x}$ being the mean value of the data to be normalized.

- *Z-score normalization* most commonly known as *standardization*. The result of this type of feature scaling is having the values of each feature (input type) in the data have zero-mean and unit-variance:

$$x' = \frac{x - \mu(x)}{\sigma(x)}$$

with $\mu(x)$ being the mean of feature x and $\sigma(x)$ being its standard deviation.

3.2 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a type of Artificial Neural Network that is built specifically to have as inputs a time-series or, more generally, any type of sequential data.

Unlike a classic feedforward neural network, a RNN has a hidden state (also referred to as *memory*) which allows it to utilize previous outputs (or for more complicated architectures, fractions of them) alongside the current input to predict the final output of the network.

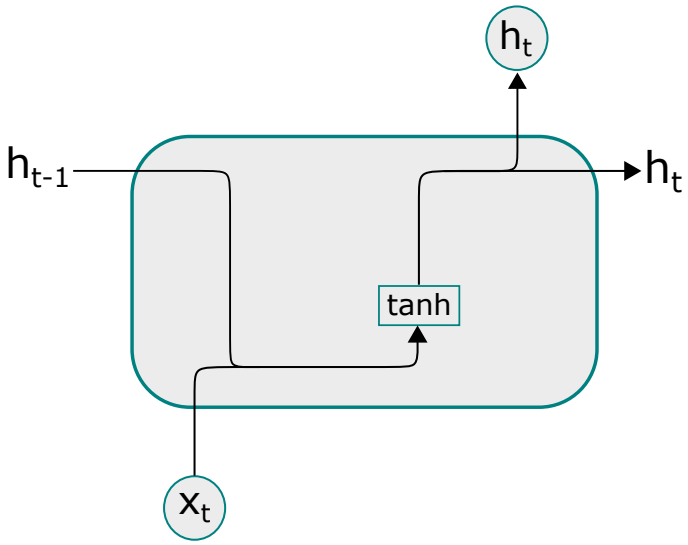


Figure 3.3: RNN cell

The original and simplest version of a RNN cell is shown in Figure 3.3. The input x_t is generally speaking a mono-dimensional array of dimension N (usually referred to as *number of features*), while the *hidden state* h_t has a dimension M (usually referred to as *number of units*) specified by the user as a hyperparameter. By considering the RNN cell as frozen in time, it can be seen that it is basically a feedforward neural network with M neurons activated through a hyperbolic tangent (tanh) function; the input of said network is the concatenation between the features x_t and the past hidden state h_{t-1}, which results in an array of dimension $N + M$. Consequentially, the RNN cell will have $(N + M) \cdot M$ weights and M biases, for a total of $M^2 + NM + M$ parameters.

A cell is the fundamental unit of a Recurrent Neural Network: depending on the type of data the network has to train on, the user decides a number T of timesteps to include

in the input, which will have dimension TxN; in Figure 3.4 I show how multiple cells, in this case three, are connected.

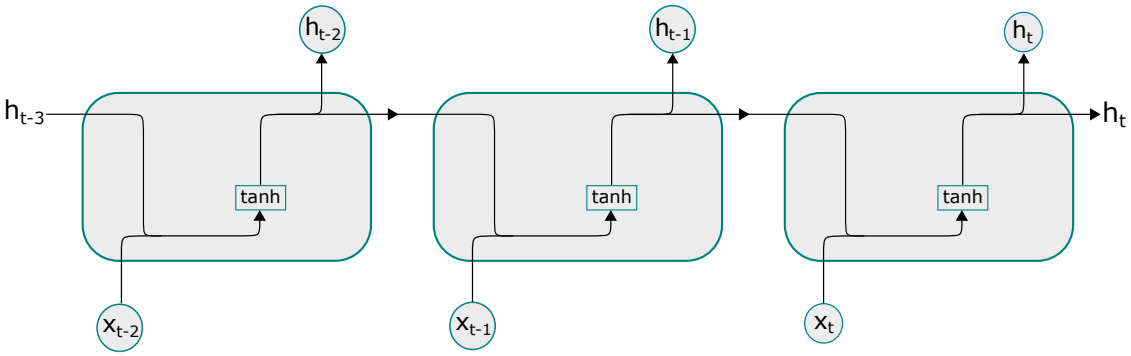


Figure 3.4: RNN cell connections

The total number of parameters of the whole network will be $T(M^2 + NM + M)$.

The output of an RNN layer coincides with a subset of the hidden spaces $[h_{t-T+1}, ..., h_t]$; in case of a deep recurrent neural network, multiple RNN layers are stacked on top of each other so that the input for the next layer becomes the previous layer's output.

In case of a shallow RNN, the subset of hidden states (generally just the last one h_t) passes through an output layer whose output dimension is the one desired and whose activation is usually *Softmax*, if the domain is discrete, or *linear*, if the domain is continuous.

These "vanilla" RNNs are nowadays outdated, mostly because of the *vanishing gradient problem*. This phenomenon occurs during the backpropagation pass to update the weigths of the network: referencing Figure 3.5 the derivative of the error E_3, which is a function of the output Y_3, with respect to the hidden state's weight is by virtue of the chain rule dependent on all the previous states:

$$\begin{gathered}\frac{\partial E_3}{\partial W_h} = \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial h_3} \cdot \frac{\partial h_3}{\partial W_h}\right) + \\ \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial h_3} \cdot \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial W_h}\right) + \\ \left(\frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial h_3} \cdot \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial W_h}\right)\end{gathered}$$

more generally, for a RNN with N cells, the gradient will be:

$$\frac{\partial E_N}{\partial W_h} = \sum_{k=0}^{N} \frac{\partial E_N}{\partial Y_N} \frac{\partial Y_N}{\partial h_N} \left(\prod_{j=k+1}^{N} \frac{\partial h_j}{\partial h_{j-1}}\right) \frac{\partial h_k}{\partial W_h}$$

Considering that the hidden states are bounded between -1 and +1 by the *tanh* activation function and hence their derivatives are bounded between 0 and 1, the product in the middle of the above formula becomes smaller as the hidden states get further back in the past with respect to the output. This leads the gradient to become very small, hence the name *vanishing gradient*, and the update of the weights ineffective. It has been proven that this type of RNN becomes inadequate for networks of more than 6-8 cells: this can be summarized by saying that "vanilla" RNNs inherently forget information too far back in the past.

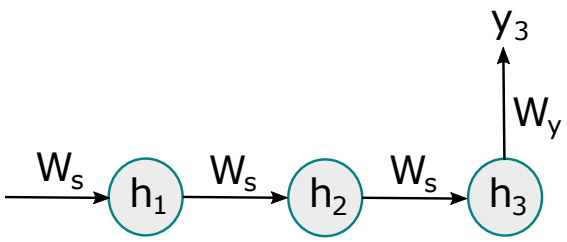


Figure 3.5: Simplified backpropagation scheme

3.3 Long-Short Term Memory Network

The solution to the vanishing gradient problem is to modify the structure of the cell.

A new cell architecture, called Long-Short Term Memory (LSTM), was presented in [22]; it introduces a more complex layout that grants the network the ability to withhold in its state only valuable informations from the past states.

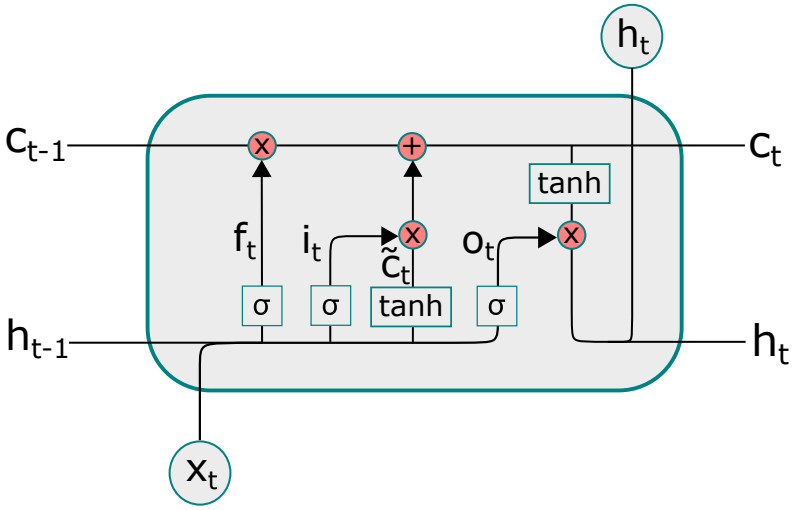


Figure 3.6: LSTM cell

The first new feature to discuss is the introduction of a cell state C_t, which contains informations related to the long term memory; the cell state is updated in two points of the cell:

1. The first one is through the red multiplication gate that we see in Figure 3.6: the array composed by the concatenation of the input x_t and past hidden state h_{t-1} passes through a *sigmoid* layer (generally called *forget gate*), which produces as output an array f_t defined as follows:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

This array f_t will determine which information of the cell state C_t have to be forgotten and by which degree, by "masking" it through element-wise multiplication.

2. The second one is through summation of the new array to be stored: the array resulting from the concatenation of x_t and h_t passes first through another *sigmoid* layer (called *input gate*) that is responsible for which values of $\tilde{C}_t$ are going to be added to the cell state; $\tilde{C}_t$ is, in turn, the output of a *hyperbolic tangent* layer that generates a vector of candidate values.

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$
$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right)$$

Lastly, the new cell state is obtained by summing the old state, multiplied by the forget array, with the new candidate state, multiplied by the input gate:

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t$$

Once the cell state is well defined, the hidden state is nothing else but a "filtered version" of it: the cell state passes through a *hyperbolic tangent* layer and is then multiplied by the output of the *output layer* o_t, as follows:

$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t \cdot \tanh\left(C_t\right)$$

By stacking more than one LSTM cells in parallel we get to a LSTM network that looks like the one in Figure 3.7. Considerations similar to the ones made for the RNN parameters can be made for the LSTM network, having 4 layers inside a cell, the total number of parameters will be: $4T(M^2 + NM + M)$.

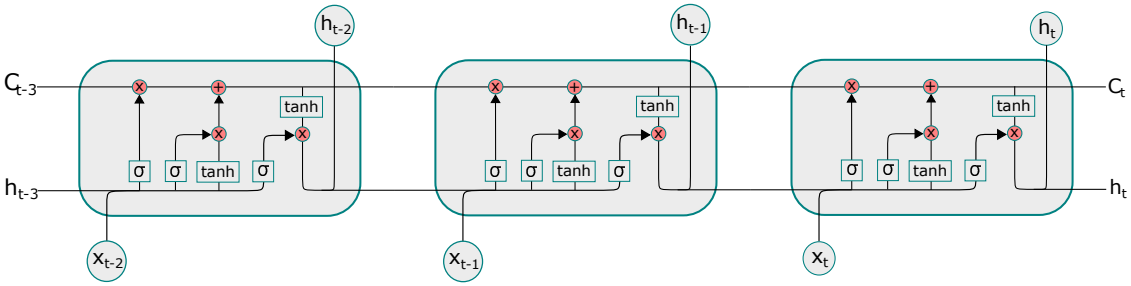


Figure 3.7: LSTM cell connections

Chapter 4

Deep Reinforcement Learning

4.1 Introduction to Reinfocement Learning

Reinforcement Learning (RL) is a subfield of machine learning whose first stages can be tracked down to the early 1980s, and that has seen an enormous surge in popularity in the last ten years thanks to the promising results shown when applying its techniques to videogames. The essential elements of RL are:

- An *agent*: it's the "learner" of the process, the entity that is trained to fulfill a specific set of tasks.
- An *environment*: it's the world in which the agent is located.
- *Actions*: they're steps that the agent takes in the environment and that result in a change of state of said agent.
- *Rewards*: they're "prizes" or "punishments" given to the agent based on how good the last action taken was, based on the task to be carried out.
- *Observations*: what the agent can see of the environment at every time-step: they are pieces of information about the current state of the agent.

The goal of RL is to learn an optimal policy, which is the mapping between the states and actions, that maximizes the expected sum of rewards received during a trial (generally called a rollout). By putting all these elements together a *closed-loop* training process is obtained: the agent gets an observation from the environment, the state changes and a reward which is a function of it is returned to the agent. This training process consists in a trial and error procedure in which the agent balances the *exploration* of the environment with the *exploitation* of the policy being trained: this is of fundamental importance because an agent that is too "greedy" and chooses the best action based on the current policy will fail to explore the environment, and thus to find the global optimum (or more generally speaking, a better local minimum).

Markov Decision Process

The Markov decision process (MDP) provides a mathematical framework which includes the elements of an environment. A MDP is based on the Markov property, which says that each state should be dependent only on the state and action immediately prior to it; a MDP is a tuple $(\mathbb{S}, \mathbb{A}, \mathcal{P}, \mathbf{R}, \gamma)$ where $\mathbb{S}$ is the state space, $\mathbb{A}$ is the action space, $\mathcal{P}$ is the state probability function:

$$\mathcal{P}^a_{ss'} = \mathbb{P}\left[s_{t+1} = s' \mid s_t = s, a_t = a\right]$$

$\mathbf{R}$ is the reward function:

$$\mathcal{R}^a_s = \mathbb{E}\left[r_{t+1} \mid s_t = s, a_t = a\right]$$

where $s \in \mathbb{S}$ and $a \in \mathbb{A}$ and γ is a discount factor bound between 0 and 1 : $\gamma \in [0,1)$

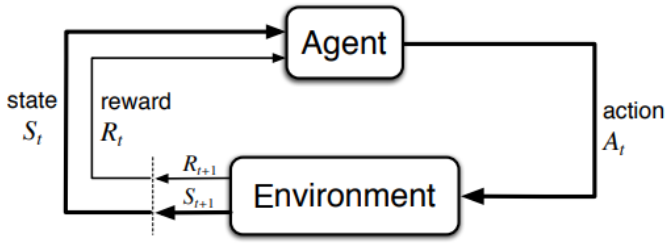


Figure 4.1: Markov decision process

Rewards and Policy

As said earlier in the introduction, the goal of the agent is to maximize the rewards during a rollout. To be more precise, the goal η_t to be maximized is the cumulative sum of the discounted rewards (also named *return*) obtained during the rollout :

$$\eta_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$$

where T is the length of the rollout in time-steps. Common values of γ in RL fall in the range [0.9,1), given that a value proximal to 0 will make the agent blind to the expected rewards that will come in the future, and a value of 1 will prevent the algorithm from converging to a solution for values of $T \to \infty$.

The policy is substantially the agent's strategy: it's a probability distribution over actions given states: $\pi(a|s)$, or in other words, the likelihood of every action when an agent is in a defined state.

State-Value and Action-State Functions

The state-value function conveys how good each state $s \in \mathbb{S}$ is by calculating the expectation of the discounted sum of rewards:

$$v_\pi(s) = \mathbb{E}_\pi \left[\eta_t \mid s_t = s\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\mathrm{T}} \gamma^k r_{t+k+1} \mid s_t = s\right]$$

where the notation $\mathbb{E}_\pi [\cdot]$ is the expected value given the agent's policy.

The action-state function is quite similar: it's still the expectation under the agent's policy over the sum of rewards, but given the action alongside the state:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[\eta_t \mid s_t = s, a_t = a\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right]$$

Having defined these two functions, it is possible to outline the concept of optimal policy π^*, which is generally non-unique and leads to optimal state-value and action-state functions:

$$\begin{aligned} v_*(s) &= \max_\pi v_\pi(s), \forall s \in \mathbf{S} \\ q_*(s, a) &= \max_\pi q_\pi(s, a), \forall s \in \mathbf{S} \\ v_*(s) &= \max_{a \in \mathbf{A}(s)} q_{\pi^*}(s, a) \end{aligned}$$

4.2 Deep Reinforcement Learning

In the first RL attempts, the values of the state-value and action-state functions were saved in tabular form; the table was then updated after many exploration episodes that lead to an optimal version of it to be used by the agent to navigate the environment in the most advantageous way.

The limitations of this approach become evident when the environment becomes very large, making the table extremely resource consuming both in matter of data stored and time, given that sweeping such a large array of values becomes computationally expensive. Moreover, when the environment is continuous, the tabular method turns out to be altogether impossible to apply unless a discretization is made, which would still result in a loss of information.

The solve this problem function approximators are used to estimate both the state-value and the action-state functions. Although a number of approaches can be employed, such as tile-coding or radial-basis functions in the early beginnings of RL, the most obvious one consists in the usage of deep neural networks, hence the name Deep Reinforcement Learning (DRL).

Neural networks are trained to predict how valuable certain actions and states are from samples of their respective spaces; the parameters to be optimized for said task are the weights and biases of the network, generally represented as $\theta^{\pi,v,q} \in \Theta$, with Θ being the parameter space, and the subscript of θ being the function to which the parameters belong: either the state-value function, the action-state function or the agent's policy π.

Policy Gradient

As said before, the goal of RL is to find a policy π_θ that maximizes the expected sum of discounted rewards:

$$\max_\theta \eta(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t r_t\right]$$

where $\mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\cdot]$ is the estimation given a *trajectory* $\tau = (s_0, a_0, s_1, a_1, ..., s_T, a_T)$ sampled from the current policy π_θ.

To do so, the policy gradient (PG) method calculates the direction of steepest ascent to the global maximum; this is possible by calculating the gradient $\mathbf{g}$ as:

$$\mathbf{g} = \nabla_\theta \eta(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t) A^{\pi_\theta}(s_t, a_t)\right] \tag{4.1}$$

where $A^{\pi_\theta}(s_t, a_t)$ is called *advantage function*:

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s)$$

Seeing that the advantage function is the difference between the value function and the Q-value function *red*write that they can also be called like this in the previous chapter, it's intuitive to understand that it provides a measure of how good an action is (given by the Q-value), given the estimation of the best outcome for a certain state (given by the value function); in fact, the advantage is always non-positive, with it being 0 for an optimal policy.

