Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Master Degree in Automation and Control Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

POLITECNICO
MILANO 1863

A cartesian model-free controller for reaching and throwing implemented on a humanoid robot

Advisor: Prof. Matteo MATTEUCCI
Co-Advisor: Dr. Egidio FALOTICO
Co-Advisor: Dr. Lorenzo VANNUCCI

Thesis by:
Gabriele GIUDICI Matr. 898173

Academic Year 2018–2019

There's a secret mission in uncharted space.

Contents

List of Figures

List of Tables

Abstract

Il mondo della robotica sta vertendo sulla progettazione di robot sempre più complessi e il controllo di questi robot attraverso i metodi più tradizionali sta diventando gradualmente non sufficiente. Per questa ragione sta diventando sempre più comune l'utilizzo di neuro controllori basati sulla teoria del Machine Learning (ML). Questi controllori risultano particolarmente utili nel campo dei robot umanoidi per i quali è usuale svolgere compiti non banali in ambienti complessi. In questo lavoro viene presentato un nuovo neuro controllore basato sull'utilizzo di reti neurali artificiali (RNN) in grado di approssimare la funzione dinamica diretta di un robot generico e in grado di svolgere compiti differenti variando solamente la funzione di costo associata al compito stesso. Questo lavoro è la conseguente continuazione di lavori precedentemente svolti in letteratura ed applicati a robot morbidi. Lo scopo che questo lavoro si pone è quello di generalizzare il medesimo approccio e validarlo su un robot di natura differente come un robot umanoide. Inoltre viene introdotto un confronto tra modelli di reti neurali artificiali differenti, in particolare viene fatto un confronto nella capacità di approssimare il modello dinamico diretto di due RNN differenti ovvero le reti neurali 'Nonlinear autoregressive exogenous model (NARX)' e le reti neurali 'Long short-term memory (LSTM)'. Il confronto presentato tra questi due modelli avviene attraverso una implementazione sul robot umanoide iCub svolgendo il compito di raggiungere un punto nello spazio Cartesiano operativo del robot. Infine, il lavoro mostra come il modello di controllo può essere esteso per lanciare un oggetto in un punto prefissato.

Abstract

The world of robotics is centering on the construction of increasingly complex robots and the control of robots through classic methods becomes gradually insufficient. For this reason, neuro controllers based on machine learning theory are becoming increasingly important. These controllers are really useful in the field of humanoid robotics for which it is usual to perform complex tasks in a complex environment. In this work we present a new neuro-controller based on recurrent artificial neural networks capable of approximating the forward dynamic model of a generic robot and capable of performing various tasks by only changing objective function. This work is the consequent continuation of previous works using a soft robot . These works have brought a great innovation in the field of literature in the approximation of the model and control of soft robots. This work aims to generalize the same approach and validate it on robots of different nature such as humanoid robots. Furthermore, we want to introduce a comparison between different recurrent artificial neural network models, in particular the relationship between the 'Nonlinear autoregressive exogenous model (NARX)' and the 'Long short-term memory (LSTM)' models. We present a comparison between the two RNN through an implementation on the iCub humanoid robot performing reaching tasks in the Cartesian space. In addition, the work shows how to extend this controller for throwing an object in a specific point.

Sommario

Al giorno d'oggi il mondo della ricerca scientifica e industriale si sta evolvendo in sinergia verso una direzione dettata dall'intelligenza artificiale. In particolare un ramo della ricerca, in collaborazione con molte realtà industriali, sta cercando di dare un corpo a questa nuova forma di intelligenza. In questo senso non è più futuristico parlare di robot bioispirati, ovvero quei robot che almeno morfologicamente si ispirano ad essere viventi come animali (topi, polipi, cani, etc.) o addirittura ad esseri umani.

Questo sviluppo tecnologico sta portando a un progresso non più solo settoriale, ma sta rendendo trasversale il progresso di diverse aree; per esempio attraverso lo studio e la progettazione di nuovi materiali, più morbidi e allo stesso tempo più resistenti, come siliconi; di nuovi processori sempre più potenti e con dimensioni ridotte; di tecnologie di visione e riconoscimento immagini e di analisi statistiche.

Tutte queste tecnologie stanno trovando sempre più applicazione pratica nei robot industriali, con l'obiettivo di progettare robot antropomorfi sempre migliori, i quali potrebbero risultare le migliori macchine di lavoro da affiancare agli operatori umani.

Questi robot dai molteplici gradi di libertà sono macchine progettate per svolgere compiti differenti e sempre più complessi, ma introducono diverse criticità e componenti con comportamenti non lineari che stanno portando a un conseguente aumento della complessità del sistema da modellare.

Con l'aumento di questa complessità potrebbe non essere più sufficiente fare affidamento solo sulle strategie 'classiche' di modellazione che vengono utilizzate per i manipolatori tradizionali, poiché abbiamo a che fare con sistemi integrati. In questo senso diventa davvero complicato riuscire a modellizare il problema in dettaglio partendo dalla formulazione di modelli cinematici e dinamici che risultano diventare sempre meno banali. Alla luce di ciò, l'uso di approcci basati sulla teoria del Machine Learning hanno trovato rapidamente spazio nel mondo della robotica umanoide durante gli ultimi anni.

Dapprima, queste tecniche sono state ampiamente utilizzate per la loro ca-

pacità di svolgere compiti di ottimizzazione e classificazione dei dati relativi all'uso dei robot, portando a un controllo di tipo adattivo in grado di fornire una previsione e un'interpretazione in tempo reale dei dati raccolti dai sensori. Ultimamente le tecniche basate sul Machine Learning stanno trovando sempre più applicazione in molti campi della ricerca e l'uso massiccio di queste tecniche statistiche sta mostrando tutta la loro capacità nel risolvere vari problemi affrontati finora con altri metodi. Uno modo innovativo di utilizzare queste tecniche è sfruttare la capacità di apprendimento di questi modelli per approssimare i modelli dinamici di robot complessi.

Lo scopo di questo lavoro è quello di estendere le opere presentate da Thuruthel et al. [1], [2] nei quali viene fatta una approssimazione del modello dinamico e poi il controllo di un robot morbido. Il nostro obbiettivo è quello di riproporre il medesimo approccio utilizzando un robot umanoide per dimostrare come sia possibile riutilizzare questa strategia per controllare robot di diversa natura. Inoltre si vuole mostrare che questo approccio sia applicabile indipendentemente dal tipo di riferimento della variabile di controllo dei motori (riferimento di forza, riferimento di velocità) e che tutto ciò possa essere utile per svolgere compiti differenti.

Con questi obbiettivi è stato implemenato un controllore sfruttando un approccio 'model-free', basato sull'utilizzo di reti neurali artificiali, per poter controllare il robot nello spazio Cartesiano. Per poter utilizzare le reti neurali ricorrentI (RNNs) è necessario avere a disposizione una grande quantità di dati descrittivi del problema che si vuole approssimare, in questo caso dunque relativi alla dinamica del robot. Per creare una collezione di dati sufficentemente ricca di informazioni utili ad approssimare il modello dinamico diretto si è fatto affidamento ad un approccio chiamato 'motor-babbling', con il quale attraverso una generazione casuale di comandi motori è possibile creare una relazione tra questi comandi e la posizione cartesiana risultante del centro della mano del robot (end effector) compiuto il movimento associato al comando motorio. Infine si è sviluppato un processo in grado di ricavare i comandi motori necessari per permettere al robot di svolgere compiti diffenti. Questo processo è stato sviluppato utilizzando un metodo numerico di ottimizzazione in grado di ricavare i comandi motori necessari a svolgere un determinato compito utilizzando il modello approssimato, che non subirà ulteriore variazioni durante questo processo, e una funzione di costo descrittiva del compito stesso.

Pertanto, la metodologia proposta prevede una fase di pre-allenamento attraverso motor babbling, una fase di addestramento della rete neurale ricorrente e una fase in cui proveremo a svolgere diversi compiti (come raggiungere un punto o lanciare una sfera) e si vuole mostrare come sia possibile eseguire questi compiti su robot di diversa natura, cambiando solo la fase di pre-allenamento

del motor babbling. Infatti utilizzando questo approccio si vuole eliminare il più possibile la necessità di conoscere i parametri dinamici del robot. Inoltre si vuole analizzare l'efficacia del metodo al variare del tipo di rete neurale, in questo modo si vuole analizzare se esista una differenza sostanziale nell'uso di un modello di rete piuttosto che di un'altro.

Chapter 1

Introduction

1.1 Motivation

Nowadays the world of scientific and industrial research is evolving in synergy towards a direction driven by artificial intelligence. In particular, a branch of research, in collaboration with many industrial realities, is trying to give a body to this type of intelligence. In this sense, it is no longer futuristic to speak of bio-inspired robots, or robots that at least morphologically are inspired by living beings as animals (mice, octopuses, dogs, etc.) or human beings [3].

This leading technology is no longer just sectoral, but it is making progress across different areas; for example through the study and design of new materials and actuators, combined with new, smaller and more powerful processors and innovative vision and image recognition technologies.

All these technologies are finding more and more practical application in industrial robotics, with the scientific aim to design better anthropomorphic robots, which are potentially the best machines to work alongside with humans. [4], [5]

These robots with multiple degrees of freedom are designed to perform different tasks, but they present different non-linearities leading to a consequent increase in the complexity of the system to be modeled.

It is no longer possible to rely on 'classic' models used for manipulators, since we are dealing with integrated systems. In this sense it becomes really complicated to be able to model the problem in detail through descriptive equations of kinematics and dynamics only. In light of this, the use of machine learning approaches, has risen at a rapid pace in the world of humanoid robotics in the last years.

Machine learning approaches have demonstrated to be a powerful tool to be used as an alternative to analytical models for the estimation/prediction of

Figure 1.1: Amico,Comau

the dynamics or kinematics of the robot models. In this work we will focus on the implementation of a model-free controller in the Cartesian space based on a recurrent neural networks (RNN) that is able to learn the forward dynamic model of the arm of a humanoid robot to perform reaching and throwing tasks.