Having calculated the gradient $\mathbf{g}$, the parameters update will be:

$$\theta_{k+1} = \theta_k + \alpha\mathbf{g}$$

with α being the learning rate.

The PG methods have a few intrinsic problems:

- When the objective function to be optimized $\eta(\pi_\theta)$ is highly dimensional and highly non-linear, too large of a step in the steepest ascent direction could cause the optimization procedure to drop in a valley, wasting the progress done until that last step. It is a very common problem in RL known as *catastrophic forgetting* .
- For such complicated optimization function it is impossible to choose a value for the learning rate α that would suit the whole training process: in fact, a value too large would result in catastrophic forgetting whereas a step too small would slow down the learning phase to an unacceptable time.
- It's not possible to update the policy every timestep or every few timesteps: doing so would fail to generalize the gradient direction for the whole episode, it would instead point it in the optimal direction for that subset of states and actions explored in those few timesteps. Considering that an episode could be made of hundreds if not thousands of timesteps, the PG method has an extremely low sample efficiency, one of the most discussed problems of RL and possibly the biggest challenge today.

4.3 Trust Region Policy Optimization

Trust Region Policy Optimization [23] is an advanced DRL algorithm first presented in 2015; it is to date one of the best performing algorithms invented alongside its derivative PPO which makes it scalable to larger policy networks.

Being fairly complicated from the mathematical point of view I'll first break down its features to make it more tractable.

Minorize-Maximization Algorithm

The minorize-maximization (MM) is a well known process to construct an optimization algorithm in an iterative way. The core concept is that it maximizes a function that is a lower-bound of the original optimization objective; for it to be effective, it is necessary that the step taken in the ascending direction of the lower bound function is small enough to respect the locality of the approximation. If this constraint is satisfied, the MM method guarantees policy improvement every time and will eventually lead to the optimal policy.

A further benefit of using MM is that it can reduce computational expensiveness of the optimization by a great deal: in fact the usual choice for the lower-bound function is quadratic, which is convex and easily optimizable. Its form is:

$$M(\theta) = \mathbf{g} \cdot (\theta - \theta_{\text{old}}) - \frac{\beta}{2} (\theta - \theta_{\text{old}})^T F (\theta - \theta_{\text{old}})$$

with θ_{old} being the current policy's parameters and *theta* being the new one's.

Trust Region

Trust region is an alternative optimization method to line search, which gradient ascent is part of. In this method the optimization takes place within a subset of the region of the objective function, which in this case is defined by the the n-sphere of radius δ. For example, let $m_k(s)$ be the approximation to the original objective function $f(s)$, the optimization will be:

$$\begin{aligned} &\max_{s \in \mathbb{R}^n} \mathrm{m}_k(s) \\ \text{s.t.} \quad &\|s\| \leq \delta \end{aligned}$$

It is to be noted that this new optimization strategy by itself does not solve the problems of PG listed earlier in the chapter: to control the learning speed it would still be necessary to shrink and expand the radius δ to allow more or less liberty in the update of the policy; this means that to prevent catastrophic forgetting it's still necessary to adapt δ based on the curvature of the function to be optimized.

Considering how expensive it can be to do this, a workaround is used in TRPO to modify the trust region span based on how different the new policy is with respect to the new one: to prevent forgetting the limit on the divergence between the two policies is reduced.

Importance Sampling

If the introduction of a trust region is a step towards solving the problem of forgetting, importance sampling has the role of solving the sample inefficiency of policy gradient methods; this is a technique for estimating the properties of a distribution, while only having samples generated from another distribution that has to be "similar" to the first one.

It is possible to use this technique to obtain the expected value of a function $f(x)$ where x belongs to a desired distribution of probability density function $p(x)$ by sampling x from a p.d.f. of a "biased" or "sampling" distribution $q(x)$:

$$\mathbb{E}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$$

where the term $\frac{p(x)}{q(x)}$ is called *importance sampling weight* for x. By calculating the variance of the above estimation we get:

$$\begin{aligned}\operatorname{var}\left(\hat{\mu}_q\right) &= \frac{1}{N}\operatorname{var}\left(\frac{p(x)}{q(x)}f(x)\right)\\ &= \frac{1}{N}\left(\mathop{\mathbb{E}}_{x\sim q}\left[\left(\frac{p(x)}{q(x)}f(x)\right)^2\right]-\mathop{\mathbb{E}}_{x\sim q}\left[\frac{p(x)}{q(x)}f(x)\right]^2\right)\\ &= \frac{1}{N}\left(\mathop{\mathbb{E}}_{x\sim p}\left[\frac{p(x)}{q(x)}f(x)^2\right]-\mathop{\mathbb{E}}_{x\sim p}[f(x)]^2\right)\end{aligned}$$

where, in the last expression, $\frac{p(x)}{q(x)}$ is the decisive factor for the tightness of the variance of the estimation.

By applying the importance sampling to Eq.4.1 we obtain:

$$\mathbf{g}=\mathop{\mathbb{E}}_{\tau\sim\pi_{\theta'}}\left[\sum_{t=0}^{T}\frac{P\left(\tau_t\mid\theta\right)}{P\left(\tau_t\mid\theta'\right)}\gamma^t\nabla_\theta\log\pi_\theta\left(a_t\mid s_t\right)A^\theta\left(s_t,a_t\right)\right]$$

The issue now is the last expression $\frac{P(\tau_t|\theta)}{P(\tau_t|\theta')}$ which can lead to an extremely misleading estimation even when there's a small difference between the two probabilities; in fact in can be proven that:

$$\frac{P\left(\tau_t\mid\theta\right)}{P\left(\tau_t\mid\theta'\right)}=\prod_{t'=0}^{t}\frac{\pi_\theta\left(a_{t'}\mid s_{t'}\right)}{\pi_{\theta'}\left(a_{t'}\mid s_{t'}\right)}$$

which indicates that the longer the trajectory τ is, the more obvious the error will become. This phenomenon is known as *exploding* or *vanishing importance sampling weights*, and just as the catastrophic forgetting problem, can be solved by limiting the divergence between the new and the old policies, and thus between their distributions: respectively the desired one and the sampling one.

So all the techniques described so far have a caveat: the new policy has to be similar to the old one for the update to be effective an monotonicly improving. To add this last property to the update, the authors of TRPO first modified the optimization function and then added a term that limits the difference between the two policies called *Kullback-Leibler divergence.*

Kullback-Leibler Divergence

For the last step, it is necessary to first modify the optimization function. To do so, the authors introduced the *relative policy performance identity*, defined as follows:

$$\eta\left(\pi'\right) - \eta(\pi) = \mathop{\mathbb{E}}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi}\left(s_t, a_t\right)\right] \quad = \frac{1}{1-\gamma}\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi'}\\ a\sim\pi}}\left[\frac{\pi'(a \mid s)}{\pi(a \mid s)}A^{\pi}(s,a)\right] \tag{4.2}$$

where d^{π} is defined as the *discounted future state distribution* sampled from the policy π (the notation $\pi_{\theta'}$ was simplified to π').

The intuitive reason for this change of optimization function is to better exploit the relationship between the performances of the two policies, making it easier to bound the difference between them and making more evident the improvement from one to the other. It can easily be proven that maximizing π' for $\eta(\pi')$ or $\eta(\pi') - \eta(\pi)$ is the same, given that $\eta(\pi)$ can be treated as a constant.

The last step is to change the state sampling distribution ($s \sim d^{\pi'}$) in Eq.4.2. The way it is done, once again, is by assuming that $d^{\pi'} \approx d^{\pi}$, making the objective function:

$$\eta\left(\pi'\right) - \eta(\pi) \approx \frac{1}{1-\gamma}\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi}\\ a\sim\pi}}\left[\frac{\pi'(a \mid s)}{\pi(a \mid s)}A^{\pi}(s,a)\right] \doteq \mathcal{L}_{\pi}\left(\pi'\right)$$

Naturally $d^{\pi'} \approx d^{\pi}$ is valid only when the policies are similar; this recurrent problem is finally solved by bounding the relative policy performance with a term dependent on the Kullback-Leibler (KL) divergence, as follows:

$$\left|\eta\left(\pi'\right) - \left(\eta(\pi) + \mathcal{L}_{\pi}\left(\pi'\right)\right)\right| \leq C\sqrt{\mathop{E}_{s\sim d^{\pi}}\left[D_{KL}\left(\pi' \| \pi\right)[s]\right]} \tag{4.3}$$

Putting the KL divergence between the two policies as an upper bound limits the maximum acceptable difference between them; in fact, the KL divergence, also called relative entropy, is a measure of how one probability distribution is different from a second reference probability distribution. The mathematical definition states that: given to discrete probability distributions **P** and **Q**, the relative entropy from **Q** to **P** is defined as:

$$D_{KL}(P\|Q) = \sum_{x} P(x)\log\frac{P(x)}{Q(x)}$$

with:

- $D_{KL}(P\|P) = 0$
- $D_{KL}(P\|Q) \geq 0$
- $D_{KL}(P\|Q) \neq D_{KL}(Q\|P)$

In our case:

$$D_{KL}\left(\pi' \| \pi\right)[s] = \sum_{a\in\mathcal{A}} \pi'(a \mid s)\log\frac{\pi'(a \mid s)}{\pi(a \mid s)}$$

Now that the inequality in Eq.4.3 is explained, the last step is to change the objective function from the relative policy performance identity on the left:

$$\underset{\pi'}{\text{maximize}}\ \eta\left(\pi'\right) - \eta(\pi)$$

to its lower bound on the right, by virtue of the minorize-maximixation method:

$$\arg\max_{\pi'}\ \mathcal{L}_{\pi}\left(\pi'\right) - C\sqrt{\underset{s\sim d^{\pi}}{\mathrm{E}}\left[D_{KL}\left(\pi'\|\pi\right)[s]\right]}$$

which is expressed in the paper as:

$$\begin{aligned} &\arg\max_{\pi'}\ \mathcal{L}_{\pi}\left(\pi'\right) \\ &\text{s.t.}\ \underset{s\sim d^{\pi}}{E}\left[D_{KL}\left(\pi'\|\pi\right)[s]\right] \leq \delta \end{aligned} \tag{4.4}$$

It can be proven that the term on the right of Eq.4.3 is 0 when the two $\pi' = \pi$, which leads to:

$$\eta\left(\pi'\right) - \eta(\pi) \geq 0$$

and hence is proof of monotonic improvement of the policy.

Gradient computation

The optimization problem in Eq.4.4 is well defined, the last element to add is the methodology for computing the solution to it, or to be more precise, for computing the update of the policy's parameters $\theta \in \Theta$.

Firstly, the terms $\mathcal{L}_{\pi}\left(\pi'\right)$ and $D_{KL}\left(\pi'\|\pi\right)[s]$ are expanded using a second-order Taylor's series, where the authors drop the second-order term of $\mathcal{L}$ for being insignificant to the result.

$$\begin{aligned} &\mathcal{L}_{\theta_k}(\theta) \approx \mathcal{L}_{\theta_k}\left(\theta_k\right) + g^T\left(\theta - \theta_k\right) \\ &\bar{D}_{KL}\left(\theta\|\theta_k\right) \approx \bar{D}_{KL}\left(\theta_k\|\theta_k\right) + \nabla_{\theta}\bar{D}_{KL}\left(\theta\|\theta_k\right)|_{\theta_k}\left(\theta - \theta_k\right) + \frac{1}{2}\left(\theta - \theta_k\right)^T H\left(\theta - \theta_k\right) \end{aligned}$$

where the constant term of both $\mathcal{L}$ and D_{KL}, and the first-order term of D_{KL} are zero, making the series:

$$\begin{aligned} &\mathcal{L}_{\theta_k}(\theta) \approx g^T\left(\theta - \theta_k\right) && g \doteq \nabla_{\theta}\mathcal{L}_{\theta_k}(\theta)|_{\theta_k} \\ &\bar{D}_{KL}\left(\theta\|\theta_k\right) \approx \frac{1}{2}\left(\theta - \theta_k\right)^T H\left(\theta - \theta_k\right) && H \doteq \nabla^2_{\theta}\bar{D}_{KL}\left(\theta\|\theta_k\right)\Big|_{\theta_k} \end{aligned}$$

By substituting in Eq.4.4 what just obtained, we get the expression of the optimization problem in terms of policy network's parameters:

$$\begin{aligned} \theta_{k+1} = &\arg\max_{\theta} g^T\left(\theta - \theta_k\right) \\ &\text{s.t.}\ \frac{1}{2}\left(\theta - \theta_k\right)^T H\left(\theta - \theta_k\right) \leq \delta \end{aligned}$$

which is a quadratic equation whose analytic solution is:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g \tag{4.5}$$

The fundamental property of this last equation is that the term $H^{-1}g$ provides a gradient direction that is independent from the parametrization used to compute it. This means that substituting the model used to calculate H will not modify the parameter change $\Delta\theta$.

The hessian of the KL divergence H is also called *Fisher information matrix*, it measures the curvature of the log probability of the policy as follows:

$$H = \underset{s,a\sim\theta^k}{E}\left[\nabla_\theta \log \pi_\theta(a \mid s)|_{\theta_k} \nabla_\theta \log \pi_\theta(a \mid s)\Big|_{\theta_k}^{T} \right]$$

Truncated Natural Policy Gradient

The problematic aspect of Eq.4.5 is that the Hessian of the KL divergence H can be computationally burdening for larger policy networks: its space complexity is $\mathcal{O}(N^2)$ whereas the time complexity for its inversion is $\mathcal{O}(N^3)$. A workaround for the inversion of H is to use the *conjugate gradient* (CG) method: the given $x_k = H_k^{-1} g_k$ for any of the k step of the training is transformed in an optimization problem for a quadratic equation, by virtue of H being symmetric and positive-definite:

$$H_k x_k = g_k \quad \rightarrow \quad \min_{x_k \in \mathbb{R}^N} \frac{1}{2} x_k^T H_k x_k - g_k^T x_k$$

This trick lowers the time complexity from $\mathcal{O}(N^3)$ to $\mathcal{O}(N)$ making it partially scalable to bigger networks with a higher number of parameters, but comes with some minor complications. The quadratic simplification might lead to violations of the KL divergence and the trust region constraints. By expanding the trust region slightly and implementing an if-statement to check whether or not the update θ_{k+1} improves the policy ($\mathcal{L}_{\theta_k}(\theta_{k+1})$) and satisfies the KL constraint ($\bar{D}_{KL}(\theta_{k+1} \| \theta_k) \leq \delta$); if the new candidate parameters don't respect said constraints, the natural policy gradient is decayed by a factor α^j with j increasing starting from 0 every time the if-statement returns false.

Chapter 5

Open-loop dynamic controller using Recurrent Neural Networks

5.1 Summary of the Method

This chapter is about the first method I developed during the thesis period. It's aimed at creating an open-loop dynamic controller for soft robots through the exploitation of artificial neural networks (ANN).

The first step towards obtaining controller is to collect a static dataset through pseudo-random motor babbling, which will then serve as training data for an ANN that approximates the inverse kinematic (IK) model of the robot. Once the IK model is obtained, a few hundred trajectories are generated by randomly sampling points in the task-space of the manipulator and then interpolating them with a spline; then, these trajectories in the task-space are transformed into the actuation-space using the IK network just trained.

After that, said actuations are given as input to the robot at a frequency that is non-static, in this case 10Hz, and the corresponding task-space positions are recorded. The trajectories outlined by the end-effector end up being significantly different from the desired ones by virtue of the dynamic effects that are present at higher frequencies and are not encompassed by the IK network.

For this reason, the dataset collected at 10Hz is used in a second training of a network with similar structure to the first one.

The result is an open-loop dynamic controller that achieves a very good precision both when tested on random trajectories which were not included in the training, and benchmark trajectories that will be used for comparison with the closed-loop dynamic controller developed later on.

5.2 Static Dataset Collection

The collection of a training dataset is the initial step towards obtaining the final controller.

To achieve a good exploration of the task space, I used pseudo-random actuations after each of which a pause of two seconds is required to allow the soft manipulator to stop oscillating; once the oscillations come to an end, the position of the end-effector is recorded with the corresponding actuation in the dataset.

The dataset is comprised of 4000 entries, which makes the process to obtain it slightly longer than two hours; the considerations in the next section clarify that it is possible to reduce the number of entries and thus the time taken to gather them, with minor losses in the accuracy of the inverse kinematics network trained later.

Algorithm 1 outlines the process.

Algorithm 1 Static Dataset Generation

max_range=100;
min_range=0;
$\delta_{\%}$=0.1;
δ=$\delta_{\%}$(max_range-min_range)
τ_0^1=0; τ_0^2=0; τ_0^3=0;
max_limit_reached$_i$=0 with i=1,2,3
min_limit_reached$_i$=0 with i=1,2,3
dataset=[ ];
while *t<4000* **do**
  while *t%200!=0* **do**
    for *i=1:3* **do**
      if $\tau_{t-1}^i < min\_range_i + \delta$ **then**
        min_limit_reached$_i$=2
      else if $\tau_{t-1}^i > max\_range_i - \delta$ **then**
        max_limit_reached$_i$=2
      if $min\_range_i > 0$ **then**
        τ_t^i= τ_{t-1}^i+random_uniform(0,δ)
        min_limit_reached$_i$-=1
      else if $max\_range_i > 0$ **then**
        τ_t^i= τ_{t-1}^i+random_uniform($-\delta$,0)
        max_limit_reached$_i$-=1
      else
        τ_t^i= τ_{t-1}^i+random_uniform($-\delta$,δ)
      end
      pause(2s)
      manipulator.input([τ_t^1,τ_t^2,τ_t^3]);
      VICON.read([x_t,y_t,z_t]);
      dataset.append([τ_t^1,τ_t^2,τ_t^3],[x_t,y_t,z_t]);
    end
  end
  τ_t^i=random_uniform(min_range,max_range) with i=1,2,3
  t+=1
end

Essentially there is a random repositioning of the end-effector once every 200 actuations

to ensure proper exploration; in fact, a range of actuation variation δ that is too high would result in stronger oscillations that would produce a noisy dataset, but on the other hand a δ that is too low would not guarantee that the end-effector visits the whole task-space. Secondly, the if-statements in the pseudocode make sure that whenever an actuation τ_i is about to saturate, the next two variations Δ_i leads τ_i away from the either saturating to the maximum or the minimum.

5.3 Inverse Kinematic Model with ANN

Once the dataset is collected, it becomes training data for the inverse kinematics (IK) network, which is a straightforward artificial neural network with the three end-effector coordinates $[x_t,y_t,z_t]$ as inputs and the three actuations $[\tau_t^1,\tau_t^2,\tau_t^3]$ as outputs. The ANN is composed by a hidden layer of 64 neurons with a hyperbolic tangent (tanh) activation function and an output layer with linear activation.

Trying different network architectures by modifying hyperparameters (such as the activation functions, the number of neurons, and the gradient descent type) has proven to make a very small difference; this is to be expected considering the relatively simplicity of the IK of this manipulator, for which the 835 parameters of the chosen network are more than enough, and the length of the dataset gathered.

Two strategies that have proven effective are instead the optimization of the batch size fed to the network during the training, and the expansion and normalization of the dataset.

The batchsize that proved to be more effective without extending the training time excessively is 64, and to expand the dataset I simply appended to it the mid-point between each end-effector position and their successive ones, doing the same for the corresponding actuations.

Even if the first assumption on the true IK model of the robot is that it is higly non-linear, linearizing it for very small segments of the task-space is acceptable; that wouldn't have been true if the δ in Alg. 1 were higher than 10%. After having expanded the dataset to a length of 8000, I normalized the inputs to the network (i.e. the end-effector positions) using a standard scaler.

The training was done on 80% of the dataset, while the remaining 20% was half dedicated to the validation set, and half to the test set. After 500 epochs of training using Adam gradient descent with a mean square error loss function, the error on each actuation was:

$$|\tau^i - \hat{\tau}_i| \approx 1.5\% \pm 1\% \qquad \text{with } i = 1,2,3$$

where the percentage is over the total actuation span.

The error shows that the accuracy of the approximated IK model is good enough to be used; a better way to test its goodness would have been to feed the actuations ouputted by the IK network and calculate the euclidean distance between the desired and actual position of the end-effector, but given that this was not the final scope of the method I omitted this type of test and accepted the individual actuation-space errors as proof of acceptability.

5.4 Open-loop Controller

The next step is to create a dataset of dynamic trajectories that will be used to train the open-loop controller. To do this, $n = 5$ points are randomly sampled from the task-space of the manipulator: these and the resting position of the end-effector are then interpolated to create the trajectories. The set of points are interpolated using a cubic spline and the resulting trajectory is saved only if it satisfies the task-space boundaries (as the interpolation might fall outside of it). The dataset $\mathbf{x} \in \mathbf{T}_{\mathbf{x}}^{tar}$ is made of 200 trajectories, composed either of 60 or 90 points to ensure that it comprises a sufficiently large diversity of step sizes $\Delta\mathbf{x}$

Then, the IK network previously trained is used to map the trajectories from the task-space to the actuation-space, generating another dataset $\mathbf{T}_\tau$ that will serve as inputs to the soft manipulator, given at a frequency to which the dynamic effects can no longer be ignored (in this case, f=10Hz). The entries in the actuations dataset $[\tau_1, \tau_2, \tau_3] \in \mathbf{T}_\tau$ are used as target values for the training of the controller network, which will have as inputs the positions obtained feeding said actuations to the manipulator ($\mathbf{x} \in \mathbf{T}_{\mathbf{x}}^{ee}$), as shown in Fig. 5.1

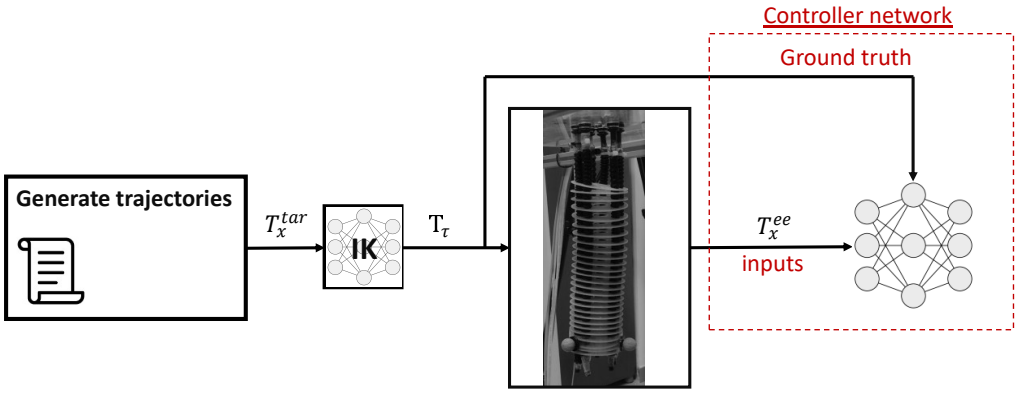


Figure 5.1: Open-loop controller training overview

An example of a trajectory generated by the script can be seen in Figure 5.2: the brown-to-black trajectory is the desired one ($\mathbf{x} \in \mathbf{T}_{\mathbf{x}}^{tar}$) and the yellow-to-red one is the one traced by the end-effector at 10Hz ($\mathbf{x} \in \mathbf{T}_{\mathbf{x}}^{ee}$). The dynamics effects are responsible for the error.

The controller, being open-loop, coincides with an approximation of the manipulator's inverse dynamic (ID) model. Considering that making assumptions over the degree of dependency of the dynamics with the past is a hard task with this type of robot, I approached the problem of chosing the inputs to the model in an empirical way; more precisely, the issue to solve is the length of the horizon over the past end-effector positions. The inputs will have the form:

$$[\mathbf{x}_{t+1}, \mathbf{x}_t, ..., \mathbf{x}_{t-T}]$$

with $\mathbf{x}_{t+1}$ being the next target position, $\mathbf{x}_t, ..., \mathbf{x}_{t-T}$ being the past end-effector positions inside the horizon length T.

I tried two different types of neural networks: an ANN like the one used for the inverse kinematics approximation, and an LSTM. In fact, the choice of architecture is not obvious:

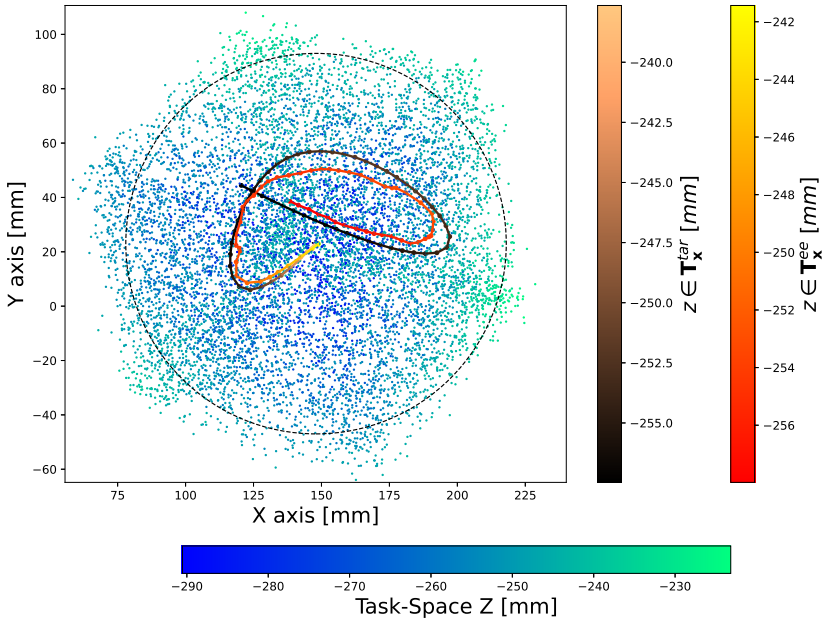


Figure 5.2: Trajectory example

an LSTM should be better for the task at hand considering the predisposition of said networks to predict outputs from a time-series, but considering the limited length of the horizon, an ANN could work just as fine, if not better considering the more limited amount of parameters to optimize.

To compare the two architectures, I use the same number of epochs (N=1000), the same batchsize (b=128) and the same activation function (*tanh*). The dataset is composed by 240 trajectories, half of 60 points and the other half of 90, which when divided in windows of the same length as the horizon length T, produce a dataset of dimensions $[\sim 170\,000, T+2,3]$ for the LSTM, and one of dimension $[\sim 170\,000, (T+2)\cdot 3]$ for the ANN.

The dataset is then split into training data and testing data, the latter including 15 trajectories not seen during training. The index used to evaluate the performance of the networks is the average of the errors $\mu(\Delta\mathbf{e})$ and their standard deviation $\sigma(\Delta\mathbf{e})$ where:

$$\Delta\mathbf{e} = \left\{ \Delta e_t^i \;:\; \Delta e_t^i = |\hat{\tau}_t^i - \tau_t^i| \;\text{ for }\; i = 1{,}2{,}3 \;\text{ and }\; t = 1, ..., |\text{dataset}| \right\}$$

The results in Table 5.1 show that the LSTM outperforms the ANN for any horizon length T, achieving the lowest error of just above 1% of the total actuation range for T=2 (i.e input=$[\mathbf{x}_{t+1}, \mathbf{x}_t, \mathbf{x}_{t-1}, \mathbf{x}_{t-2}]$). It's debatable whether such a relatively low performance

improvement is worth having such a wider parameter space, but considering the limited activation frequencies that make the computational expensiveness less paramount, it's better to opt for the best performing network.

Table 5.1: Inverse dynamics results

Net	**Horizon length**	$\mu(\Delta \mathbf{e})$ [%]	$\sigma(\Delta \mathbf{e})$ [%]
ANN	0	2.07	1.62
LSTM	0	1.92	1.54
ANN	1	1.68	1.35
LSTM	1	1.48	1.21
ANN	2	1.40	1.11
LSTM	2	1.13	0.91

5.5 Results

To test the dynamic controller, I first transferred the LSTM to MATLAB, from which the soft manipulator is controlled. The way it is deployed is shown in Figure 5.3, where the *test trajectory* is either one of the 15 random trajectories in the test dataset, or a benchmark trajectory (either a circle or the infinity symbol).

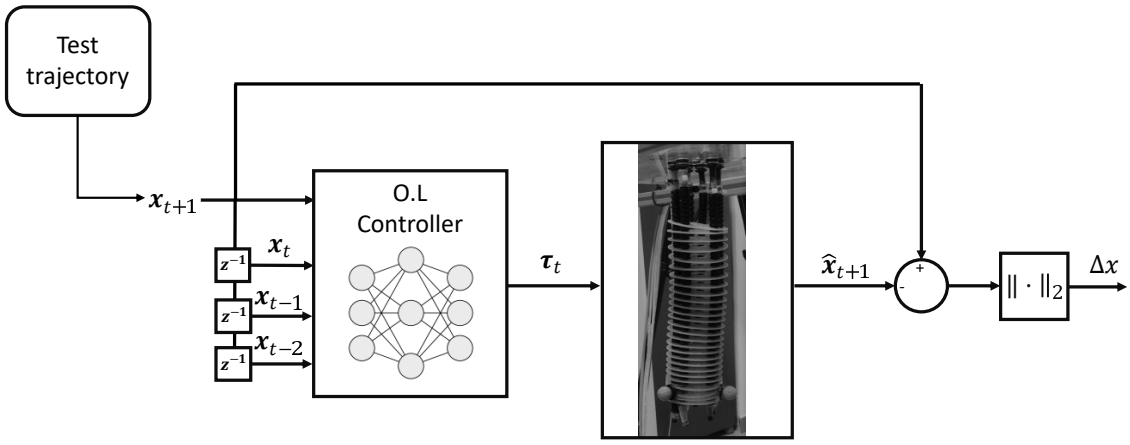


Figure 5.3: Open-loop configuration on the robot

Firstly I tested the controller on the trajectories in the test dataset; this provides a proof of the improvement of the performances when going from the IK controller to the newly trained ID controller. As can be seen from comparing Fig. 5.4 and Fig. 5.2, the former being ID controller's result and the latter being the IK controller's one, the refinement of

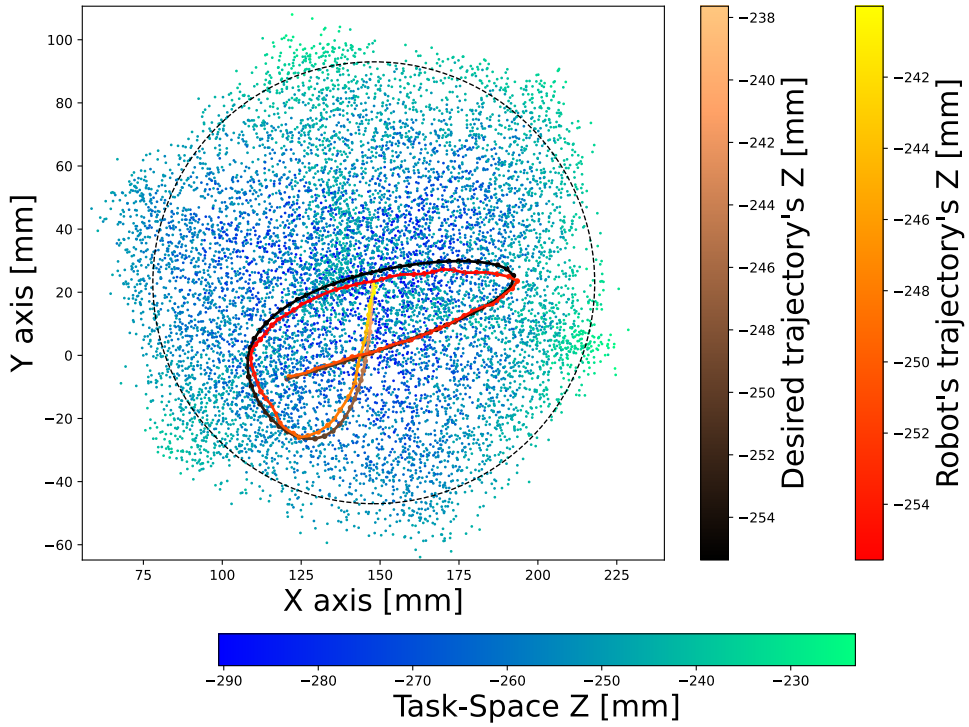


Figure 5.4: Random trajectory example

the execution is immediately evident.
More specifically, the IK produces an average error $\mu(\Delta\mathbf{X}_{IK}) \approx 10.6mm$, with $\Delta\mathbf{X}_{IK}$ being:

$$\Delta\mathbf{X}_{IK} = \left\{\Delta\mathbf{x}_t^i \;:\; \Delta\mathbf{x}_t^i = \|\hat{\mathbf{x}}_{t,IK}^i - \mathbf{x}_t^i\|_2 \quad i = 1,2,3;\; t = 1,...,|\text{trajectory}|;\; \mathbf{x}_t^i \in \mathbf{T}_x^{tar(TEST)}\right\}$$

whereas the ID controller gets an average error, calculated as above, $\mu(\Delta\mathbf{X}_{ID}) \approx 3.2mm$.

To be more precise, the average absolute errors in three dimensions x, y, z, for both cases are:

Table 5.2: Open-loop's errors on x,y,z, random trajectory

	IK	ID
$\mu(\Delta x)$	6.8 mm	2.6 mm
$\mu(\Delta y)$	6.5 mm	1.8 mm
$\mu(\Delta z)$	1.1 mm	1.1 mm

The results for the other random test trajectories are more or less consistent with the example shown above.

The benchmark trajectories are a circle and an infinity symbol, both at constant velocity and both repeated twice. They are 2D, meaning that the end-effector has to drop a few centimeters from its starting position, and then draw them on a plane. The result for the infinity symbol can be seen in Figure 5.5, and the errors for both of them are reported in Table 5.3.