1.2 Humanoid Robots

The father of the term Robot was the playwright Karel Chapek who took his cue from the Czech word 'Robota' which means 'hard work' or 'forced labor' and later Isac Asimov, whose centenary of birth is celebrated in 2020, introduced the concept of Robotics through its science fiction narrative.

Among the anthropomorphic robots it is necessary to distinguish three types: Cobots, Androids and Humanoids [6]. The first are essentially industrial robots, such as mechanical arms, that emulate human actions and that collaborate with humans to perform a task, these robots can have the peculiarity of learning while working through the integration of multiple sensors and algorithms capable to give meaning to the data collected. Androids like Sophia [7](Figure 1.2) are designed to emulate human beings primarily in appearance, therefore focusing mainly on the tissues and on the gestures and expressiveness. In parallel, they are tools used to give a physical implementation to different branches of machine learning that study and analyze computers' understanding of emotional stimuli and recognition of expressions and everything related to the most emotional

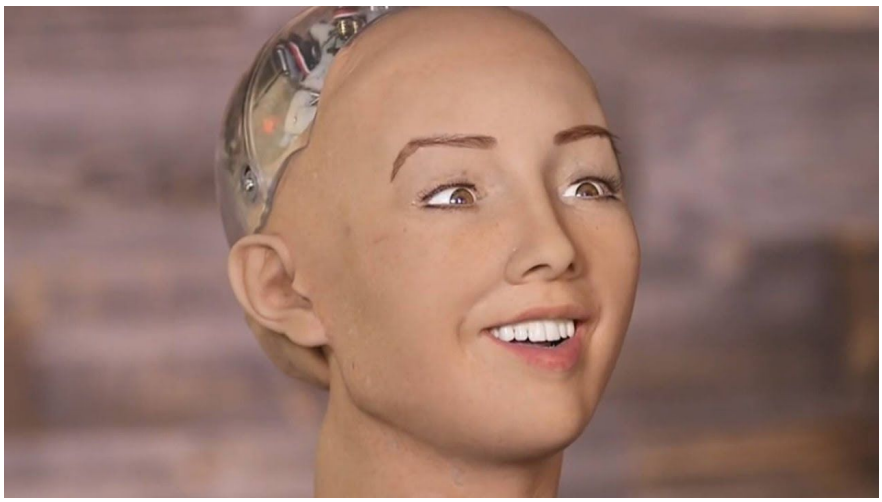

Figure 1.2: Sophia,Hanson Robotics Limited.

part of human expression.

Humanoids are inspired by the ability to move and perform actions capable of interacting with the surrounding reality. For this purpose, humanoids have as main characteristics two legs, two arms, a core, a head and more accurate grippers, as could be reproductions of hands. These robots are more complex and besides being collaborative they could have a certain level of autonomy.

Obviously this classification is purely illustrative and in recent developments it is becoming increasingly difficult to distinguish between these types of robots due to the great integration of the various functionalities in the developed robots.

In our work we will deal with various issues concerning those that we have classified as humanoid and as previously said, the concept of humanoid robot was only science fiction, but in recent decades several research centers and companies have tried to give body to these fantasies.

We can consider the first real attempt of humanoid robotics with the presentation in the 70s of Wabot-1 [8], a robot designed by a team from the Waseda University of Tokyo that included in addition to arms and legs also visual sensors.

And ten years later from WABOT-2 [9] 'In 1970, four laboratories in Waseda University's School of Science and Engineering teamed up and started the Wabot project. The efforts were led by Professor Ichiro Kato, a pioneer in humanoid robotics. In 1973, the group unveiled the Wabot 1. It was the world's first full-scale anthropomorphic robot, capable of walking with a quasi-dynamic gait. It could also speak and grasp objects. In 1980, the group started working on a new robot. In 1984, they introduced Wabot 2, a 50-degrees-of-freedom humanoid that was able to read musical scores and play an electronic keyboard.'

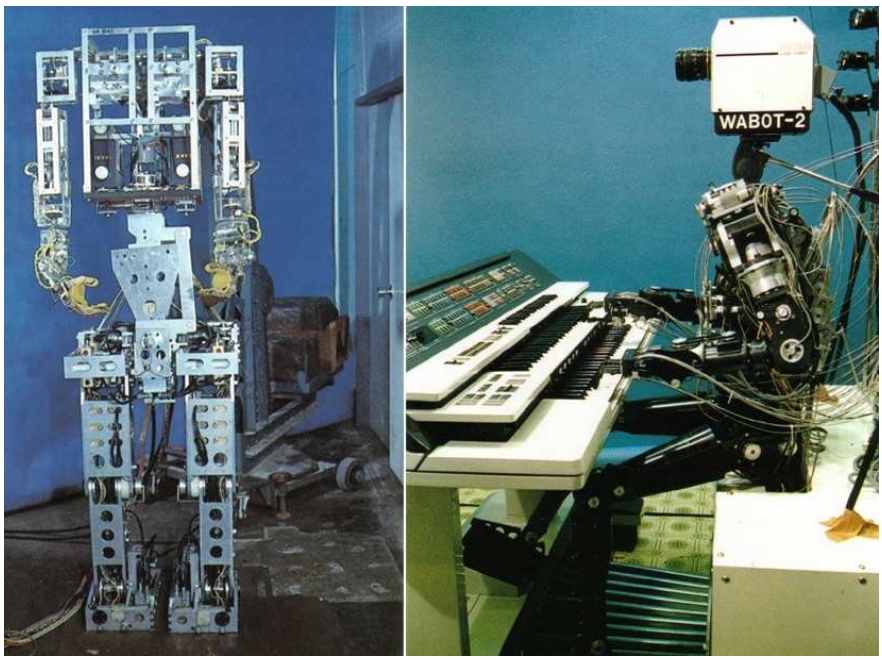

Figure 1.3: Wabot1 and Wabot2,Waseda University.

In our work all the simulations and related experiments are performed using an iCub (Figure:1.4), an Italian humanoid robot created by the IIT of Genoa and which has as its purpose the use in Research [10]. This work is continuously on a previous work [1] [2] which had as a study tool a soft robot (or a robot designed with soft materials such as silicone).

1.3 State of the art for Cartesian model-free control of the arm

In this work, we use a machine learning method to approximate the function that maps the motor control commands into the Cartesian position of the end effector. In literature, several studies have focused on learning the angular position of the individual joints so that the Cartesian position of the tip of the robotic arm can be obtained using the kinematic relationship [11]. Other studies instead use neural networks to estimate some unknown parameters of the dynamic model, such as for example the coefficients of friction or other non-linear terms difficult to identify [12] or to ensure a performance of the position control of the joints [13]. Khan et al. [14] presents the implementation of a model-free Q-learning based discrete model reference compliance controller for a humanoid robot arm capable of controlling the position of the hand of a

Figure 1.4: iCub, Istituto Italiano di Tecnologia

humanoid robot through complex Reinforcment Learning(RL) and Computer Vision algorithms through visual feedback of the position of the end effector.

Braganza et al. [15] presents a controller for continuum robots, which utilizes a neural network feedforward component to compensate for dynamic uncertainties. Falkenhahn et al. [16] presents a dynamic controller in actuator space in order to provide a good and intuitive dynamic behavior of the manipulator that in this case was a soft robot.

An interesting elaborate is that of Plooij et al. [17] presents an approach to perform repetitive tasks with robotic arms, without the need for feedback.

Starting from the works of Thuruthel et al. [1] who tried to learn the forward dynamics of a soft robot through the use of a NARX neural network and then to control the same robot through a numerical optimization algorithm [2], we focused on extending this work by using a rigid humanoid robot to validate the method on a different type of robot, with the future aim of being able to use this approach to generate throwing movements.

In fact, having robots capable of throwing would broaden the field of application of robots. There are many ways to implement this feature, but today it is still an open question, in the following we will show some of the most interesting works that have been implemented and then we will explain how our approach can bring innovation in the answer to this question.

Sugimoto et al. [18] use a RL method and the desired angular velocities are learned through the trials. A simple nominal trajectory was provided for the elbow joint that slowly extended it. The reward function is update each trials

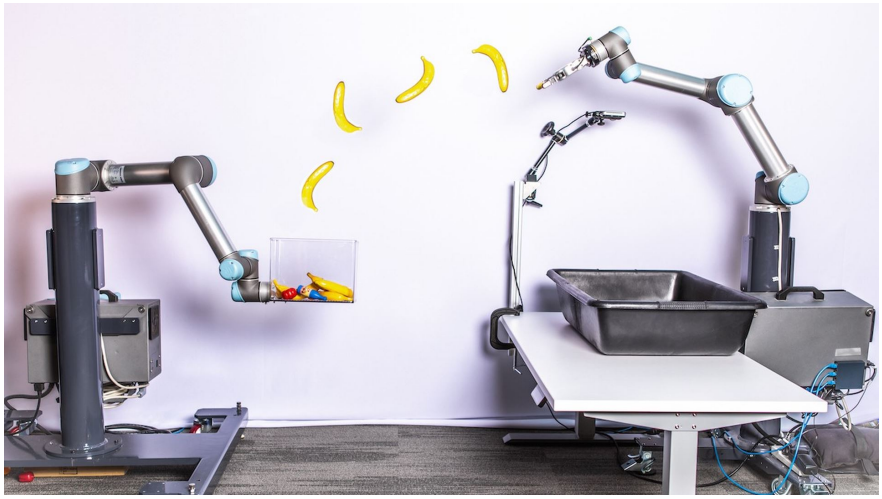

Figure 1.5: TossingBot,Google.

and at the end of the learning the robot has acquired the capability of repeat a successful basketball-shooting task.