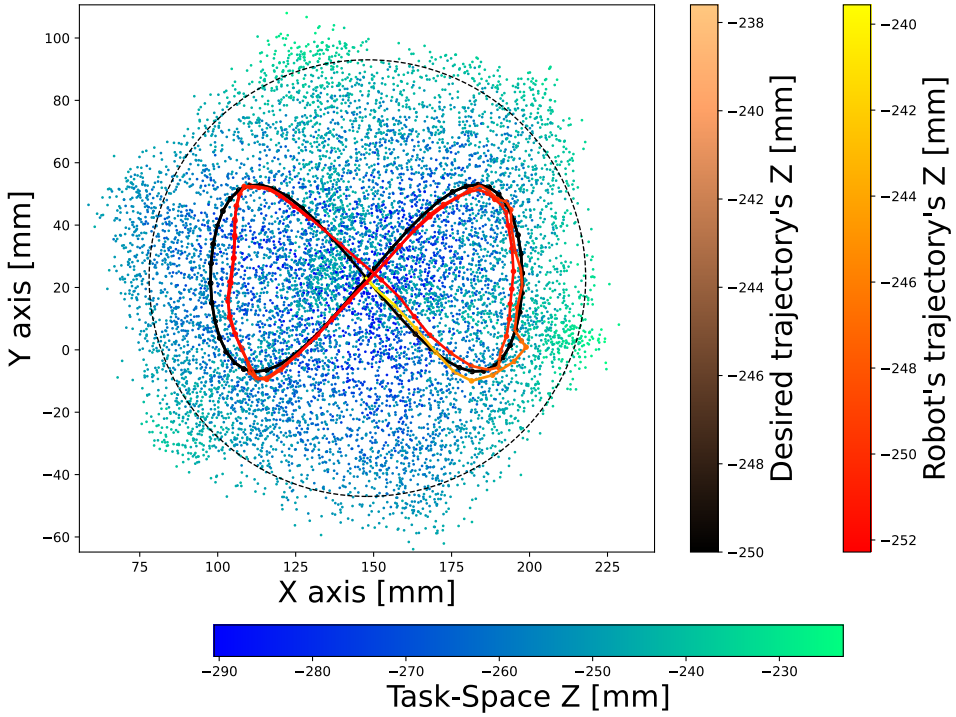


Figure 5.5: Infinity symbol in open-loop

Table 5.3: Open-loop's errors on x,y,z, benchmark trajectories

	Infinity	Circle
Δx	2.6 mm	2.9 mm
Δy	1.8 mm	2.1 mm
Δz	1.1 mm	0.9 mm
$\Delta \mathbf{X}$	3.8 mm	4.0 mm

Chapter 6

Closed-loop Dynamic Controller using TRPO

This second method was tested both on the I-Support (see section 2.1) and on the more complex AM-I-Support (see section 2.2); I tried to make the distinction between the two as clear as possible.

The chapter is presented in the chronological order of the steps that belong to the method. For clarity sake, I am going to explain them in logical order: the objective of the method is to train a controller using a RL algorithm called Trust Region Policy Optimization (see section 4.3). To do that, it is necessary to have an environment to train in; ideally this would be the real robotic platform but, due to reinforcement learning's biggest downside, the sample inefficiency, it would take days if not weeks of continuous training. This is of course unacceptable, so the controller must be trained in silico on an approximation of the robot's dynamic model. To obtain said model, I employed a Long-short term memory (LSTM) network (see 3.3) which necessitates a dataset to be trained on. The dataset collection is hence the first thing to do in chronological order to arrive at the final result: the closed-loop dynamic controller.

6.1 Gathering the Dataset

The first step to be taken is to gather a dataset on which to train the dynamic forward model. To do so, I generated a set of pseudo-random actuations; the nomenclature "pseudo" is due to the fact that adopting a completely random exploration policy (i.e. allowing the inputs to go from the bottom to the top of the actuation range) wouldn't create a dataset usable to get the Forward Model, due to the excessive stochasticity. On the other hand, allowing an insufficient rate of change, hence producing smoother input curves in time, would result in a better prediction of the next end-effector position, but wouldn't permit exploration of the environment by the Reinforcement Learning agent during training; this last factor is of paramount importance: without a forward model able to predict the end-effector position after sudden actuation changes, the Value Function estimated in the training process would be largely random in case of sudden changes, making it impossible to descend its gradient to produce an effective control policy.

The dataset is taken at 10Hz; at each timestep a number $i = 1, ..., k$,with k being the dimension of the actuation-space, of $\Delta\tau_i$, uniformly sampled in a range going from -20% to +20% of the total actuation range ($[-0.2,0.2] \cdot (\tau_{max} - \tau_{min})$), are added to the previous actuations. If the actuations exceed the extremes of the actuation range, they are clipped to said extremes. The actuations are then given as input to the soft robot, and almost simultaneously the position of the end-effector is read by the VICON, meaning that the actuations $\tau_t^{1,...,k}$, while in position $\mathbf{x}_t$ will lead the end-effector to position $\mathbf{x}_{t+1}$. The steps described above are summarized by Algorithm 2, whereas the task-spaces for both robots can be seen in section 6.5.1.

Algorithm 2 Dataset Generation

τ_0^i=0; with $i = 1, ..., k$
$\delta_{\%}$=0.1;
δ=$\delta_{\%} \cdot (\tau_{max} - \tau_{min})$
dataset=[ ];
for *t=1:10000* **do**
  for *i=1:3* **do**
    $\Delta\tau^i$=random_uniform($-\delta$,δ)
    τ_t^i=min($\tau_{max}, max(\tau_{min}, \tau_{t-1}^i + \Delta\tau^i)$)
  end
  manipulator.input([τ_t^1,...,τ_t^k]);
  VICON.read([x_t,y_t,z_t]);
  dataset.append([τ_t^1,...,τ_t^k],[x_t,y_t,z_t]);
  pause(0.1s))
end

6.2 Getting the Forward Model

6.2.1 Artificial Neural Network

Once the dataset is gathered, the Forward Model has to be obtained. To do so, I initially employed a simple Artificial Neural Network (ANN) whose inputs are:

- For the *I-Support*: the last set of actuations $\tau_t^{1,2,3} \in \mathbb{R}^3$, the last end-effector position $\mathbf{x}_t^{ee} \in \mathbb{R}^3$, and the previous end-effector positions until the T^{th} timestep in the past $\mathbf{x}_{t-1}^{ee}...\mathbf{x}_{t-T}^{ee} \in \mathbb{R}^{3T}$, where T is the horizon length:

$$\text{Input}_{ANN} = [\mathbf{x}_t^{ee}, \tau_t^{1,2,3}, \mathbf{x}_{t-1}^{ee}...\mathbf{x}_{t-T}^{ee}] \ \in \mathbb{R}^{6+3T} \tag{6.1}$$

  and whose output is the predicted position of the end-effector at the next timestep:

$$\text{Output}_{ANN} = \mathbf{x}_{t+1}^{ee} \ \in \mathbb{R}^3 \tag{6.2}$$

- For the *AM-I-Support*: the last set of actuations $\tau_t^{1,...,6} \in \mathbb{R}^6$, the last end-effector position $\mathbf{x}_t^{ee} \in \mathbb{R}^3$, the last mid-section position $\mathbf{x}_t^{mid} \in \mathbb{R}^3$, the previous end-effector positions for T timesteps in the past $\mathbf{x}_{t-1}^{ee}...\mathbf{x}_{t-T}^{ee} \in \mathbb{R}^{3T}$, and the previous mid-section positions for T timesteps in the past $\mathbf{x}_{t-1}^{mid}...\mathbf{x}_{t-T}^{mid} \in \mathbb{R}^{3T}$:

$$\text{Input}_{ANN} = [\mathbf{x}_t^{ee,mid}, \tau_t^{1,...,6}, \mathbf{x}_{t-1}^{ee,mid}...\mathbf{x}_{t-T}^{ee,mid}] \ \in \mathbb{R}^{12+6T} \tag{6.3}$$

  and whose output are the predicted positiond of the end-effector and the mid-section at the next timestep:

$$\text{Output}_{ANN} = [\mathbf{x}_{t+1}^{ee}, \mathbf{x}_{t+1}^{mid}] \ \in \mathbb{R}^6 \tag{6.4}$$

The first step towards the training of the network is to normalize the inputs in the dataset. Considering that the outputs of the forward dynamic model will be a subsest of the inputs of the controller, I normalized them too; this step shouldn't have any effect on the training of this network, but I did it for convenience (I would have had to do it before the training of the controler's network). The method of normalization was considered as a hyperparameter of the network to be optimized. I tried two different methods: standardization and min-max normalization (from -1 to 1 for the positions of the end-effector and the mid-sections, and from 0 to 1 for the actuations, see the dedicated subsection *"Feature Scaling"* in Section 3.1).

To get the best prediction accuracy as possible, I firstly selected the best horizion length T by fixing the other hyperparameters of the neural network; said hyperparameters are shown in Table 6.1.

I split the dataset in Training set, Validation set, and Test set with a ratio of 75%-10%-15%. The results obtained from the analysis of the horizon length effects are written in Table 6.2 and shown in Figure 6.1. The errors reported are in the cartesian space:

$$\|\Delta\mathbf{x}\|_2 = \|\mathbf{x}^{ee} - \hat{\mathbf{x}}^{ee}\|_2 \tag{6.5}$$

Table 6.1: ANN's hyperparamters

Number of neurons	64	**Number of hidden layers**	1
Inputs' scaler	Standard	**Loss function**	MSE
Outputs' scaler	Standard	**Optimizer**	Adam
Activation function	tanh	**Batch size**	64

with $\mathbf{x}^{ee}$ being the ground truth value in the test dataset and $\hat{\mathbf{x}}^{ee}$ being the prediction of the neural network.

It should be noted that I only made this analysis on the I-Support, given that the experiments took place earlier; I assumed that the results would be valid for the AM-I-Support too.

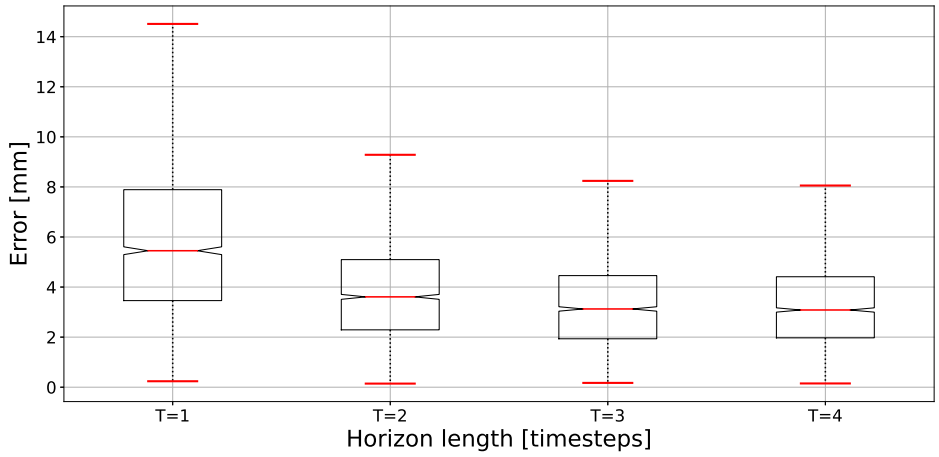


Figure 6.1: Analysis of the effect of the horizon length

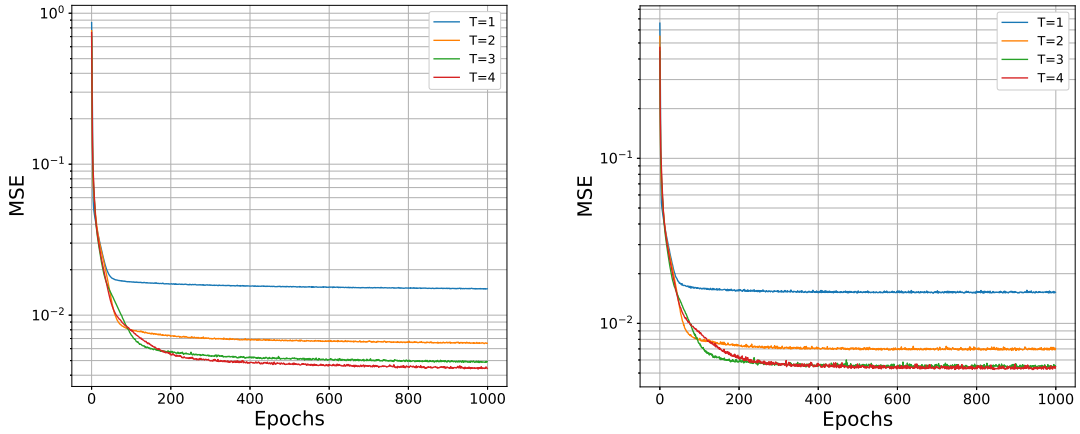
Table 6.2: Different horizon results

Horizon length	Mean error [mm]	Error's standard deviation [mm]
1	6	3.3
2	3.9	2
3	3.3	1.8
4	3.3	1.7

As it can be seen from the aforementioned, the model behaves very poorly when the horizon covers uniquely the first timestep in the past, and gets better by extending it to the first two. The model I found more suitable was the one with an horizon of $T = 3$, for a few reasons:

- The improvement from $T = 3$ to $T = 4$ is almost imperceptible given that it provides an betterment of the error's standard deviation in the order of a 10^{th} of millimiter
- Said improvement is not worth aggravating the model with an extra degree of freedom, looking at it from a performance-complexity ratio point of view.

- From $T = 4$ onward, the network tends to go in overfitting during training, as can be seen from Figure 6.2. This is proof that the extra timesteps in the past from the third are not worthy of consideration.



(a) Loss

(b) Validation loss

Figure 6.2: Training results

I omit the hyperparameter study results for brevity's sake, but I report the different values tried for each one of them:

- *Number of neurons*: 64,128
- *Inputs' scaling*: Min-Max or Standard
- *Outputs' scaling*: Min-Max or Standard
- *Activation function*: ReLu, tanh, or sigmoid
- *Outputs' scaling*: Min-Max or Standard
- *Loss function*: MSE or RMSE
- *Batch size*: 64, 128, or 256

The best configuration is the one shown in Table 6.1. The results obtained for the AM-I-Support forward model are fairly similar: the prediction error is $3.4 \pm 1.8mm$ for the end-effector position, and $2.0 \pm 1.0mm$ for the mid-section position.

6.2.2 Long-Short Term Memory Network

To improve the performances of the controller, a fruitful path is to improve the prediction power of the Forward Model. The neural network architecture that I adopted is a Long short-term memory (LSTM) recurrent neural network (RNN) which is better suited

for time-series predictions with respect to a standard feedforward network. For a light introduction to RNNs and LSTMs I redirect the reader to sections 3.2 and 3.3 respectively.

In addition to the inputs specified in Eq. 6.1, this new forward model recieves as inputs also the past actuation included in the time horizon:

- for the I-Support:

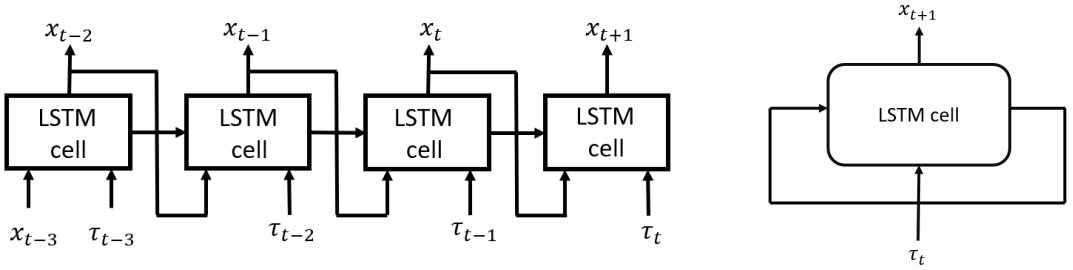
$$Input_{LSTM} = [\mathbf{x}_t^{ee} ... \mathbf{x}_{t-T}^{ee}, \tau_t^{1,2,3}, ..., \tau_{t-T}^{1,2,3}] \in \mathbb{R}^{6(T+1)} \tag{6.6}$$

- for the AM-I-Support:

$$Input_{LSTM} = [\mathbf{x}_t^{ee,mid} ... \mathbf{x}_{t-T}^{ee,mid}, \tau_t^{1,...,6}, ..., \tau_{t-T}^{1,...,6}] \in \mathbb{R}^{12(T+1)} \tag{6.7}$$

whereas the output remains the same as in Eq. 6.2.

A graphical explanation on how a recurrent neural network differs from a normal feed-forward network is shown in Figure 6.3



(a) Unrolled representation

(b) Classic representation

Figure 6.3: LSTM network scheme

As it can be seen from the classic representation of an LSTM (Fig. 6.3b), the input to each cell of the network is the current actuation and the output is the next position of the end-effector; alongside inputs and outputs there are 2 informations that are propagated forward in time:

- The hidden state, which in a shallow LSTM (only one layer) is essentially the output of the previous cell
- The cell state, which withholds data from the past that is useful to sharpen the prediction accuracy in the "present cell". The array constituted by the cell state, unlike the one of the hidden state, has only an intrinsic meaning to the network but no physical one.

For sake of clarity, I included a representation "unrolled" in time (Fig. 6.3a), which further explains what said above: for simplicity the hidden state is represented directly as a feed-back from the output of the past cell. From Figure 6.3a it's possible to understand that the input of the network during training is not:

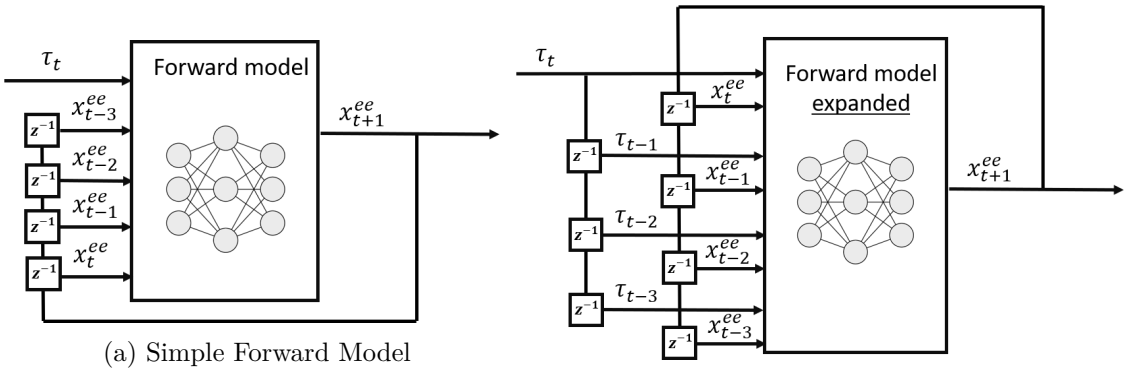
$$\text{Input}_{ANN}\text{=[batch size,input size]} \in \mathbb{R}^{64\text{x}24}$$

anymore, but it becomes a 3-dimensional array:

$$\text{Input}_{LSTM}=[\text{batch size,timesteps,input size}] \in \mathbb{R}^{64\text{x}4\text{x}6}$$

for the I-Support, in case $T = 3$.

A graphical comparison between this new forward model and the one presented in Section 6.2 is shown in Figure 6.4.



(a) Simple Forward Model

(b) Expanded Forward Model

Figure 6.4: Forward Model Schematics

I used the same procedure described in Section 6.2 to get the best horizon length for the inputs; once again I only did this for the I-Support and assumed the same applied for the AM-I-Support.

The hyperparameters are very similar to the ones in Table 6.1, with the notable addiction of the recurrent activation function being *sigmoid*. The considerations to be made by looking at the results written in Table 6.3 and shown in Figure 6.5 are comparable to the ones I already made; the errors presented are again in the cartesian space (Eq.6.5). In this case, once the horizon length goes over $T = 3$, there is actually a performance drop.

Table 6.3: Different horizon results

Horizon length	Mean error [mm]	Error's standard deviation [mm]
1	4.3	2.4
2	1.6	0.9
3	1.4	0.8
4	1.5	0.8

Once again the results obtained for the AM-I-Support forward model similar to the ones attained on its simpler counterpart: the prediction error is $2.0 \pm 0.9mm$ for the end-effector position, and $0.9 \pm 0.4mm$ for the mid-section position.

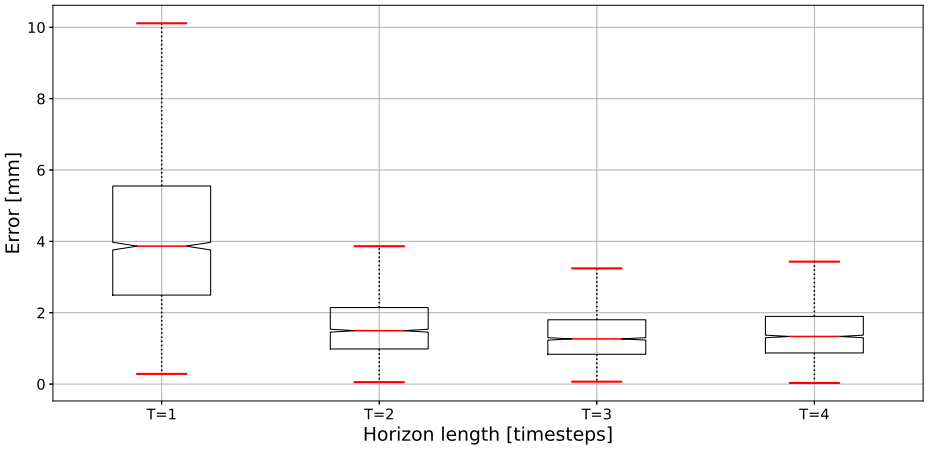


Figure 6.5: Analysis of the effect of the horizon length

6.3 Obtaining the Controller

To obtain the controller the steps I followed are:

1. Decide the shape of the controller: specifically what are going to be the feedforward and feedback inputs to it.
2. Build a suitable environment that simulates the robot: this is based on the Forward Model obtained. It should give *observations* and *rewards* to the controller which in Reinforcement Learning terms is the *agent* to be trained; this *agent* will output actuations (or *actions* from the training point of view) to be fed to the enviroment.
3. Train the *agent*/controller using Trust Region Policy Optimization (TRPO).
4. Test the trained *agent*/controller on two benchmarks (a circle and the infinity symbol) first on the obtained Forward Model and then on the real soft robot.

To fully understand this section, I strongly recommend to first read chapter 4, which is about deep reinforcement learning, and more specifically section 4.3, which encompasses an accurate description of the specific algorithm used during training.

In this section I will always refer to the experiments done on the I-Support. However, the same structure of the controller can apply to the AM-I-Support even if I implemented on the latter only the best controller structure chosen in this next section. A few considerations have to be kept in mind to apply the controllers to the AM-I-Support:

- The actuation space has a dimension of 6 instead of 3 as in the I-Support
- The inputs of the controller are not going to be $\mathbf{x}_t^{ee}$ as in the I-Support, but they're going to include also the mid-section position: $\mathbf{x}_t^{ee,mid}$
- The outputs of the controller are going to change from $\mathbf{x}_{t+1}^{ee}$ to $\mathbf{x}_{t+1}^{ee,mid}$ as well

- The target position $\mathbf{x}_t^{tar}$ refers instead always to the end-effector's desired position, hence it is 3-dimensional for both robots.
- The same can be said for the reward function, it is always a function of the target position and the current end-effector position: in the AM-I-Support it won't include the mid-section's position.

6.3.1 Choosing the Controller

As said above, the first step to be taken is to decide what the agent is allowed to know about the environment, that's to say what are the inputs to our controller to be trained. I came up with a few controller structures, whose inputs and representations are summarized in Table 6.4.

My first approach was to limit the controller's inputs uniquely to the target and the current end-effector position, following the work already published on similar topics [13]. Even though the agent is able to learn a control policy (even faster than the other models, considering that its network has less parameters) I later realized that with this particular model the agent does not have informations about how well of a job it's doing so far, meaning that it's oblivious of the tracking error. Furthermore it wouldn't be able to adapt to environment changes, such as an external disturbance or a alteration of the robot's dynamics, because the network actually learns the correspondence between a couple of points in the task-space ($\mathbf{x}_{t+1}^{tar}$ and $\mathbf{x}_t^{ee}$,) and the correct actuation to go from one to the other; for example, in case of a change of environment, let's say a 10% pressure drop in one of the chambers, this correspondence would be slightly different, but given that the controller does not receive any supplementary information, it would just select the actuations as if the forward model were the one it trained on, maintaining a constant error.

Following this reasoning, I added to both Model 3 and 4 the current tracking error $\mathbf{e}_t = \mathbf{x}_t^{tar} - \mathbf{x}_t^{ee}$.

Table 6.4: Models' summary

	Inputs	**Figure**
Model 1	$[\mathbf{x}_{t+1}^{tar}, \mathbf{x}_t^{ee}]$	Fig. 6.6a
Model 2	$[\mathbf{x}_{t+1}^{tar}, \mathbf{x}_t^{ee}, \mathbf{x}_{t-1}^{ee}, \mathbf{x}_{t-2}^{ee}, \mathbf{x}_{t-3}^{ee}]$	Fig. 6.6b
Model 3	$[\mathbf{x}_{t+1}^{tar}, \mathbf{x}_t^{ee}, \mathbf{e}_t]$	Fig. 6.6c
Model 4	$[\mathbf{x}_{t+1}^{tar}, \mathbf{x}_t^{ee}, \mathbf{e}_t, \tau_{t-1}^{1,2,3}]$	Fig. 6.6d

6.3.2 Training setup

The next phase consists in setting up the learning environment for the agent/controller.

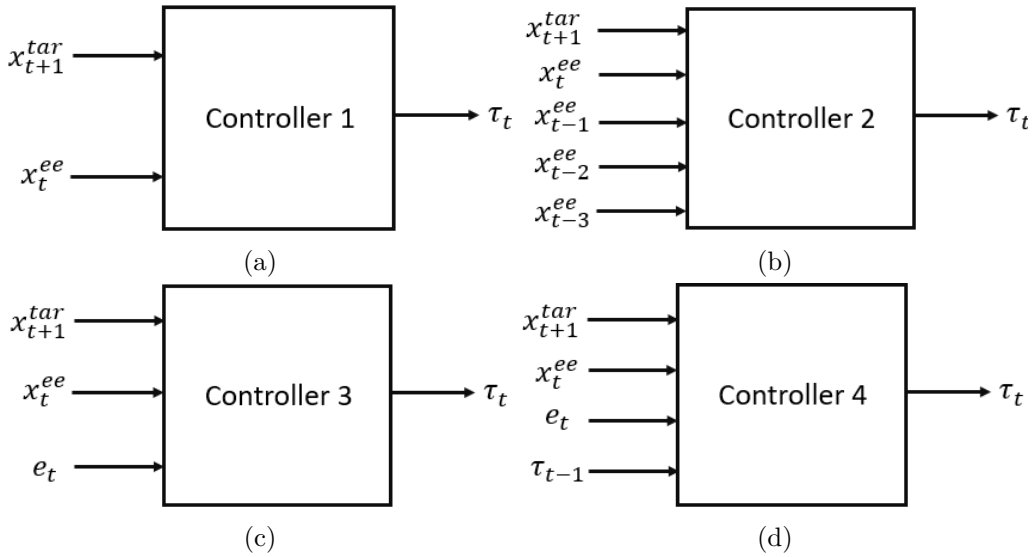


Figure 6.6: Controller shapes

The training procedure is composed by *episodes* which are in turn composed by *steps*. In this particular case, the *steps* are taken with the same frequency with which the dataset was gathered, 10Hz; at each step, the agent is provided with an environmental observation (i.e. the target position, the current end-effector position, etc.) after which the agent has to choose an action/actuation that maximizes a *reward* R_t. An *episode* is instead a randomly generated trajectory of N points which are sampled at each step to supply the target to the agent as explained at the beginning of section 5.4. The *reward* has to be engineered to provide the controller's desired result, naturally, but also to speed-up the training process; I experimented with various reward functions:

- The simplest reward function for a tracking task is the opposite of the distance between target and end-effector at each timestep:

$$R(t) = -\|\mathbf{x}^{tar}(t) - \mathbf{x}^{ee}(t)\|_2 = -\|\Delta\mathbf{x}(t)\|_2 \tag{6.8}$$

- The reward function above can be expanded with an additional reward whenever the end-effector gets within a sphere of a certain radius surrounding the target:

$$R(t) = -\|\Delta\mathbf{x}(t)\|_2 + \begin{cases} 0.25, & \text{if } \ 3mm < \|\Delta\mathbf{x}(t)\|_2 < 10mm. \\ 1, & \text{if } \ \|\Delta\mathbf{x}(t)\|_2 < 3mm \end{cases} \tag{6.9}$$

- A further way to expand Eq. 6.8 is to include a punishment whenever the action chosen leads the end-effector further away from the target:

$$R(t) = -\begin{cases} \|\Delta\mathbf{x}(t)\|_2, & \text{if } \|\Delta\mathbf{x}(t)\|_2 < \|\Delta\mathbf{x}(t-1)\|_2. \\ 1, & \text{if } \|\Delta\mathbf{x}(t)\|_2 \geq \|\Delta\mathbf{x}(t-1)\|_2. \end{cases} \tag{6.10}$$

I ended up using the simplest version of the above (Eq. 6.8) because the advantages of Eq. 6.9 were insignificant from the performance perspective and small from the training speed one. Regarding Eq. 6.10, the training time was improved slightly more significantly, but I decided to avoid it after considering that it notably impedes the exploration of the task space, meaning that the *agent* is denied taking wrong decisions which lead the end-effector away from the target. This consideration is a speculation that might be wrong, but never the less led me to avoid this reward function.

I expected, while building the environment and coming up with its reward function, that its lack of dependence on the variation of actuation could possibly lead to unsmooth actuation curves while following a smooth trajectory; if that were the case, I would have used a reward function like the following:

$$R(t) = -\|\Delta\mathbf{x}(t)\|_2\mathbf{Q}\|\Delta\mathbf{x}(t)\|_2^T - \|\Delta\boldsymbol{\tau}(t)\|_2\mathbf{R}\|\Delta\boldsymbol{\tau}(t)\|_2^T \tag{6.11}$$

where $\mathbf{Q}$ and $\mathbf{R}$ would have been $3x3$ diagonal matrices whose elements would have become hyperparameters of the learning architecture. This was not the case: the actuations coming out of the controller trained using the reward function in Eq. 6.8 were already smooth; this is due to how the TRPO architecture works: the maximization is in fact not of the reward at a single time instant, but of the cumulative sum of the expected rewards in the future, discounted by a factor γ. This means that an outlying set of actuations would have a negative effect on the future rewards that the agent would receive, hence it is avoided by default.

Once the reward function is decided, I put together the environment as explained in Algorithm 3.

The hyperparameters for the TRPO's training and for the controllers' networks are presented in Table 6.6 and Table 6.5 respectively.

Table 6.5: Cotrollers' networks hyperparameters

Number of hidden layers	2	
Neurons per hidden layer	64	
Hidden layers' activation	tanh	
Output activation	linear	
Total number of parameters	**Model 3**	**Model 4**
	4995	5187

Algorithm 3 Environment overview

task_space=dataset.load()
FM_simple=forward_model_network.load()
FM_expanded=forward_model_expanded_network.load()
for *episode=1:7500* **do**
  N=random_integer(60...120) ▷Random number of trajectory points
  task_space_points=task_space.sample(5) ▷Pick 5 points from the task space
  trajectory=task_space_points.interpolate(N) ▷Get a trajectory of N equidispaced points
  $x^{ee}_{t-3}, ..., x^{ee}_{t}$=trajectory(1)...trajectory(4)
  x^{tar}_{t+1}=trajectory(5)
  $\tau_{t-3}, ..., \tau_{t-1}$=[0,0,0]...[0,0,0]
  Δx_t= [0,0,0]
  for *t=6:N* **do**
    Observation=$[x^{ee}_{t}, x^{tar}_{t+1}, \Delta x_t]$
    τ_t=agent.choose_action(Observation)
    if *t<9* **then**
      ▷Use the simple F.M. to fill the buffer of the expanded F.M.
      $\hat{x}^{ee}_{t+1}$=FM_simple.predict($[x^{ee}_{t-3}, ..., x^{ee}_{t}, \tau_t]$)
    end
    else
      $\hat{x}^{ee}_{t+1}$=FM_expanded.predict($[x^{ee}_{t-3}, ..., x^{ee}_{t}, \tau_{t-3}, ..., \tau_t]$)
    end
    $\Delta x_t = \hat{x}^{ee}_{t+1} - x^{tar}_{t+1}$
    $x^{ee}_{t-3}, ..., x^{ee}_{t-1} = x^{ee}_{t-2}, ..., x^{ee}_{t}$
    $x^{ee}_{t} = \hat{x}^{ee}_{t+1}$
    x^{tar}_{t+1}=trajectory(t)
    $\tau_{t-3}, ..., \tau_{t-1} = \tau_{t-2}, ..., \tau_t$
    $R_t = -\|\Delta x_t\|_2$
  end
end

Table 6.6: TRPO's Hyperparameters

Hyperparameter	**Value**
γ	0.99
Max KL	0.005
λ	0.98
Timesteps per batch	1024
Number of episodes	75000

6.4 Disturbance Resistant Controller

The fact that the trained controller is able to work on the real robot while being trained on an imprecise forward model, is the proof that it is resistant to changes of its forward model; this is one of the objectives that I started from: unlike a rigid robot, whose direct dynamics and kinematics are very unlikely to change, a soft robot is much more susceptible to internal damages or, more generally, is prone to deform and change its body under the structural stress provided by prolonged actuations.

A second objective was to make it resistant to more substantial changes in its forward model, such as the ones coming from hanging a weight to its end-effector. This kind of disturbance rejection was not acheived by the controller: hanging a 25g weight to its bottom leads to very poor performances, meaning that the circle is reproduced in a shrunk version, proving that it's unable to make up for the missing pressure required. It has to be said that the dataset was taken with a maximum input of 100 (which is equal to a pressure of about 0.85bar), while its theoretical limit is 1.2bar; considering that for the circle benchmark the actuations reach alternately their saturation point, it would be impossible for the robot to reproduce the same circle if the limit remained 100, hence during the weighted test I removed this saturation point, expanding it to 150. The trained neural network creates a *policy* which should, theoretically, be able to generalize also for outliers of the training dataset; this theoretical feature has proven to be false, probably due to the fact that the neurons, actuated by *tanh*, clip their output to a saturation point.

Initially I imagined that shrinking the benchmark trajectories to a size that is reachable both with and without the weight attached would solve this problem. This wasn't the case: even with a smaller circle the controller fails to increase the actuations to counteract the weight effect on its dynamics, proving definitively that the controller is not resistant to large changes of the manipulator's forward model.

Clearly the model had to include an input that could help the agent identify the magnitude of the disturbance being applied to the robot. For this reason I decided to mimic a very diffused control technique that includes the forward model in the controlling architecture to provide a prediction of the expected end-effector position. As shown in Figure 6.7, the forward model receives the actuation $\boldsymbol{\tau}_t$ alongside all the inputs indicated in Eq. 6.1 and 6.6 for the I-Support and in Eq.6.3 and 6.7 for the AM-I-Support; in the outline shown in Fig. 6.7 [...] encompasses all the inputs shown in Fig. 6.4, dropped for sake of compactness.