Google's Tossing bot [19]is able to grasp and throw arbitrary objects into boxes located outside its maximum reach range at 500+ mean picks per hour (600+ grasps per hour with 85% throwing accuracy); and generalizes to new objects and target locations. They propose an end-to-end formulation that jointly learns to infer control parameters for grasping and throwing motion primitives from visual observations (images of arbitrary objects in a bin) through trial and error.

TOYOTA CUE-3 [20] generates a three-dimensional image of where the basket is using sensors on its torso and then adjusts the motors inside its arm and knees to give the shot the right angle and propulsion for a perfect shot each time

1.4 Thesis objectives

The purpose of this work is to extend the works presented by Thuruthel et al. [1], [2] in which the author proposed an approximation of the dynamic model, learned with RNN, and then the control of a soft robot to perform a reaching task. Our goal is implement this approach in a humanoid robot arm to demonstrate how it is possible to reuse this strategy to control different robots. In addition, we want to show that this approach is applicable regardless of the control mode (force control, velocity control) and how to extend this approach to perform a throwing task. Another objective is to analyze whether there is

Figure 1.6: CUE-3,Toyota.

a substantial difference in using different RNN models for learning the forward dynamics.

The proposed methodology involves a pre-training phase (data collection through motor babbling), a training phase of the RNN and an optimization phase in which we generate inputs for the robot arm actuators according to the task to be performed. By using this approach, we want to eliminate as much as possible the need to know the dynamic parameters of the robot.

Chapter 2

Proposed model

2.1 Control Scheme

The idea behind this work is to develop a control model like the one shown in the Figure 2.1. The goal is to obtain the values of the control variables (force, velocity, etc.) necessary to reach a Cartesian point in the robot's task space or to throw an object into a target position using an optimization algorithm. We propose a model-free approach that uses a RNN trained through via motor-babbling and able to associate the motor commands to the movement of end effector. By using only the position of the target in space as input to the system, an optimal solution for the task is computed through an optimization algorithm that searches for the best trajectory.

2.2 Motor Babbling

Motor babbling is one of the techniques used to perform the self-learning part of complex robots such as humanoid robots or soft robots. This process consists in generating random stimuli on some or all the motors of a part or of the whole robot. In our case, if we want to perform tasks that basically concern the use of only one arm, we will use this technique only on one of them. The technique is similar to a task space exploration algorithm, in fact for better learning it is necessary to have a large amount of data available. Then random trajectories of different types of motor commands are generated (forces, position, velocity) and the association between this stimulus and the final position of the end effector is recorded. For our study it is interesting to know the position of the end effector in Cartesian terms as well as with respect to the rotation of the joint, because

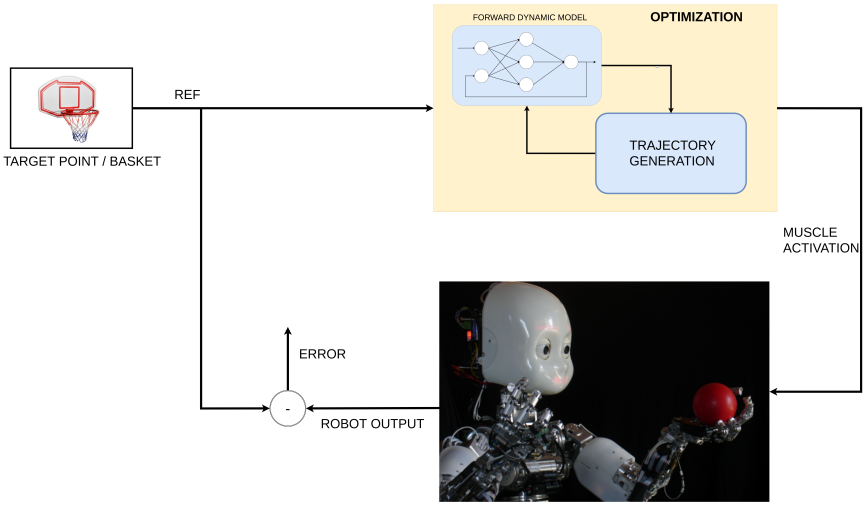

Figure 2.1: Control scheme architecture

the end effector is the final part of the arm that corresponds to the palm of the hand or gripper where the object is held .

A forward dynamic model of the arm is learned through motor babbling, during which the arm is repeatedly moved into random postures. The arm stops moving only when a pre-specified number of training samples are gathered [21]. The size of the data set is related to the size of the joint vector.

The trajectory generation strategy is aimed at the use of machine learning methods, and for this work in particular in training neural networks. Neural networks of different types may require different babbling strategies, for example a NARX type network could accept a long sequence of babbling as data sets, while other networks, such as LSTM, require many shorter series. In the following chapters we will make a comparison between the use of different approaches.

2.3 Learning the forward dynamics

In this section, referring to the work of Thuruthel et al. [1] we want to briefly describe how the model we want to learn is formulated. The kinematic function of the robot can be written as:

$$x = F(q) \tag{2.1}$$

where x is the Cartesian position within the task space and $q \in R^7$ is a vector of θ which describes a configuration of the robotic arm, we can therefore always replace in our formulation all instances of q with an instance of x. Taking then

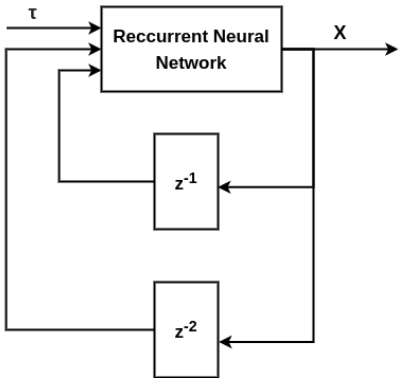

Figure 2.2: Dynamic model architecture using a RNN model.

the direct dynamic model:

$$M(\boldsymbol{q})\ddot{\boldsymbol{q}} + V(\boldsymbol{q})\dot{\boldsymbol{q}} + P(\boldsymbol{q}) = \tau \tag{2.2}$$

we can turn it into:

$$\bar{M}(\boldsymbol{x})\ddot{\boldsymbol{x}} + \bar{V}(\boldsymbol{x})\dot{\boldsymbol{x}} + \bar{P}(\boldsymbol{x}) = \tau \tag{2.3}$$

Here,$\tau \in \mathbb{R}^m$ are the control inputs. M, V and P represent the inertia matrix, centripetal–Coriolis forces and potential energy stored due to gravity/deformation,respectively. $\bar{M}, \bar{V}$ and $\bar{P}$ are the corresponding matrices obtained after the transformation. This implies that, under these assumptions, it is always possible to learn a directmapping between the states of the task space variables and the control inputs:

$$(\boldsymbol{\tau}, \boldsymbol{x}, \dot{\boldsymbol{x}}) \rightarrow \ddot{\boldsymbol{x}} \tag{2.4}$$

knowing the structure of a recurrent neural network it is possible to rewrite a new mapping formula between the motor commands and the end position of the end effector

$$(\boldsymbol{\tau}^{\mathrm{c}}, \boldsymbol{x}^{\mathrm{p}}, \boldsymbol{x}^{\mathrm{c}}) \rightarrow \boldsymbol{x}^{\mathrm{n}} \tag{2.5}$$

2.4 Recurrent Neural Network

The human nervous system contains billions of processing units: the neurons. These processing units interact each other through thousands of synapses. The nervous system is in fact a fast parallel processing system, capable of solving many tasks. Same as for Machine Learning systems, the brain learns from

experience through the concept of plasticity, namely the ability of adapting the system to its environment. For instance, the nervous system can change the strength of interconnections or create new ones in order to adapt to new observations.

Artificial Neural Networks (ANNs) are a class of ML models. They share the same principles of the biological neural networks: an ANN is a weighted directed graph whose nodes are the neurons and edges are the synapses. There are several types of ANNs that are distinguished by the connections between the neurons and the computations made by the processing unit.

Recurrent Neural Networks are a class of networks suitable for dealing with sequential data. They contains feedback loops which allow the network to maintain an internal state. The internal state implicitly encodes the history of past computations. In other words, it acts as a memory of already seen input and it permits the network to exhibit dynamic behavior. Hence, RNNs are dynamical systems which map sequences to sequences.

There are two possible causes of poor performance in NN: underfitting and overfitting. The model is in an underfitting situation when it cannot model neither the training set nor the test set. This could mean that the model is not enough complex to succeed in the task or that the learning algorithm failed in the searching of hypothesis.

The model is overfitting the data when it reaches great performance on the training set but it cannot generalize on new data. Overfitting must be avoided by evaluating the learning process on a third partition: the validation set. Its role is similar to that of test set, which is the evaluation of the system's performance, but it can be used more than once, even for training. Tracking the validation error, it is possible to understand if the model is overfitting and act accordingly with regularization techniques. Due to these critical issues, depending on the complexity of the artificial neural network used, a model selection phase may be necessary. In this phase the different parameters of the network are adjusted to improve performance and ensure that the model of the selected network is not affected by phenomena of underfitting or overfitting.

For this treatise we have used different types of neural networks, we will briefly illustrate them theoretically in these sections and then we will show their use for our work.

2.4.1 NARX

The NARX or nonlinear autoregressive network with exogenous inputs is a recurrent dynamic network, with a feedback connection that closes a loop with several layers of the network inside. The NARX model is based on the linear

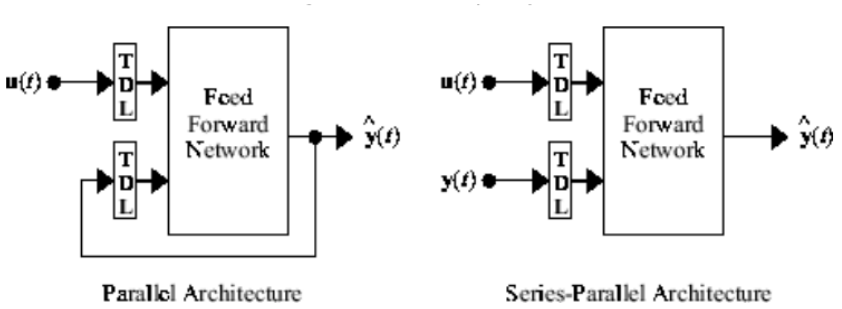

Figure 2.3: NARX architecture

ARX model, which is commonly used in time-series modeling. The defining equation for the NARX model is

$$y(t) = f(y(t-1), y(t-2), \ldots, y(t-n_y), u(t-1), u(t-2), \ldots, u(t-n_u)) \quad (2.6)$$

The next value of the output $y(t)$ is regressed to previous values of the output signal and (optionally) to the previous values of the signal of the exogenous input data set. The NARX is a type of model that using a feedforward neural network allows to approximate a function f and finds many uses, in particular it is possible to use it as a predictor, to predict the next value of the input signal, or much more useful for our purpose the modeling of nonlinear dynamic systems.

Before the training phase, which will be explained later, it is good to show a peculiarity of this model. In fact, the network can be used and/or trained in two different configurations, namely in open and closed loops. The output of the NARX network can be considered as an estimation of the output of some nonlinear dynamic system that we are trying to model. This output is called 'target output' and is an estimate made from the network input data. The network takes as inputs the model inputs, i.e. the random data of the babbling, and the associated outputs always collected in the babbling phase, called 'true output', which will be available only during the training phase. You could create a series-parallel (open loop) architecture as shown, in which the true output is used instead of feeding back the estimated output. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the resulting network has a purely feedforward architecture, and static backpropagation can be used for training. Now the training phase could be considered concluded or, by closing the network replacing the true output with target output, we could continue the training.

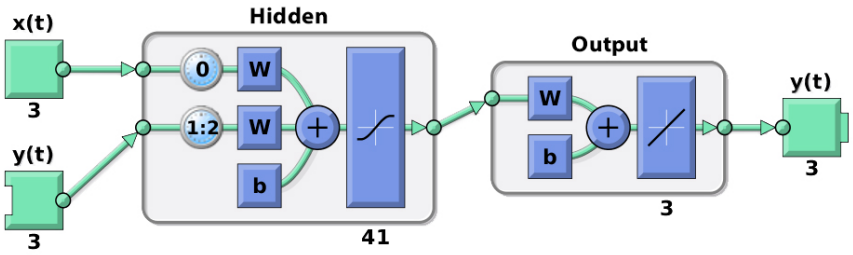

Figure 2.4: NARX Open Loop.

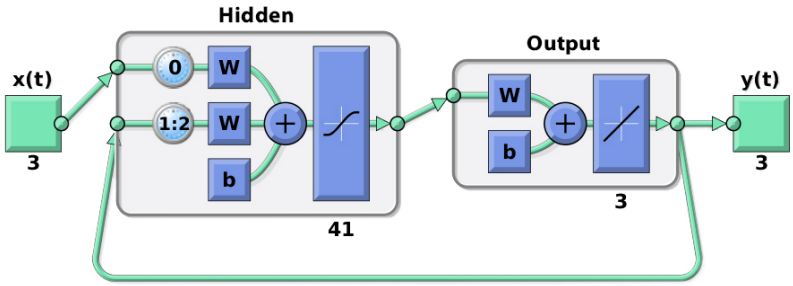

Figure 2.5: NARX Closed Loop.

2.4.2 LSTM

One of the most common shortcomings of RNNs is the problem of vanishing/ exploding gradients. This problem occurs in back-propagation through RNNs and is all the more critical as the network is deep. The continuous updates that the network undergoes due to the chaining rule during this process can lead to excessive vanishing or an exponential explosion of the gradient. Having too small gradient prevents the weights from being updated and learning, while on the contrary too high values lead the system to instability. Due to these issues, RNNs are unable to work with longer sequences and hold on to long-term dependencies, making them suffer from "short-term memory". For this reason we decided to use a type of network that solves, through a gating system, the problem of "short-term memory", that is the Long Short-Term Memory networks (LSTMs) [22], which in addition to using the most recent information can also keep memory on previous sequences.

The LSTM have a more complex structure in fact at each step, the LSTM cell receives 3 different information: the current input data, the short-term memory of the previous cell (similar to the hidden states in the RNN) and finally the long-term memory. Short-term memory is commonly referred to as 'hidden state' and long-term memory is generally known as 'cell state'.

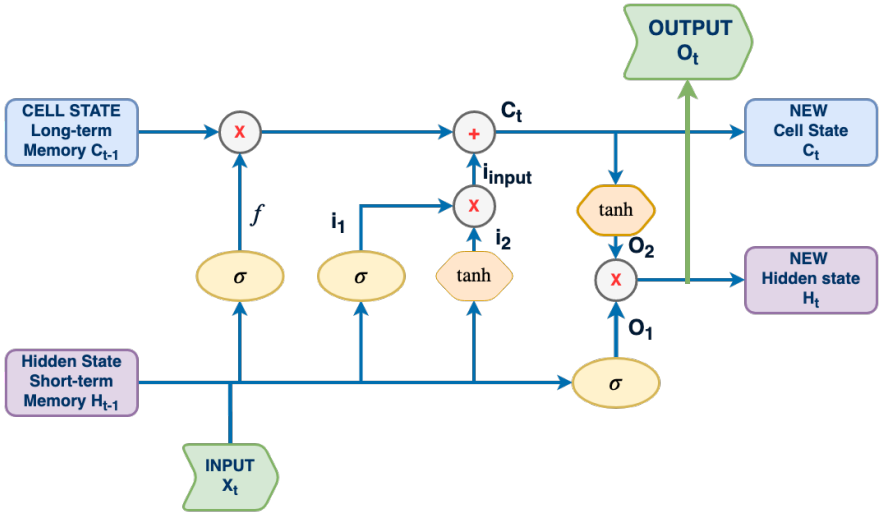

Figure 2.6: LSTM Architecture.

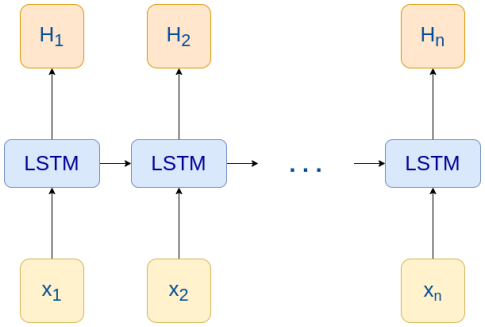

Figure 2.7: LSTM scheme.

The cell then uses gates to adjust the information to be kept or discarded at each step before passing the long and short term information to the next cell, the role of these gates should selectively remove any irrelevant information. Of course, these gates need to be trained to accurately filter what is useful and what is not. These gates are called the Input Gate, the Forget Gate, and the Output Gate.

Input gate The input gate is the one that filters the information to save it in the long-term memory and does this through 2 layers. The first level can be seen as the filter that takes short-term memory and current input and passes them into a sigmoid function that will transform values between 0 and 1, with 0 indicating that part of the information is useless, while 1 indicates that the information will be used. This allows us to distinguish useful values from those to be forgotten. As the layer is trained through backward propagation, the

weights in the sigmoid function will be updated so that it learn to let the profit pass only by discarding the less critical features.

$$i_1 = \sigma\left(W_{i_1} \cdot (H_{t-1}, x_t) + bias_{i_1}\right) \tag{2.7}$$

The second layer takes the short term memory and current input as well and passes it through an activation function, usually the tanh function, to regulate the network

$$i_2 = tanh(W_{i_2} \cdot (H_{t-1}, x_t) + bias_{i_2}) \tag{2.8}$$

The outputs from these 2 layers are then multiplied, and the final outcome represents the information to be kept in the long-term memory and used as the output.

$$i_{input} = i_1 * i_2 \tag{2.9}$$

Forget Gate This gate is fundamental for the selection of the information that must be kept in the long-term memory and which instead must be discarded. To do this we define from the short-term memory and the current input a Forget Vector which will multiply the information contained in the long-term memory.

Basically this vector will behave like a selective filter layer, and get the vector in a useful form we will have to put it into a sigmoid function (with weights different from the one previously used in the Input Gate). This vector will be composed of zeroes and ones in order to filter the information in the long-term memory.

$$f = \sigma\left(W_{\text{forget}} \cdot (H_{t-1}, x_t) + \text{ bias forget }\right) \tag{2.10}$$

Now it is necessary to sum the output of the Forget Gate with the output of the Input Gate in order to use the new version of the Long-term memory in the final Output Gate.

$$C_t = C_{t-1} * f + i_{\text{input}} \tag{2.11}$$

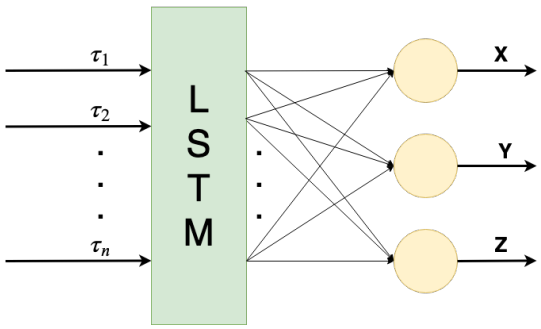

Figure 2.8: LSTM Linear Layer.

Output Gate This gate has the function of computing a new version of the short-term memory which will be passed on to the cell in the next time step. To do this we must put together the information generated by the previous gates.

To create the final filter we must pass the short-term of the previous state and the current input into a new sigmoid function, also it is necessary to pass the new long-term memory through a tanh activation function. Finally multiplying these two information we will get the final value of the short-term memory which will correspond to the new hidden state H_t.

$$O_1 = \sigma\left(W_{\text{output}_1} \cdot (H_{t-1}, x_t) + \text{bias}_{\text{output}_1}\right) \tag{2.12}$$

$$O_2 = \tanh\left(W_{\text{output}_2} \cdot C_t + \text{bias}_{\text{output}_2}\right) \tag{2.13}$$

$$H_t, O_t = O_1 * O_2 \tag{2.14}$$

Now this new information can be reported iteratively as initialization of the new cell and the process is repeated until the learning phase is complete finally. To return to Cartesian coordinates, a linear layer shown in the Figure 2.8 is introduced which allows to combine all the information in an output of three values.