The predicted end-effector position at the next time step $\hat{\mathbf{x}}^{ee}_{t+1}$ is used to calculate the magnitude of the disturbance, indicated as $\boldsymbol{\delta}_m$. The reason why I chose to calculate $\boldsymbol{\delta}_m$ as:

$$\delta^i_m = \hat{x}^{i,ee}_{t+1} / x^{i,ee}_{t+1} \quad \text{for} \quad i = 1, ..., k \quad \text{or} \quad \boldsymbol{\delta}_m = \hat{\mathbf{x}}^{ee}_{t+1} \odot (\mathbf{x}^{ee}_{t+1})^{-1}$$

and not by simply using the difference between the two is because, considering that the task space has been normalized so that the resting position of the end-effector is at [0,0,0], a $\boldsymbol{\delta}_m > 1$ would point to and expansion of the task space whereas a $\boldsymbol{\delta}_m < 1$ would instead characterize a contraction of it. To train the controller the environment had to include the disturbance in it; this can be achieved by adding a random multiplying factor $\boldsymbol{\delta}_m$ to the output of the forward model during the training: at the beginning of every episode the factor is sampled randomly from a uniform distribution $\delta_m = \mathcal{U}(0.6, 1)$, and every prediction of the next end-effector position is multiplied by it for the entire duration of the

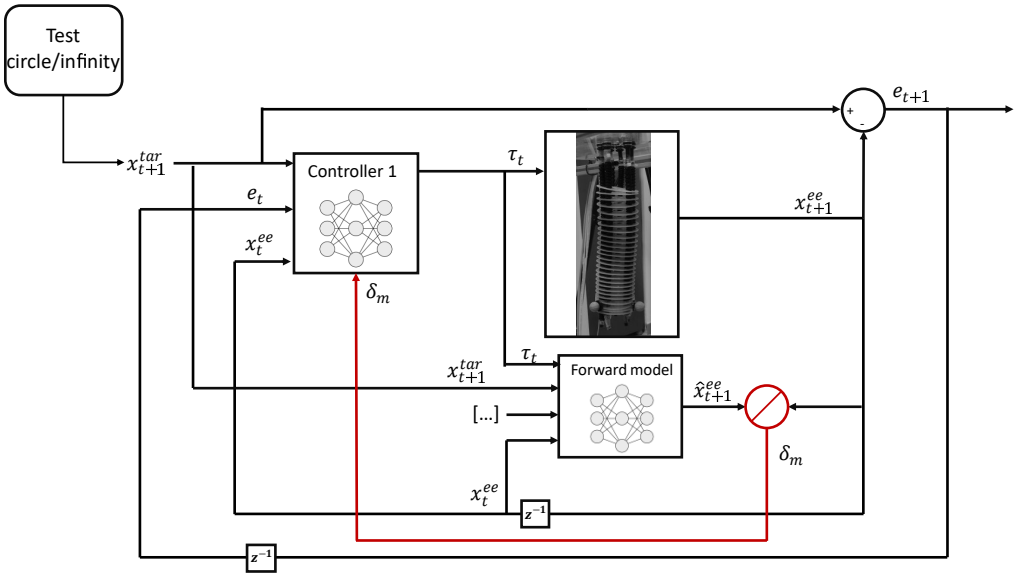


Figure 6.7: Control strategy that includes the forward model's prediction

episode:

$$\hat{\mathbf{x}}_{t+1}^{ee} = \delta_m \cdot \mathrm{FM}([\mathbf{x}_{t-3}^{ee}, ..., \mathbf{x}_t^{ee}, \boldsymbol{\tau}_{t-3}, ..., \boldsymbol{\tau}_t]) \quad \text{for } t = 4...N.$$

As said before, the predictions coming from the forward model belong to a standard distribution, so the multiplication has a deforming effect, rather than the translational one that it would have on the unscaled cartesian coordinates (a similar result could be achieved by setting the origin of the reference system coinciding with the rest position of the end-effector). A graphical overview of this procedure is shown in Figure 6.8.

The goal of this procedure was to augment the span of forward model modifications that the controller can reject, and to make it even less dependant on the model on which it trained.

I tested three variants of this methods:

1. The first one is the one briefly described above: the deformation $\boldsymbol{\delta}_m$ is sampled uniformly in the range [0.6,1], and for the duration of the episode is kept constant. This means that if the episode was comprised of a circular trajectory, that trajectory would be perfectly shrunk by a factor $\boldsymbol{\delta}_m$.

2. The second is similar to the first one: the deformation $\boldsymbol{\delta}_m$ is still sampled at the beginning of the episode, but instead of applying it as is to every step's prediction, there is a 50% chance that the environment adds a random factor to the deformation as: $\boldsymbol{\delta}'_m = \boldsymbol{\delta}_m + p$, with $p \sim \mathcal{U}(-0.1,0.1)$.

   This should mimic a realistic scenario in which the prediction from the forward model is not 100% consistent with the actual output of the plat (the manipulator). Considering that the model is an approximation, the prediction error can vary significantly between one step and the other.

3. The third is a simplified version of the first approach: the deformation $\boldsymbol{\delta}_m$ is not sampled uniformly in the range [0.6,1] but is instead chosen uniquely from either 1 (the undisturbed scenario) and 0.6 (which happens to be the value of $\boldsymbol{\delta}_m$ obtained empirically by attaching the 25g weight to the end-effector).

   The value $\boldsymbol{\delta}_m = [0.6,0.6,0.6]$ was obtained by feeding the robot the actuations to make a perfect circle in open-loop; firstly they're fed without any disturbance, and then with the weight attached. Each point of the disturbed trajectory is then divided by the corresponding point of the undisturbed one. The mean value of the array obtained turned out to be ~ 0.6:

$$\mu(D_m) \approx [0.6,0.6,0.6] \text{ with } D_m = \left\{ \boldsymbol{\delta}_m^t | \boldsymbol{\delta}_m^t = \left[\frac{x_t^d}{x_t^u}, \frac{y_t^d}{y_t^u}, \frac{z_t^d}{z_t^u} \right] \text{ for } t = 1, ..., N \right\} \quad (6.12)$$

   with N being the length of the trajectory, and the superscripts d and u meaning disturbed and undisturbed respectively.

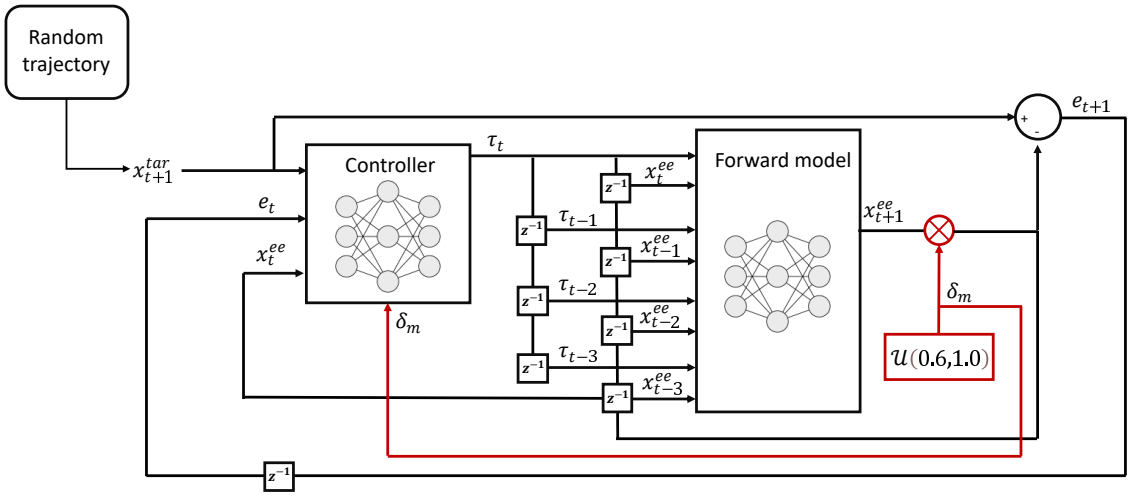


Figure 6.8: Overview of the domain randomization approach

6.5 Results

6.5.1 Task spaces

The actuations for the I-Support robot produced by algorithm 2 can be seen in Figure 6.9; the ones for the AM-I-Support are made in the same way.

Scattering all the 10000 end-effector positions of the I-Support produces Figure 6.10; as can be seen, its task space is fairly limited: it is included in a cube of size $15x15x10cm$.

On the other hand, the end-effector positions of the AM-I-Support, as shown in Figure 6.11, define a much larger task space. In Fig. 6.11a both the positions of the mid-section (in blue) and of the end-effector are plot; while the mid-section is very limited in its movements due to the weight of the distal module below it, the end-effector defines a task space included in a cube of size $50x50x17cm$.

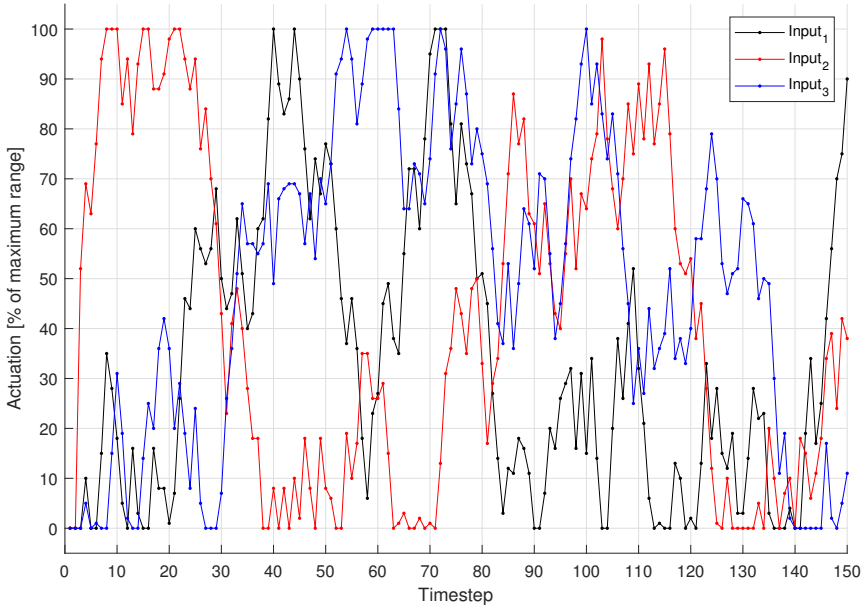
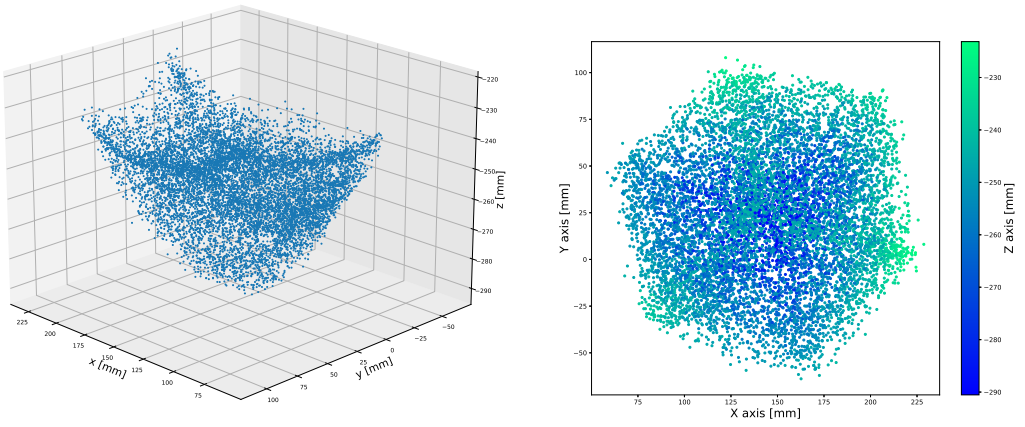
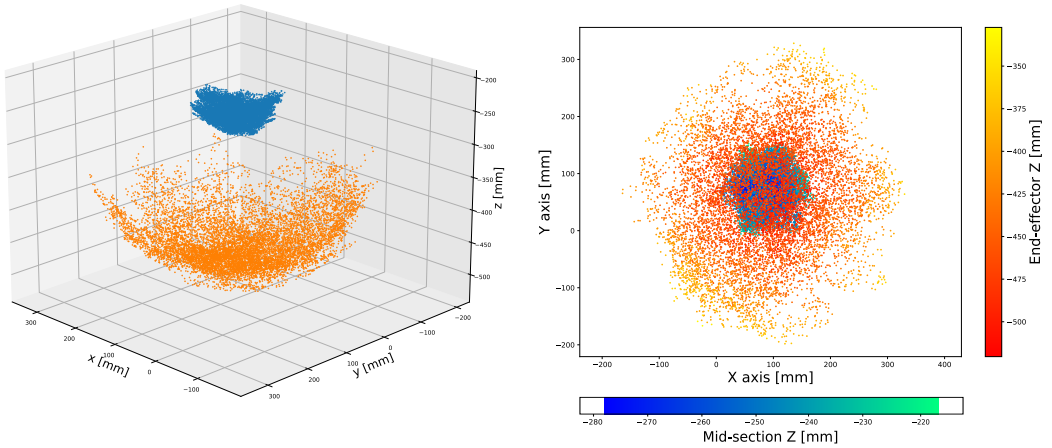


Figure 6.9: First 150 actuations for the I-Support



(a) 3D (b) 2D with heatmap

Figure 6.10: Task Space of the I-Support



(a) 3D (b) 2D with heatmap

Figure 6.11: Task Space of the AM-I-Support

6.5.2 Training results

The training initially took place entirely on a GPU Nvidia 1050Ti with 6 GB of dedicated memory and each; with this configuration 1000 training steps, which are equivalent to 100 seconds in real time on the robot, took averagely 2 seconds to complete.

This training time, even if already satisfactory, can be further reduced by parallelizing multiple environments on as many CPUs, leaving to the GPU only the duty of descending the gradients for the policy and value function optimization. I did this to optimize the the procedure, running 8 parallel rollouts of the same policy; with this distributed configuration the time taken for 1000 steps went down to 0.8 seconds, improving the training time of about 60%. The reason why the time is not reduced by a factor of 8, as one would expect, is to be found in the time overhead provoked by the communication between the 8 environments, as well as the bottlenecks created by slower rollouts.

The training for the two robots took different times: for the simpler I-Support 750 000 timesteps, equivalent to ∼10 minutes, were sufficient for the average tracking error to converge on a satisfactory result. On the more dimensionally more complex AM-I-Support, on the other hand, a much longer training of 3 000 000 timesteps, equivalent to ∼45 minutes, was required for the convergence.

In Figure 6.12b and 6.13b it's possible to observe how the total episode reward:

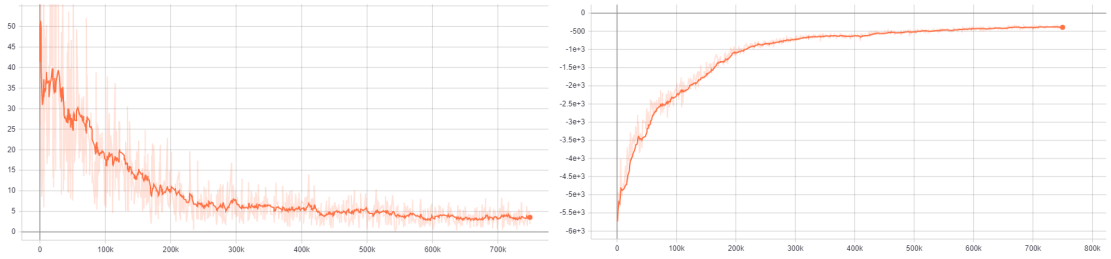
$$\frac{T}{N}\sum_{t=1}^{N} R_t = -\frac{T}{N}\sum_{t=1}^{N} \|\hat{x}^{ee}_{t+1} - x^{tar}_{t+1}\|_2 \quad \begin{cases} \text{with } N = 60...120\ ,\ T = 120 & \text{for } \textit{I-Support} \\ \text{with } N = 120...240\ ,\ T = 240 & \text{for } \textit{AM-I-Support} \end{cases} \tag{6.13}$$

stabilizes on a value of ∼ 420 for the I-Support and on a value of ∼ 1600 for the AM-I-Support. The cumulative reward is normalized on a standard length of 120 for the I-Support and 240 for the AM-I-Support, otherwise the episodes' different lengths wouldn't provide an intuitive learning curve.

In Figure 6.12a and 6.13a it's possible to see every steps' target/end-effector distance; it can be seen that, even if it decreases until an average value of about 3.5mm for the I-Support and 8.5mm for the AM-I-Support, there is a great variability between adjacent steps. This is due to the fact that,during training, the agent does not always pick the optimal action that the current policy outputs, but it samples from a normal distribution that has μ coinciding with the current optimum; the variance σ of the distribution is reduced the further it gets into the training, making the agent less prone to random exploration, but still allowing it to do so.

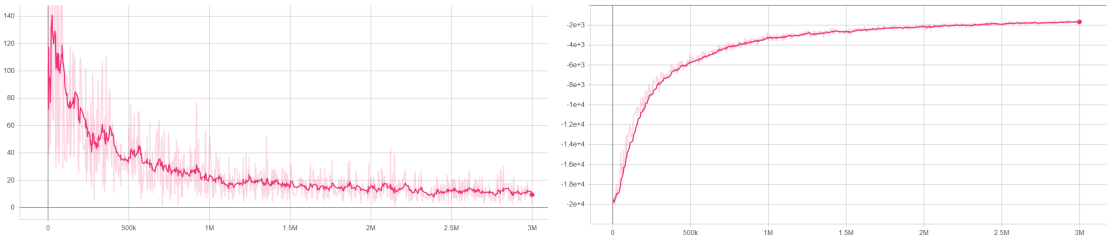
The timesteps specified for the training on the two platforms is an indicative number after which the continuation of the training leads to insignificant improvements. For example, an extra million timesteps for the training of the AM-I-Support only improves the average episode reward from 1600 to 1400 (equivalent to an improvement of less of a millimiter on the tracking error). Even if this might seem promising, testing controllers that trained for longer times proved that their performances when applied to the real robots decrease. A probable reason for this phenomenon is that after a certain amount of training the RL agent starts to overfit on the approximated environment, meaning that it adapts to the erroneously predicted positions of the end-effector to drive the error down. Although this is just an hypothesis, a proof of it comes by looking at the actuations generated in the tests run in-silico: these are much more fluttery with respect to the ones coming from a less-trained controller, a sign that the controller tries to reach every mispredicted point along

the trajectory; when transfering this on the robot, the fluttery and unsmooth actuations prove to be detrimental for the performance because of the oscillations they provoke.