2.5 Optimization

Once the robot has learned the dynamic model, it will be possible to perform a trajectory optimization to control movement in order to perform various tasks, such as reaching a point or throwing an object. Fixed the control horizon t_f which can vary depending on the task and the specifics of the problem, and fixed a control size step dt depending on the type of network, we want to be able to apply an optimization algorithm that taken a generic input vector allows the minimization of an objective function of a different nature. For example, in the case of reaching it could be the euclidean distance between the target point and the point reached with a series of inputs. To do this, we have chosen numerical methods such as SQP (Sequential Quadratic Programming) [23], a class of algorithms for solving non-linear optimization problems, that can vary a generic input vector in an iterative way until finding a combination capable of respecting the constraints of minimization (such as the physical limits of the inputs). The possibility of using SQP is guaranteed by the fact that the dynamic model is represented by an NN and with a smooth objective function and therefore always twice derivable.

The optimal policy can be estimated by minimizing the objective function given below:

$$\Pi^* = \arg\min_{\tau} \left[D_{task}(x_{\mathrm{NN_{out}}}, x^{des}) + ||\tau_{1:k_f-1} - \tau_{2:k_f}||_2 \cdot \beta \right] \tag{2.15}$$
$$\text{subject to } \tau^m_{\min} \leqslant \tau^m_k \leqslant \tau^m_{\max} \quad \forall m = 1 \dots M \text{ and } k = 1 \dots k_f$$

where $x_{\mathrm{NN_{out}}}$ is the cartesian output of the neural network, x^{des} indicates the target given as reference and D_{task} it is a characteristic function of each task that calculates the distance between these two quantities; $k_f = \frac{t_f}{\mathrm{d}t}$ indicates the final discrete instant of the prediction interval t_f; k indicates the position within the predicted trajectory; M corresponds to the number of actuators activated and $\tau^m_{\min}$ and $\tau^m_{\max}$ indicate the limits of the motor controls. The part on the right of the sum indicates a factor that control the motor effort: for safety reasons and to make the movement smoother, we decided to weigh, in addition to the final position, also the variation of the input commands by adding a regularization term on successive steps of the control variable vector.

For the reaching the control objective is suited to reach the point at the end of the control horizon while simultaneously optimizing the control effort. Other constraints or different factor could be keep in account to weight in a different way the position error and the effort of the motors or to add some constraints over the speed or other.

2.5.1 Reaching

For reaching task we want to find Control Variable values able to bring the arm, and in particular the Cartesian position of the end effector, in a precise position within the task space at the end of the control horizon. Obviously, we have redundancy of solutions, so optimization will return an optimal local solution. Thus, the equation of the cost function has been defined as follows:

$$D_{reach}(x_{\mathrm{NN_{out}}}, x^{des}) = \left\| x^{\mathrm{NN_{out}}}_{\substack{t_f \\ \overline{dt}}} - x^{des} \right\|^2 \tag{2.16}$$

where the term on the left of the sum describes the distance between the target and the end position of the end effector, i.e. the last position predicted by the RNN and the term on the right is the norm of the vector that describes the difference between a value and its successive of the input vector and this allows to regulate the effort of the control. Once the control sequences have been obtained, both using the NARX and LSTM, these were used on the the robot.

2.5.2 Throwing

For the Throwing task several problems arose. In fact, having set a target outside the task space that could be a point or circular basket, some questions arise such as: "what is the trajectory to be followed?" , "What is the best launch point to leave the object?", "How fast should you leave the object?" . To simplify the problem we decided to focus on a rigid and uniform object, such as a plastic ball. Certainly in the future it would be interesting to integrate this method with a model for recognizing the object and optimizing the trajectory of non-uniform objects. We therefore decided to leave optimization to the arduous task of answering all these questions, formulating an optimization function that contained all the highlighted problems.

Once the generic equation of projectile motion has been written:

$$\begin{cases} x(t) = x_0 + v_{0x}t \\ y(t) = -\frac{1}{2}gt^2 + v_{0y}t + y_0 \end{cases} \tag{2.17}$$

we can adapt it for our problem. For every control instant k, we can compute a different parabola $P_k = [Y_k, Z_k]$:

$$\begin{cases} Z_k(T) = z(k) + \frac{z(k)-z(k-1)}{dt}T \\ Y_k(T) = y(k) + \frac{y(k)-y(k-1)}{dt}T - \frac{1}{2}gT^2 \end{cases} \tag{2.18}$$

where $p = [y, z] = X_{NNout}$ is the RNN output prediction that contains the information of the cartesian position relative to the current input evaluated by the optimization algorithm and T is a vector of time instant of dt duration:

$$T = 0 : dt : T_{throw} \tag{2.19}$$

The optimization function, for each set of inputs it is evaluating, will calculate a velocity vector V by dividing by dt the difference between one instant $p(k)$ and the previous one $p(k-1)$ of the network output associated with that input. We will therefore have a vector P of $n = \frac{T_{trow}}{dt}$ positions, associated with n control variable values, and a vector V of n velocities,that describes the velocity necessary to reach $p(k)$ from $p(k-1)$ considering the first value $p(0) = 0$.

$$V = \left[\frac{p(1) - p(0)}{dt}, \frac{p(2) - p(1)}{dt}, ..., \frac{p(k) - p(k-1)}{dt}, \right. \\ \left. ..., \frac{p(T_{throwmax}) - p(T_{throwmax} - 1)}{dt}\right] \tag{2.20}$$

Of these n parabolas built we will save the one that has minimum distance with the basket and its point of release and launch velocity. It is evident that if we find a parabola that passes through the center of the target, it will have zero distance from it, resulting optimal. The distance between a parabola and the target can be computed as follows:

$$D_k(X_{NNout}, x^{des}) = \min_T \left|\left|P_k(T) - x^{des}\right|\right|^2 \tag{2.21}$$

Finally, the closest parabola will also give the minimal distance that can be obtained with the current motor commands that are being evaluated:

$$D_{throw}(X_{NNout}, x^{des}) = \min_k D_k(X_{NNout}, x^{des}) \tag{2.22}$$

For each iteration cycle of the optimization function we will therefore have an optimal parabola (the red one in the Figure 2.9), and finally the algorithm will return the control variable vector to obtain this parabola together with the related information necessary to perform the launch, in particular the release point and launch velocity. Now, the work continues by giving the robot this input vector and ensuring the release at the point indicated by the algorithm we should have a launch that hits the target.

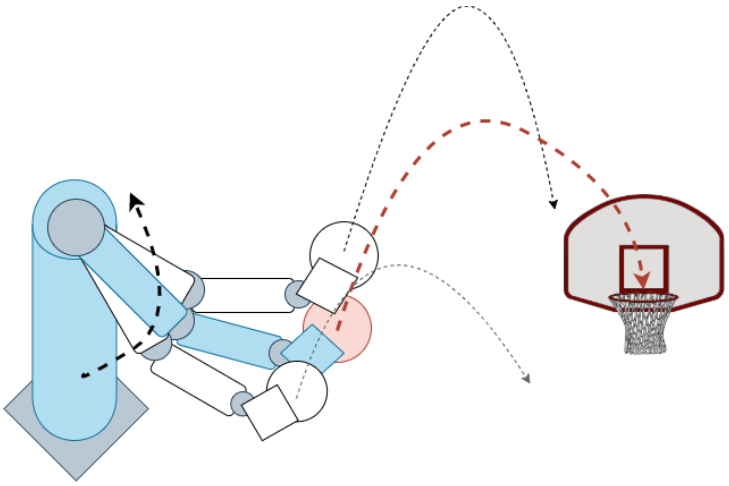

Figure 2.9: Simulation of a throwing.

Chapter 3

Robotic Implementation

Our project is mainly designed to be used with very complex robots whose dynamic functions are not easily identifiable. We have used an iCub whose arm, characterized by seven degrees of freedom, presents a non-trivial dynamic and it is possible to implement different types of control on different motors simultaneously.

3.1 iCub

iCub is a child-size humanoid robot capable of crawling, grasping objects, and interacting with people [10]. It's designed as an open source platform for research in robotics, AI, and cognitive science. There are around 20 icubs in the world, almost all of them in Europe, and one in America. Cub stands for "cognitive universal body".

> The RobotCub Consortium, funded in part by the European Commission's Cognitive Systems and Robotics program, started developing the humanoid iCub in 2004. The first version was released in 2008. New versions followed, upgrading the robot's head mechanics, upper-body skin, and sensing. Future versions will focus on bipedal locomotion. Other institutions participating in the project include University of Genoa, Scuola Sant'Anna, University of Ferrara, Telerobot, University of Uppsala, University of Sheffield, University of Hertfordshire, IST, EPFL, and University of Zurich [From `robots.ieee.org/robots/icub/`]

Figure 3.1: iCub,IIT.