(a) Distance vs. Timesteps (b) Total episode reward vs. Timesteps

Figure 6.12: Training plots for I-Support



(a) Distance vs. Timesteps (b) Total episode reward vs. Timesteps

Figure 6.13: Training plots for AM-I-Support

6.5.3 Testing results on the Forward Model

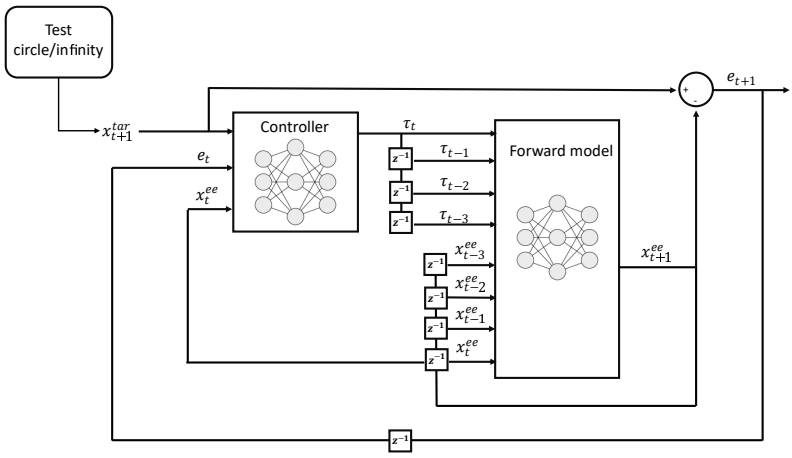


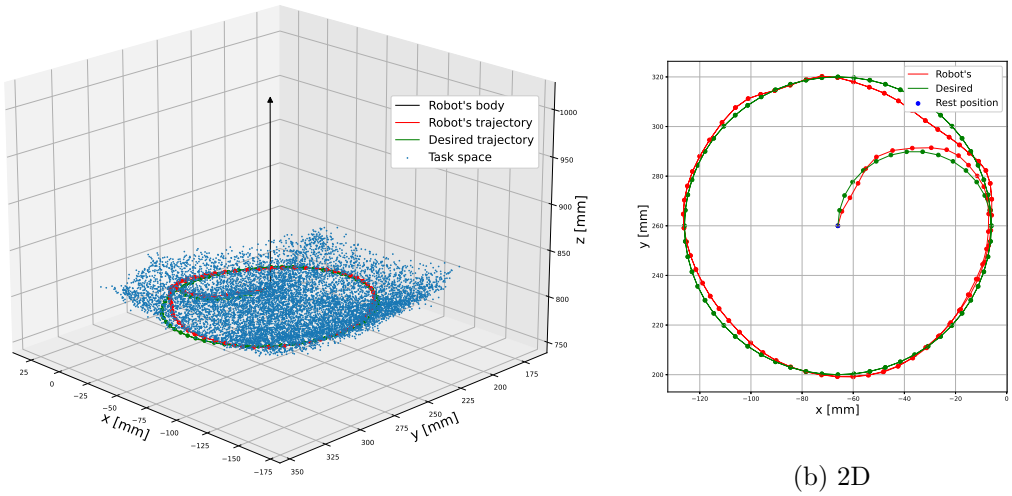
Figure 6.14: Testing overview (on Forward Model)

Once the training stops, the model is ready to be tested on two benchmarks: a circle and the infinity symbol, both made twice. The first test is run in-silico, on the same

forward model on which the controller was trained (Fig. 6.14); the utility of this is to verify whether the training procedure was capable to produce a controller that works on every subspace of the task space.

I-Support

The results for the I-Support are shown in Figure 6.15.



(a) 3D

(b) 2D

Figure 6.15: Test results on the Forward Model (I-Support)

As it can be seen, the controller works as well as it was shown in the training plots; the errors are very low, more precisely: $\|\Delta\mathbf{x}\|_2 = 3.5 \pm 2mm$ even if it's clear that they accumulate in the upper right corner of the circle from Figure 6.15b. This is further proved by looking at the actuations in Figure 6.16: τ_3 clearly saturates in the same points, given that the circle is repeated twice.

The circle benchmark is still valuable despite this problem; considering only the points for which the actuations don't saturate, as expected, returns a better error distribution: $\|\Delta\mathbf{x}\|_2 = 3.0 \pm 1.8mm$

Very similar results are obtained for the other benchmark, the infinity symbol, which I won't show for brevity's sake.

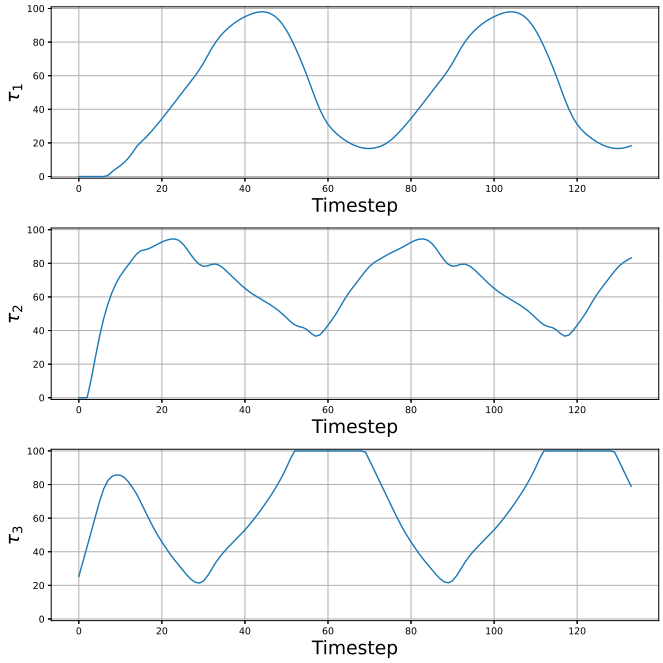
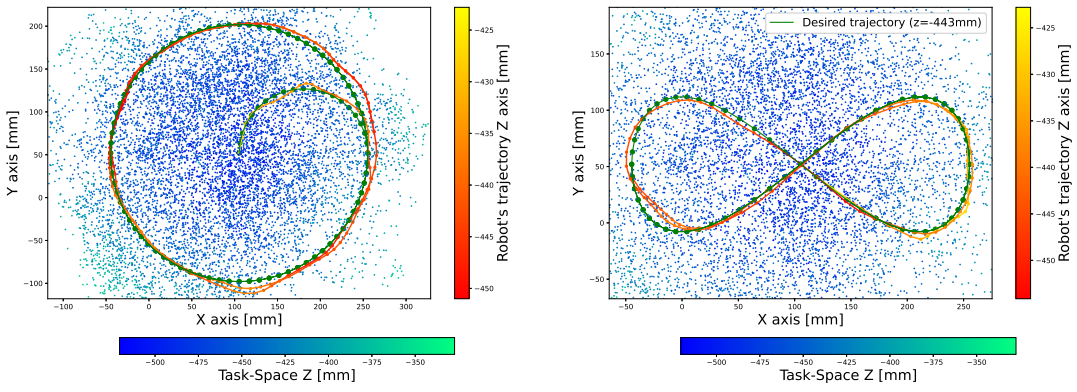


Figure 6.16: Actuations from the test on the Forward Model

AM-I-Support

Similarly, the results for the I-Support are shown in Figure 6.15.



(a) Circle (b) Infinity symbol

Figure 6.17: Test results on the Forward Model (AM-I-Support)

Once again the test reflects the final distance values in the training plots: the tracking

error while doing a circle is $\|\Delta \mathbf{x}^{ee}\|_2 = 8.5 \pm 2.5mm$, while for the infinity symbol the error is $\|\Delta \mathbf{x}^{ee}\|_2 = 6.8 \pm 2.4mm$.

6.5.4 Testing results on the Soft Robot

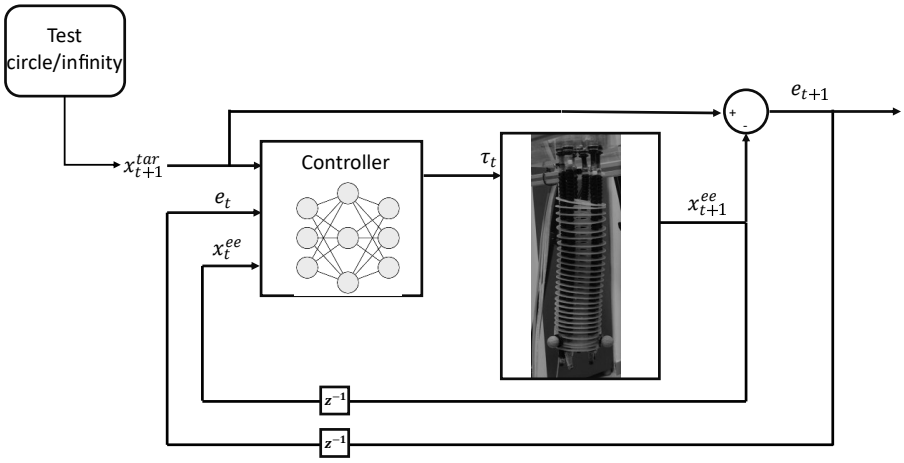


Figure 6.18: Testing overview (on the robot)

The second and most significant test is run by applying the controller to the real robot (Fig. 6.18).

After exporting the trained network from Tensorflow to MATLAB, with which the robot is given the inputs, I used the same benchmarks explained earlier as the objective trajectory; to be clear, the benchmark trajectories are actually shifted to accommodate changes in the reference system of the VICON, which happen fairly often.

I-Support

The results, as shown in Figure 6.19 and 6.20, are satisfactory: the data about the errors is shown below, in table 6.7:

Table 6.7: Errors on the robot

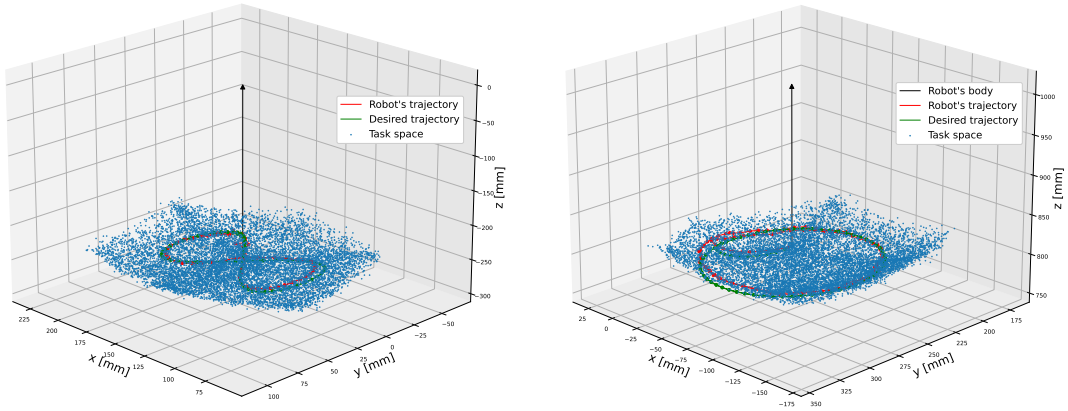
		Infinity	**Circle**
Δx	$[mm]$	3.9 ± 2.5	3.9 ± 2.5
Δy	$[mm]$	3.4 ± 1.9	3.3 ± 2.0
Δz	$[mm]$	1.2 ± 0.9	3.1 ± 2.4
$\Delta \mathbf{x}$	$[mm]$	5.9 ± 2.0	6.6 ± 2.9

Where:

$$\Delta x = \mu \pm \sigma \quad \text{with} \quad \mu = \frac{1}{N}\sum_{t=1}^{N} |x_t^{tar} - x_t^{ee}| \quad \text{and} \quad \sigma = \sqrt{\frac{1}{N}\sum_{t=1}^{N} \left(|x_t^{tar} - x_t^{ee}| - \mu\right)^2} \tag{6.14}$$

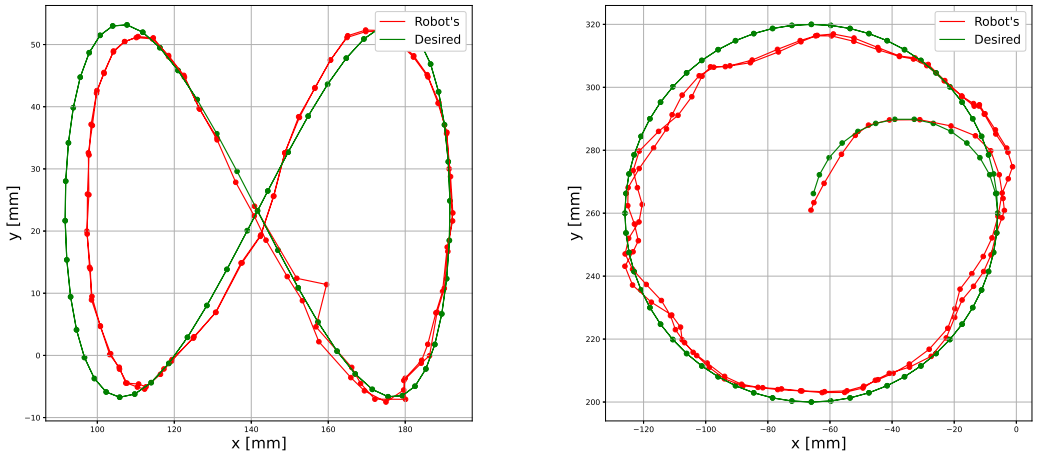
and the same for Δy and Δz, whereas:

$$\Delta \mathbf{x} = \mu \pm \sigma \quad \text{with} \quad \mu = \frac{1}{N}\sum_{t=1}^{N} \|\mathbf{x}_t^{tar} - \mathbf{x}_t^{ee}\|_2 \quad \text{and} \quad \sigma = \sqrt{\frac{1}{N}\sum_{t=1}^{N} \left(\|\mathbf{x}_t^{tar} - \mathbf{x}_t^{ee}\|_2 - \mu\right)^2} \tag{6.15}$$



(a) 3D infinity symbol

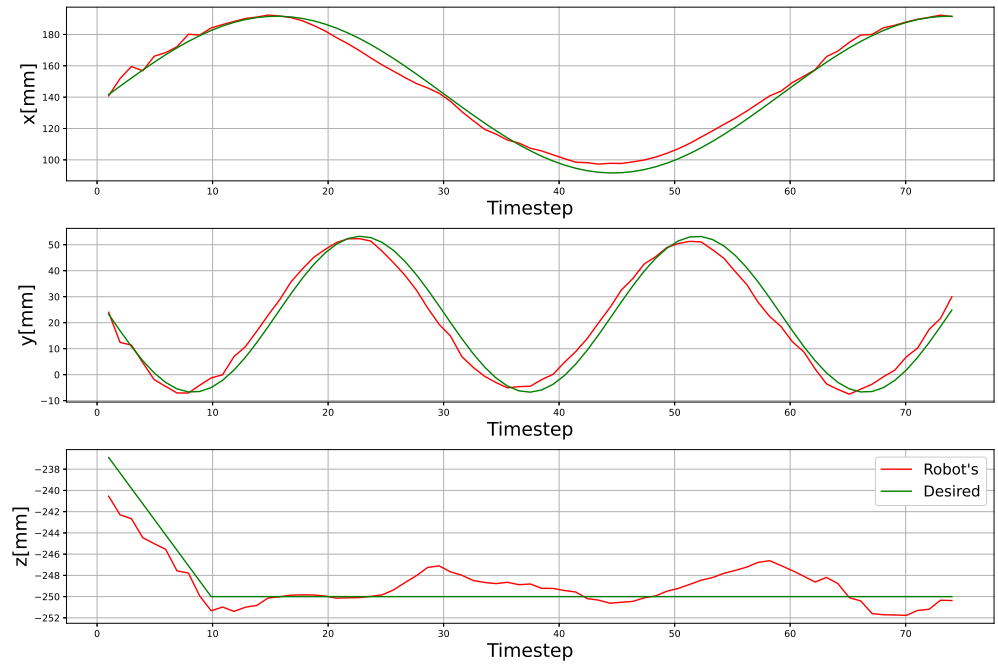
(b) 3D circle



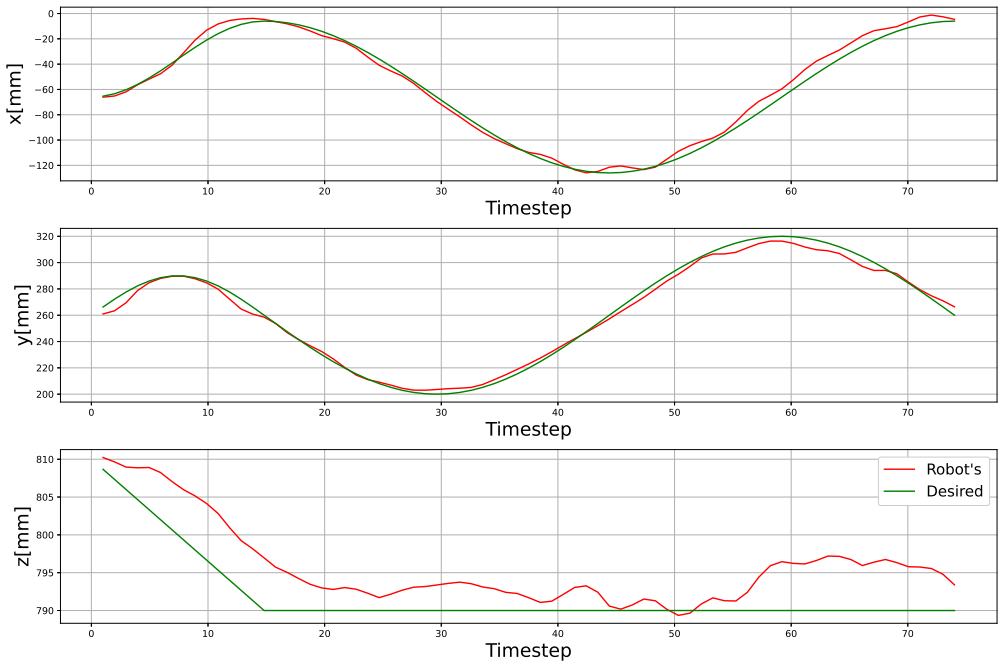
(c) 2D infinity symbol

(d) 2D circle

Figure 6.19: Test results on the robot



(a) On the infinity symbol



(b) On the circle

Figure 6.20: Δx,Δy, and Δz errors

AM-I-Support

The results on the AM-I-Support are similar to the one gotten on the I-Support.