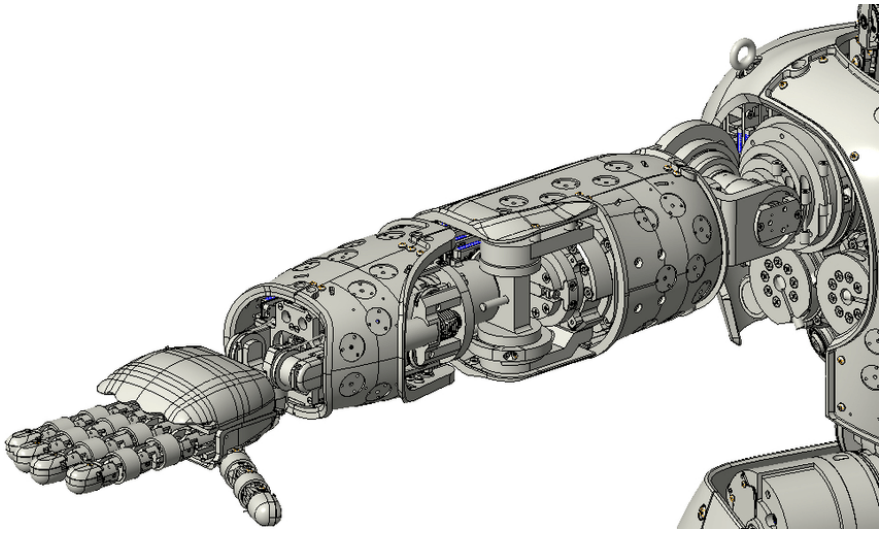

Figure 3.2: Robotic arm

3.2 Robotic arm structure

To deal with the problem we decided to use the left arm of the iCub robot . This arm is driven by 7 motors that corresponds to the degrees of freedom of the arm: 3 for the shoulder, 1 for the elbow and 3 for the wrist, with the possibility of also introducing the control on the fingers of the hand, which do not in any way affect the position of the end effector which is placed in the center of the palm [24].

Figure 3.3: iCub Simulator,IIT.

3.3 iCub Simulators

Two different simulation environments are available for this robot: iCub 3D simulator (IIT) [25], a platform built specifically for iCub, and a simulated version of the robot in Gazebo, a platform very popular in the field of simulation. To use the motor torque control the choice to use Gazebo [26] is mandatory because only in the latter was there an available implementation of this kind of controller, this was not yet available in the simulator provided by the IIT.

3.4 Interface with iCub - YARP

"YARP stands for Yet Another Robot Platform. What is it? If data is the bloodstream of your robot, then YARP is the circulatory system"

Yarp [27], [28] is not an operating system but it is a tool to interface with the robot, in fact it is designed to be integrated with different operating systems and, although it is written in C++, there are several tools to be used with other programming languages, such as Python. The philosophy that prompted the creation of this protocol is to be usable in different contexts in order not to make iCub a closed environment and to increase the longevity of the robot software projects regardless of the operating system and hardware. YARP provides a detailed language to use correctly the robot, and moreover provides several

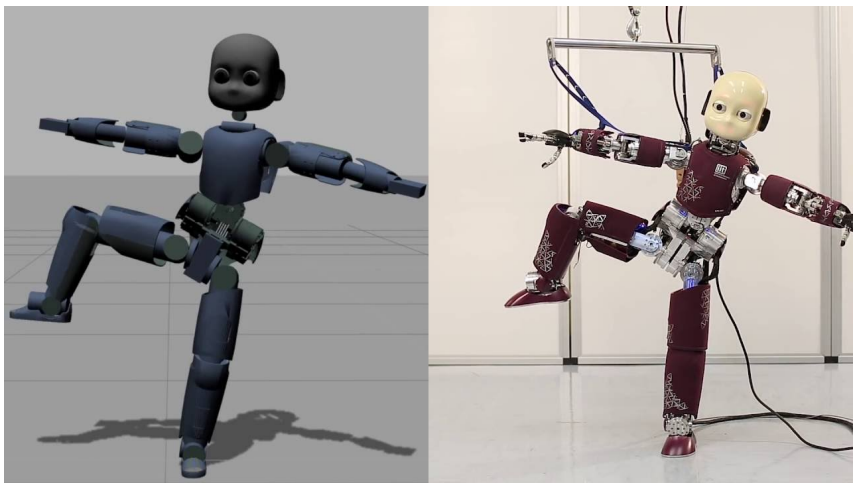

Figure 3.4: Comparison between Gazebo and repeatability on real robot

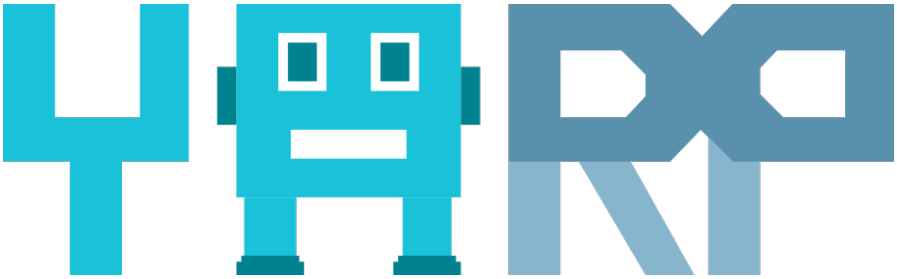

Figure 3.5: YARP Logo

tools, one of that is 'yarp motorgui' which allowed us to monitor the behavior of the motors in real time to actually see what was going on.

As can be seen from the image it is possible to control the robot actuators in a different way, and for our purpose we will use different control strategies depending on the task, but as we will show later on a methodological level it will not change to control the joint in force rather than in velocity although on a control level it does not have the same meaning.

The most functional feature of YARP is that by changing few lines of code it allows us to use the same script for both simulation and real robot, simplifying and speeding up the work.

3.5 Motor Control Mode

The model of iCub(v1.x) that we use is equipped with four 6-axis Force/Torque sensors mounted on the arms, this devices measure the force and the torque from all the three Cartesian coordinates. iCub(v1.x) thus exploits a model-based

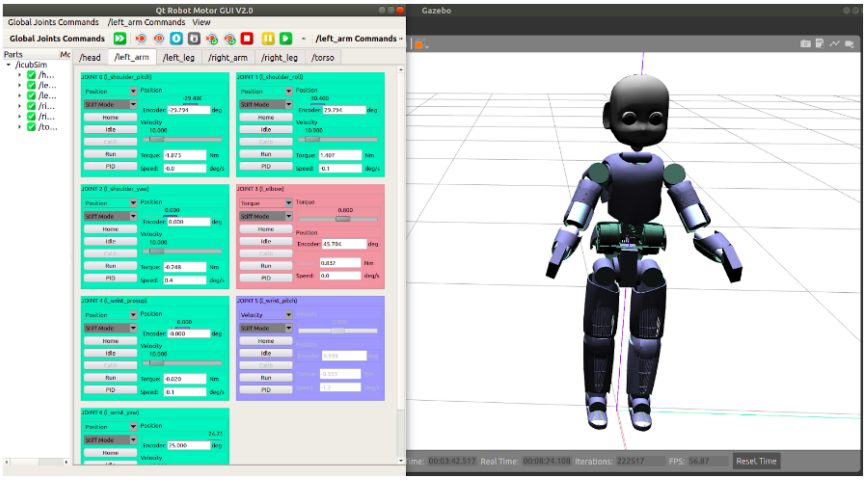

Figure 3.6: yarpmotorgui

approach based on a modified Newton-Euler algorithm in order to estimate joint-level torques from the four proximal sensors. The controller is thus distributed in three different levels[1]:

- **wholeBodyDynamics (application level):** the modules takes the measurements from the four F/T sensors of the robot limbs to make a model-based estimation of joint torques, with the hypothesis that external forces are applied only on the end-effector.
- **iCubInterface (middleware):** it sends (through yarp ports) the 6-axis F/T sensors measurements to the wholeBodyDynamics module and receives from it the computed joint torques.
- **motor control boards (firmware level):** The control boards receive the computed estimation of the joint torques from iCubInterface and implement different PID control algorithms in order to track the desired position/torque commands.

We focus on three different control modes that are currently implemented in the firmware of the control boards:

- **Position control mode:** In this control mode, the motors pulse-width modulation(PWM) is computed using a PID controller the receives in input the desired joint position and the current measurement from the joint encoders:

$$PWM = PID\,(q - q_d) + PWM_{offset} \tag{3.1}$$

[1]From the iCub online manual `http://wiki.icub.org/wiki/Force_Control`

Note that when you command a new joint position, you are not instantaneously assigning the reference q_d in the above formula. Instead, a mimum jerk trajectory generator takes in input your commanded position and the desired velocity, and produces a smooth movement creating a sequence of position references q_d tracked by the PID controller.

- **Velocity control mode:** Velocity control mode allows you to control the robot by assigning a desired volocity/acceleration to a joint. The control law is the same of position control, but in this case qd is not directly controlled by the user, but it is obtained from the integration of the commanded user velocity.

- **Torque control mode:** Torque control mode allows you to directly control the robot joints torque:

$$PWM = PID\,(\tau - \tau_d) + PWM_{offset} \tag{3.2}$$

In this case the motors PWM is computed using a PID controller the receives in input the desired joint torque and the current measured joint torque. Additionally, a PWM offset can be added to the output of the control algorithm. If both the commanded reference torque and the PWM offset is set to zero, the robot joint will be free to be moved in the space (eventually it will move down as an effect of the gravity acting on that joint).

The implementation of the velocity control shown in the Figure 3.7(i.e. integration of the reference command) corresponds to the current implementation on the iCub 4DC/BLL control boards. However,this is not the only possible implementation: an explicit velocity control loop is also possible. Torque control must implement an additional low level check to prevent the movement of the joint against the hardware limit. In all control modes, an additional feed-forward input, φ_{off} is available.

Our initial idea was to directly control the motors through the generation of motor stimuli, or torque commands but for iCub the wrist joints are not controllable in torque by the manufacturer's choice, but only in velocity and position, so we opted for a velocity control [29]. Moreover, we found problems in the torque control of the elbow joint, probably due to an error in the torque estimation, we therefore decided to change the control mode of this joint also in velocity mode.

Obviously this choice remains dictated by the robot used, in a different case it would have been interesting to control all the joints through direct commands to the motors. From the point of view of learning the model, however, if a

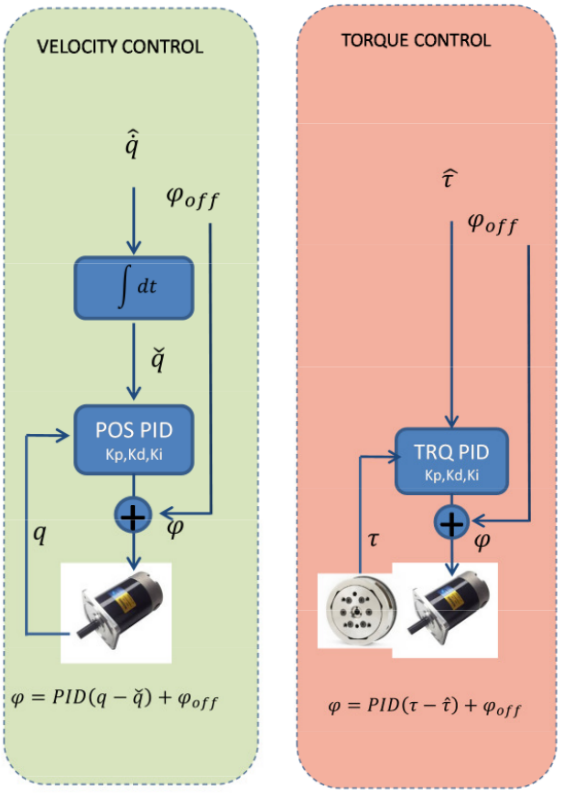

Figure 3.7: Motor Controllers

correct normalization of data is applied, it remains guaranteed that the neural network is able to learn a mapping between the control commands and the final position of the end effector.

Finally, for safety reasons it was decided to implement a safety zone for robot movement which necessarily bounded the task space in order to avoid dangerous situations for the robot and the surrounding environment. To do this, a policy for the use of all the joints has been defined, such that the robot is not asked to work around the joint limit. In addition, a limit zone has been defined on the y axis, i.e. the axis that passes vertically with respect to the robot, in order to avoid a collision with the workbench. this area limits the movement of the robot in the vertical coordinate in the range of values: $[0.47, 0.8]$ meters.

3.6 Software and IT tools

Various IT tools and different programming languages have been used for this project. The part of the work that required the use of the robot or simulators was carried out by writing in C ++, able to integrate directly with YARP, and

Python, able to interface with YARP only thanks to C ++ bindings.

The machine learning part was developed in Python for the LSTM and in Matlab2019b for the NARX part. Let's briefly illustrate some of the most useful libraries for our work.

- PyTorch(Python)

  PyTorch is an open source machine learning library used for developing and training neural network based deep learning models. It is primarily developed by Facebook's AI research group. PyTorch uses dynamic computation, which allows greater flexibility in building complex architectures. Pytorch uses core Python concepts like classes, structures and conditional loops — that are a lot familiar, hence a lot more intuitive to understand. This makes it a lot simpler than other frameworks like TensorFlow that bring in their own programming style. PyTorch provides two high-level features:

  - Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPU)

  - Deep neural networks built on a tape-based autodiff system

- Deep Learning Toolbox^{TM}(Matlab2019b)

  Deep Learning Toolbox^{TM} (formerly Neural Network Toolbox^{TM}) provides a framework for designing and implementing deep neural networks with algorithms, pretrained models, and apps.

- SciPy (Python)

  SciPy is an Open Source Python-based library, which is used in mathematics, scientific computing, Engineering, and technical computing. SciPy contains varieties of sub packages which help to solve the most common issue related to Scientific Computation.

- Optimization Toolbox^{TM}(Matlab2019b)

  Optimization Toolbox^{TM} provides functions for finding parameters that minimize or maximize objectives while satisfying constraints. The toolbox includes solvers for linear programming (LP), mixed-integer linear programming (MILP), quadratic programming (QP), nonlinear programming (NLP), constrained linear least squares, nonlinear least squares, and nonlinear equations. You can define your optimization problem with functions and matrices or by specifying variable expressions that reflect the underlying mathematics.

Chapter 4

Results

The results chapter will be presented in this way: first we will show how the motor babbling data set was created,then we will show the results of the training phases of the neural networks, then we will show how the optimization was performed with the results of each network for the reaching task. Finally we will show some considerations that we made in view of future throwing experiments.

4.1 Motor Babbling

For motor babbling we decided to build sinusoidal profiles to avoid a too sudden variation of the reference signals, that would have led to a saturation of the motors in some cases. In fact the motor is not able to follow a reference that changes of a large amplitude at high frequency.

Due to this problem and the danger of working in an unsafe area, we decided to create the data set as the sequential union of several tests all of the same duration , that is 5 seconds, and each test restart from the robot's setup position. This time was considered appropriate to be able to collect enough data relating to each sinusoidal signal.

To ensure the randomness of the signal, we decided to compose the data set with sinusoids of different profiles. For this we have randomly selected both the frequency of the sinusoid and the amplitude of the sinusoid. After a certain number of tests the signal changes and becomes a sum of sinusoids, and so on until the signal is a sum of five random sinusoids. For obvious reasons the maximum amplitude of the sum of the sinusoids has been limited so as never to have a signal with an excessive amplitude. For each motor a different reference signal has been created.

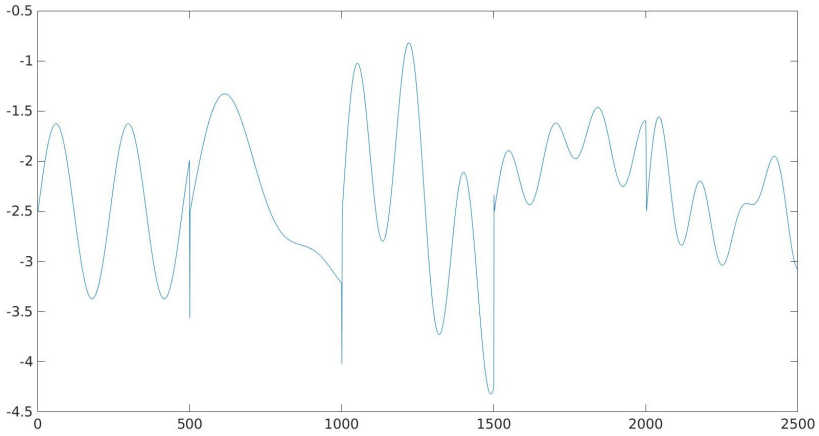

Figure 4.1: Example of Babbling references.

```
nsin=1 to 5
for (int s = 1; s < nsin; s++) {
 ampl = randomgen();
 freqs = randomgen();

 for ( int i = 0; i < refs.size(); i++) {
  sigs[i] = sigs[i] +
            sin(ampl[i]/nsin, freqs[i]) * amplmax[i];
        }
}
```

Below we show an example of a profile of 5 sinusoidal sequences (Figure 4.1), where the first represents a single sinusoid and the last one a sum of 5 sinusoids, joined sequentially to form the data set. It can be seen that at every 500 instant of time ($0.01s$) we have a jump in the figure that represents that the robot has returned to the setup position.

The final form of our reference signals is composed of sequences lasting 5 seconds. We have set a number of 500 tests but some have reached the limits of the safety zone and were interrupted and therefore discarded, so at the end of the 500 tests we collected 473 sequences useful for training neural networks.

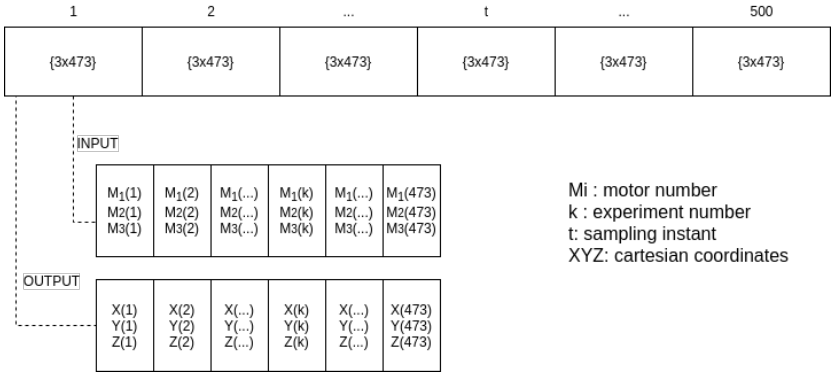

Figure 4.2: Shape of input and output vectors.

4.2 Model Selection NARX

We need to specify that some precautions have been made that have brought a substantial modification to the NARX, as shown in the Figure 4.2 . In fact, in order to use the same data set that we will then use for the LSTM(i.e. short sequences), we took each of the sequences created during babbling and instead of putting them in a long sequence, we created a sequence of 500 cells (this number represents the instants sampled every $0.01s$ for the duration of each $5s$ babbling experiment). In each cell we put sampling at the relative time instant of all the experiments made (which in this case are 473).

This choice was deemed necessary both to be able to reuse the same LSTM data set and to have a more realistic comparison between the two networks, but also because given the limits imposed by the safety zone we would have had discontinuities in the data set that would have introduced a further error in the approximation phase.

For the training of this network, a very deep model selection was not made. This choice was dictated by several factors, the first was that our goal was not to find the best existing network, but for our approach we were convinced that a network that approximated the dynamic model in a realistic way for our task was enough if it wasn't the optimal one.

A further factor of this choice derives from the fact that for the use we make of it, this network allows to change a few features, the most relevant are certainly the size of the single layer and the values to be kept in the TDL(Tapped Delay).

Referring to the previous work of Thuruthel et al. [1], we made several attempts until we found a configuration that gave us results that we considered acceptable. Below is an image of the training performance in OpenLoop configuration, to be sure that it was not a lucky case, the same configuration has been tried several times, giving almost identical results.