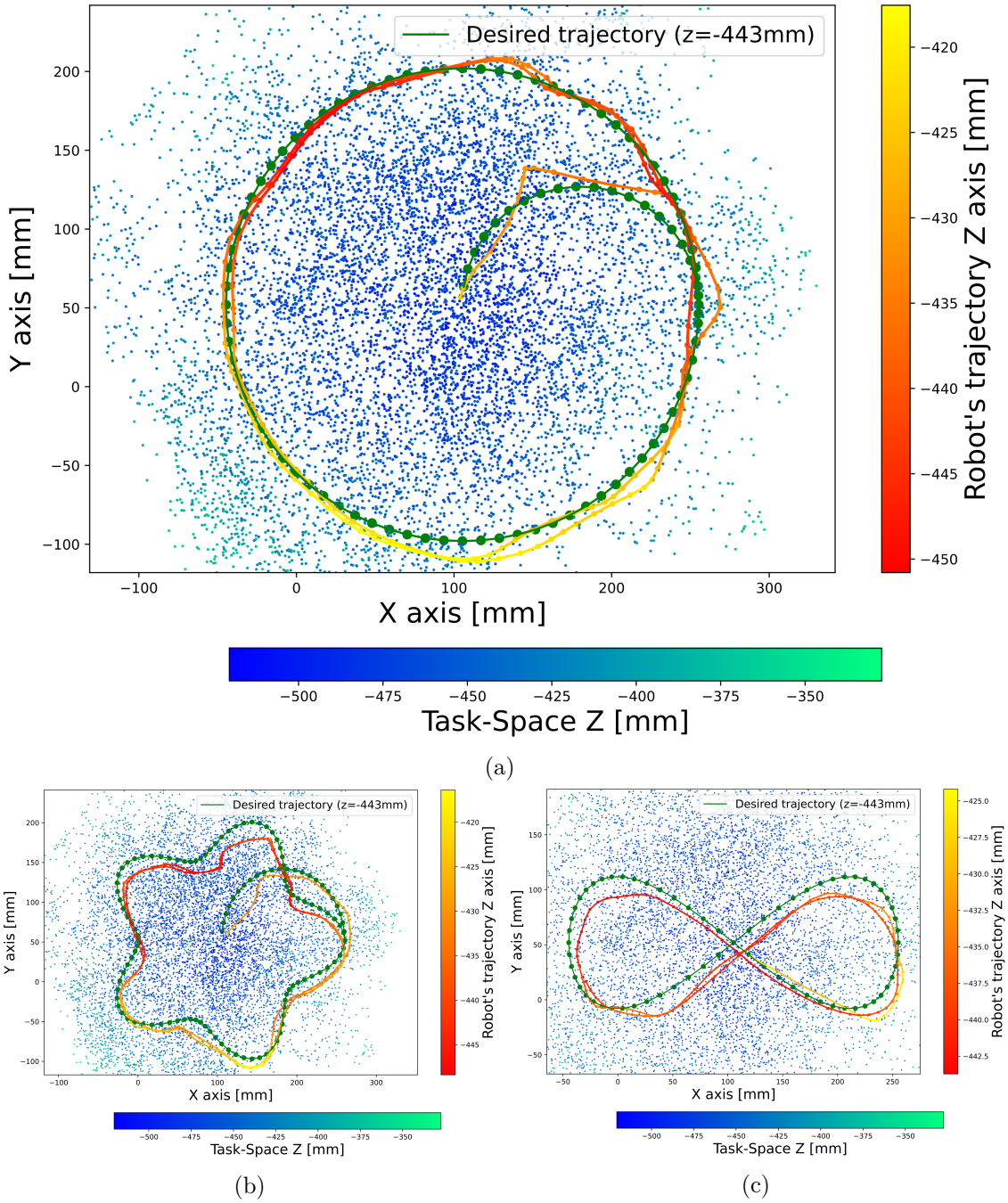


Figure 6.21: Closed-loop trajectories on the AM-I-Support

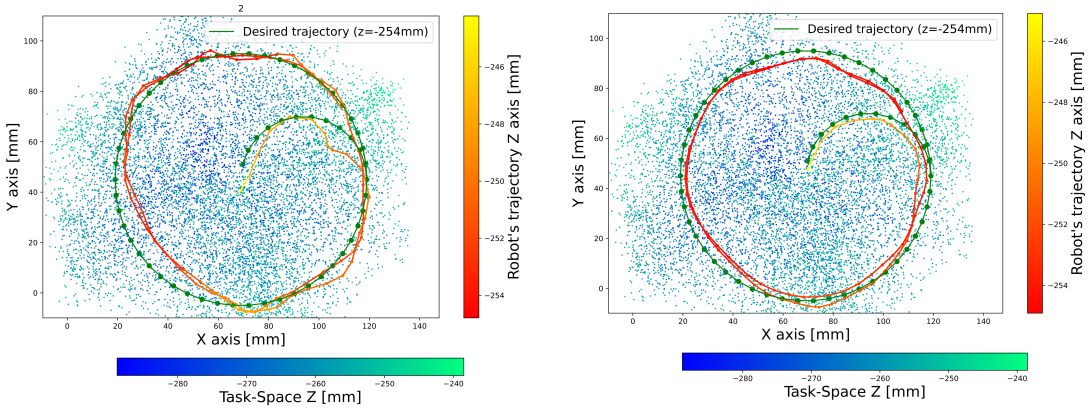
The resulting trajectories for the two benchmarks used so far, a circle of radius 15cm and

an infinity symbol of width 30cm, and a star of similar dimensions are shown in Figure 6.21. The average tracking errors for the circle and the star are 10mm and 12mm respectively, while on the infinity symbol it is slightly higher, more precisely of 15mm.

The errors are about twice as high as the ones achieved on the I-Support; this is to be expected, considering that the actuation space is double the size its simpler counterpart, and the task space is considerably larger. Ultimately, the performance of the controller is a consequence of the precision of the forward model obtained: lower prediction errors from the forward model allow the controller to train on a more accurate simulation of the model and hence attain better results.

6.5.5 Testing Results with Disturbances

Of the three variants of the controller developed to be resistant to the disturbances generated by hanging a 25g weight to the end-effector (explained in section 6.4), only the third one proved to be successful in completing a circle both in an undisturbed situation and in the disturbed one, with the weight present.



(a) Disturbed trajectory

(b) Undisturbed trajectory

Figure 6.22: Trajectories with a 25g weight attached and without

As it can be seen from Figure 6.22, the new architecture of the controller is able to distinguish between the undisturbed and the disturbed rollout thanks to the prediction error δ_m.

The average error in the cartesian space $\Delta\mathbf{x}$ (as defined in Eq.6.15) is as low as $6.2mm$ for the trajectory without disturbance, and stays in that interval for the trajectory with the disturbance, more precisely th average error for this scenario is $\Delta\mathbf{x} = 6.8mm$.

In Figure 6.23 it can be seen how the actuations are substantially different: although they keep the same overall shape, as it is to be expected, during the test trial with the weight attached they span a larger portion of the actuation-space, reaching the saturation value of 150 for τ_2.

Although the results are satisfactory, it has to be underlined that the controller is trained to reject only this precise disturbance, which was identified empirically by giving the same actuations to the manipulator in an undisturbed and disturbed scenario, and then assessing

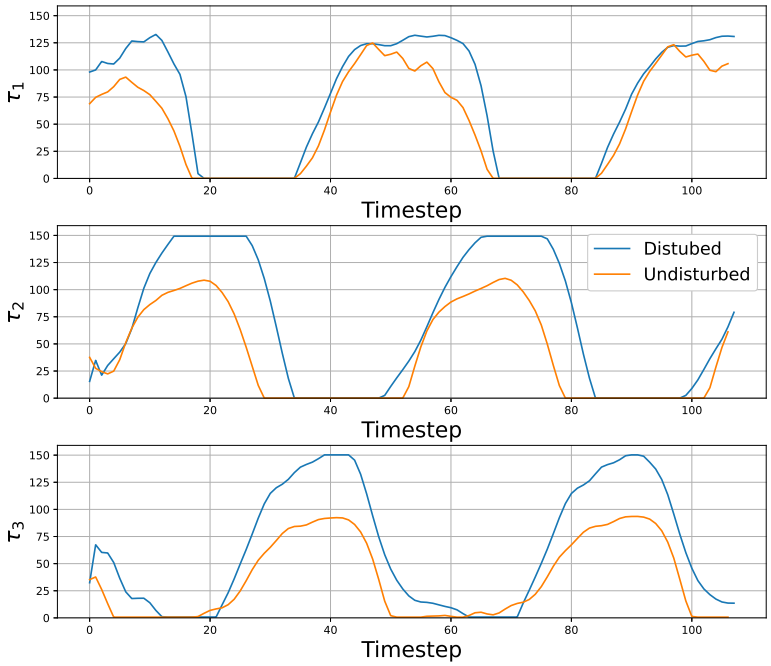


Figure 6.23: Actuations with a 25g weight attached and without

how the taskspace is consequentially shrunk. The reason why training the controller on a span of disturbances that go from a null one to the one generated by the 25g weight proves to be unfruitful is related to prediction error δ_m. In fact, although the controller sees only constant values of δ_m during training, during the test on the real manipulator said values are far from being constant because they include the inherent prediction errors of the precise, yet not perfect, forward model. These intrinsic errors cause the values of δ_m to oscillate around the ones that the controller sees during training, making different disturbed scenarios impossible to discern from one another. These oscillations of δ_m do not affect the performances of the controller trained on only two scenarios, because the distinction between the two is big enough not to be influenced by the forward model's errors.

This is a big limitation that could be solved by filtering the noise generated by the model inherent errors, feeding to the controller a constant δ_m; this is a strategy that I plan to test in the future.

Chapter 7

Conclusions

The goal of this thesis was to explore data-driven control strategies using neural networks and reinforcement learning.

I presented two different methods: one to achieve an open-loop controller using recurrent neural networks (as shown in chapter 5), and one to attain closed-loop one using a state-of-the-art reinforcement learning algorithm (as shown in chapter 6). I used two robotic platform to test these methodologies: a simpler one, the I-Support, and a more dexterous manipulator called AM-I-Support; given that one of the priorities I started with was to develop a closed-loop controller that is platform independent, I only tested this one on both robots.

The open-loop controller (in chapter 5) was the first one to be developed; despite its simplicity, it manages to produce extremely low tracking errors: under 4mm on the I-Support manipulator. While I'm more proud of the second method produced, considering that it was my first intent at the beginning of the thesis, I still consider that the procedure to achieve the open-loop controller has great potential; to make said procedure more streamlined, I would direct a possible future effort to eliminating the need to train two distinct networks (one for the kinematic and one for the dynamic behavior), potentially through a short and partial re-training of the first network. Furthermore, I did not conduct an analysis of the amount of static data needed to train the first kinematic network; considering that it is the most time-consuming part of the method, a thorough investigation on this detail would aid enormously its efficiency.

The closed-loop controller is instead obtained through the training of a reinforcement learning agent; this method manages to transfer the learned control policy from the approximated environment on which it trains to the actual robot that it is intended to control. The controller attains a tracking error of about 6mm on the I-Support, and of under 13mm on the AM-I-Support, partially proving that it is platform-independent. It must be said that although it works well on an undisturbed manipulator, it proved not to be successful in rejecting disturbances such as weights applied to the robot's body. I was later able to develop a variant of the controller that trains on an environment that simulates disturbances; this new controller showed promising results, managing to track a trajectory both in an undisturbed and in a disturbed situation.

In conclusion, I reckon I reached the goals that I set at the beginning of the thesis, succeeding in exploring and expanding the field of neural network-based controllers. The

results obtained are on par to the ones belonging to closely related researches (presented in the last part of section 1.2 regarding the state-of-the-art), often proving to achieve better results, although it is hard to make a direct comparison due to the slightly different configurations. The paper that is closest to the experimental set-up and objectives I used is [12], in which an open-loop dynamic controller obtained through learning based techniques attains worse results for the benchmarks proposed for my controllers, even though the manipulator used is a partially actuated version of an I-Support with two modules.

These methods are particularly useful for low-budget robots and, more generally, for robots whose manufacturing process does not allow the repeatability of kinematic and dynamic properties in every individual manipulator.

Bibliography

[1] T. George Thuruthel, Y. Ansari, E. Falotico, and C. Laschi, "Control strategies for soft robotic manipulators: A survey," *Soft robotics*, vol. 5, no. 2, pp. 149–163, 2018.

[2] M. W. Hannan and I. D. Walker, "Kinematics and the implementation of an elephant's trunk manipulator and other continuum style robots," *Journal of robotic systems*, vol. 20, no. 2, pp. 45–63, 2003.

[3] M. Rolf and J. J. Steil, "Constant curvature continuum kinematics as fast approximate model for the bionic handling assistant," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3440–3446, IEEE, 2012.

[4] B. A. Jones and I. D. Walker, "Kinematics for multisection continuum robots," *IEEE Transactions on Robotics*, vol. 22, no. 1, pp. 43–55, 2006.

[5] I. A. Gravagne, C. D. Rahn, and I. D. Walker, "Large deflection dynamics and control for planar continuum robots," *IEEE/ASME transactions on mechatronics*, vol. 8, no. 2, pp. 299–307, 2003.

[6] D. Trivedi, A. Lotfi, and C. D. Rahn, "Geometrically exact models for soft robotic manipulators," *IEEE Transactions on Robotics*, vol. 24, no. 4, pp. 773–780, 2008.

[7] N. Naughton, J. Sun, A. Tekinalp, G. Chowdhary, and M. Gazzola, "Elastica: A compliant mechanics environment for soft robotic control," *arXiv preprint arXiv:2009.08422*, 2020.

[8] A. Melingui, R. Merzouki, J. B. Mbede, C. Escande, B. Daachi, and N. Benoudjit, "Qualitative approach for inverse kinematic modeling of a compact bionic handling assistant trunk," in *2014 International Joint Conference on Neural Networks (IJCNN)*, pp. 754–761, IEEE, 2014.

[9] T. George Thuruthel, E. Falotico, M. Manti, A. Pratesi, M. Cianchetti, and C. Laschi, "Learning closed loop kinematic controllers for continuum manipulators in unstructured environments," *Soft robotics*, vol. 4, no. 3, pp. 285–296, 2017.

[10] V. Falkenhahn, A. Hildebrandt, R. Neumann, and O. Sawodny, "Dynamic control of the bionic handling assistant," *IEEE/ASME Transactions on Mechatronics*, vol. 22, no. 1, pp. 6–17, 2016.

[11] A. D. Kapadia, I. D. Walker, D. M. Dawson, and E. Tatlicioglu, "A model-based sliding mode controller for extensible continuum robots," in *Proceedings of the 9th WSEAS international conference on Signal processing, robotics and automation*, pp. 113–120, 2010.

[12] T. G. Thuruthel, E. Falotico, M. Manti, and C. Laschi, "Stable open loop control of soft robotic manipulators," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 1292–1298, 2018.

[13] T. G. Thuruthel, E. Falotico, F. Renda, and C. Laschi, "Model-based reinforcement learning for closed-loop dynamic control of soft robotic manipulators," *IEEE Transactions on Robotics*, vol. 35, no. 1, pp. 124–134, 2018.

[14] L. Arleo, G. Stano, G. Percoco, and M. Cianchetti, "I-support soft arm for assistance tasks: a new manufacturing approach based on 3d printing and characterization," *Progress in Additive Manufacturing*, pp. 1–14, 2020.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] M. Nielsen, "Neural networks and deep learning." http://neuralnetworksanddeeplearning.com/index.html.

[17] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[18] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[19] M. Minsky and S. Papert, "An introduction to computational geometry," *Cambridge tiass., HIT*, 1969.

[20] J. Schmidhuber, "Who invented backpropagation?." https://people.idsia.ch/~juergen/who-invented-backpropagation.html.

[21] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, p. 1735–1780, Nov. 1997.

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