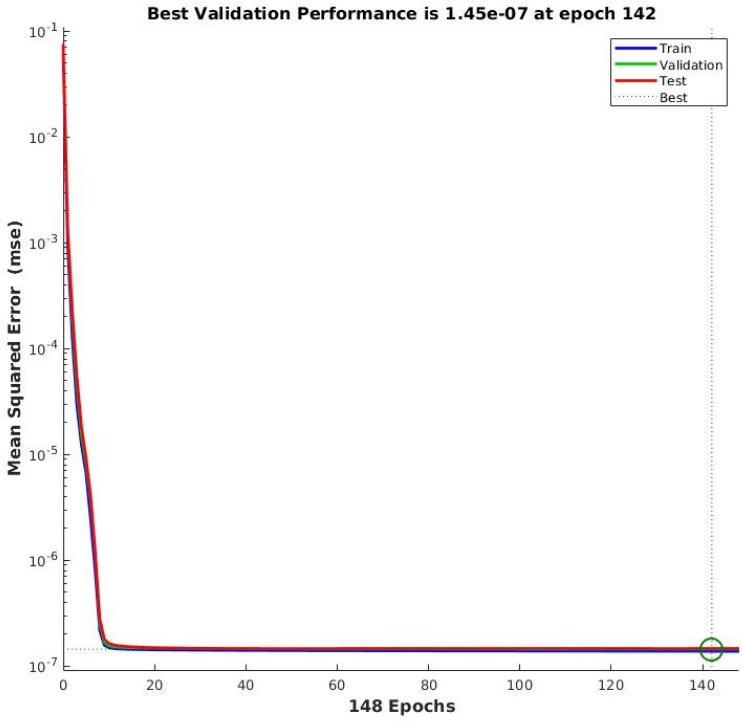

Figure 4.3: NARX OpenLoop performance.

For this type of network the training is done in two steps, firstly the network was trained in the open loop configuration (the choice to train the network directly in closed loop is not indicated because the training could be very sensitive to the problem of the explosion of the gradient).

The Figure 4.3 of the performance of the open loop training shows how the error tend to decrease both in the training and in the test and validation phases, this behavior means that the network has learned well from the data it received as input.

Once the training in open configuration is completed, the network can be easily closed and another training phase can be carried out, this is necessary because the open loop training could be subject to overfitting. For training we tried to use two different backpropagation algorithms, Levenberg–Marquardt (lm) and Bayesian Regularization (br), noting that the results of the second are slightly better but with significantly longer training times. So we opted to use the lm.

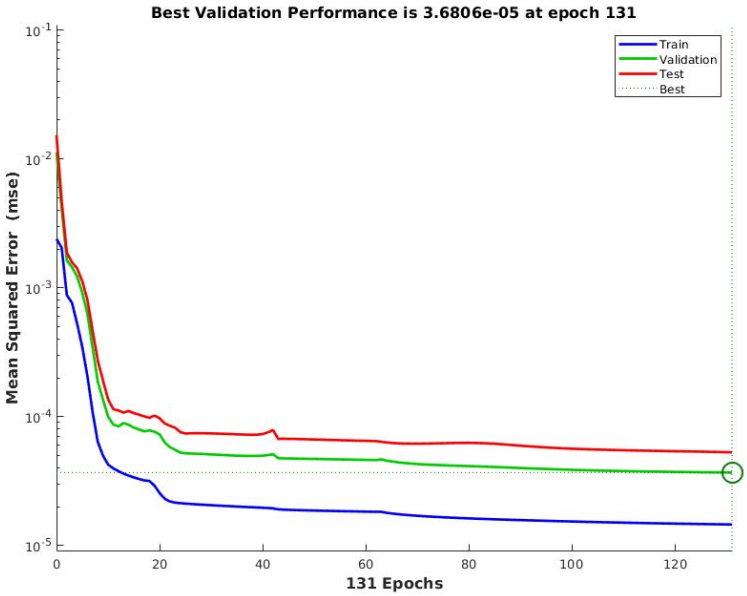

Figure 4.4: NARX ClosedLoop performance.

The perfomance function for the open training phase is calculated as:

$$\text{MSE} = \frac{1}{T}\sum_{t=0}^{T} \|X_t - f\left(X_{t-1}, U_t\right)\|^2 \tag{4.1}$$

where X is the input vector and U is the exogenous input vector. The function f represents the mapping formed by the neural network

Once the training in open configuration is finished, the network can be easily closed and another training phase can be carried out, this second training phase is necessary because the open loop training could be subject to overfitting.

The performance function for the closed training phase is the only thing that changes, in fact the size of the network remains constant, and is calculated as:

$$\text{MSE} = \frac{1}{T}\sum_{t=0}^{T} \left\|X_t - f\left(\widehat{X}_{t-1}, U_t\right)\right\|^2 \tag{4.2}$$

Here, $\widehat{X}_{t-1}$ is the prediction of the NARX network in the previous iteration. Now the learning algorithm is not trying to reduce the single-step error but the

whole multi-step prediction error (the performance function is the only difference between the open loop network and the closed loop network).

4.3 Model Selection LSTM

For this network instead we preferred to use Python and in particular the use of the PyTorch package, already described in the implementation part. This choice was dictated by the possibility of having a random search algorithm already implemented by Nardo [30] available. With this algorithm we were able to do a fairly deep search and this was indispensable in the LSTM having it a much greater number of parameters to tune. It is good to say that even in this case we have not found the optimal solution to solve our problem, as the LSTM model selection requires many days of computation, for this reason we opted for an accurate but not refined selection.

```
rnn_ml_adam.search_holdout((models.LSTM, inp_Dim, out_dim)

{'epochs': 1000, 'timesteps': [8, 16, 32, np.nan],
  'lr': [-4.0, -2.0, 0.15],'hidden': [30, 90, 0.15, 10],
  'beta1': [0.7, 1.0, 0.15],
  'beta2': [0.9, 0.98, 0.15],
  'epsilon': [-8, -8, 0.15],
  'wd': [-8, -5, 0.15], 'batch': [1],
  'grad_clip': [0]},
{'trials': [20], 'repeat': 3,
  'patience': [20],
  'es_tr': (patience_tr, 0, min_delta_tr),
  'es_vl': (patience_vl, 0, 0, avg_epoch),
  'fcn': (loss_fcn, obj_fcn, 'min'),
  'seed': seed,
  'retain_hidden': False,
  'metrics': metrix
}, device, 'task', load=True, verbose=True)
```

For the model selection, we divided the data set into 70:30 (training: test) and concentrated on finding some hyperparameters such as the learning rate $lr \in [-4, -2]$; the number of hidden state $hidden \in [30, 90]$; weight decay $wd \in [-8, -5]$ is the L2 regularization and its role is to avoid the overfitting. There are other parameters such as β_1 and β_2 which are not trivial to understand, but are necessary for the Adaptive moment estimation (Adam) to control the first and second moment of the gradient. In this treatise we are not interested

in going deeper into the explanation of these parameters, but we can refer to (Adam; Kingma and Ba, 2014) [31]

In addition, *trials* indicates the number of hyperparameter configurations performed; *patience* indicates the number of epochs before stopping training in the event of overfitting and has the function of guaranteeing early stopping; *repeat* indicates the number of times we have retried training with these hyperparameters in order to evaluate if the result was repeatable. Obviously by setting a number of attempts and greater repetitions we could have found better results.

Result of the 3 repetitions

- Training loss(TR): 6.301e-05 Test loss(TS): 2.883e-05 epochs: 105
- Training loss(TR): 6.709e-05 Test loss(TS): 5.387e-05 epochs: 98
- Training loss(TR): 6.329e-05 Test loss(TS): 5.049e-05 epochs: 100

After model selection we take the average values of TR and TS to choose the best model:

- TR: 6.446e-05(-+2.281e-06) — TS: 4.44e-05(-+1.359e-05)
- *hidden*: 41
- *lr*: 0.004428
- β_1: 0.8262
- β_2: 0.9662
- ϵ: 1e-08
- *wd*: 2.746e-08

The loss Figure 4.5 show how the design error, i.e. the loss of the training phase, decreases and this means that the network has learned information from the data it has taken as input. The profile of the loss of the test phase instead is not linear and has discontinuities, this tells us that the model approximation is probably not accurate enough for each point of the task space.

4.4 Trajectory Optimization

Once a learned model is obtained, we will use it as if it were actually a function that taking inputs univocally associates outputs. From this moment on, therefore, the model will no longer change its characteristics and will no longer face

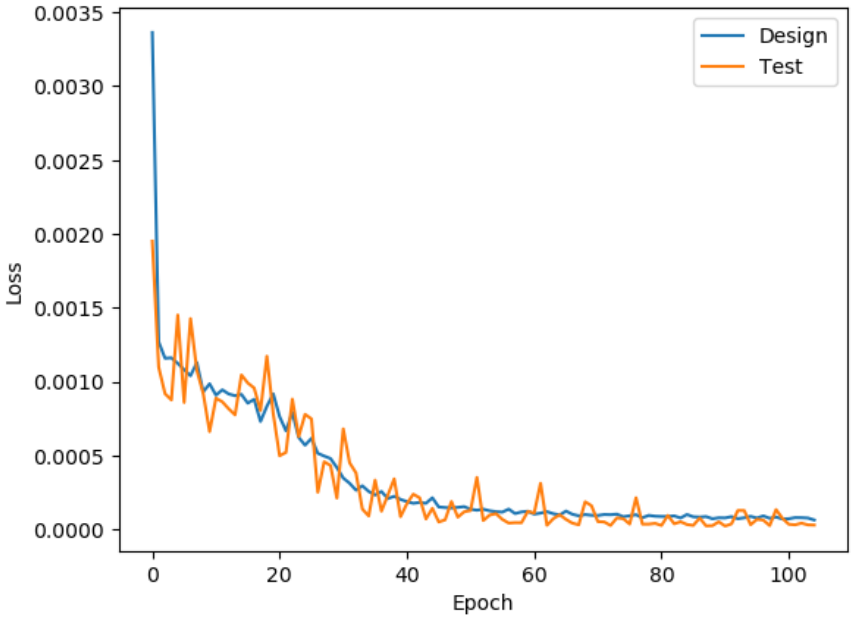

Figure 4.5: LSTM loss.

any learning phase. In this phase, we decided to try different tasks: reaching and throwing. The final purpose of this phase is to obtain a control variable vector that, once given as input to the robot, can control it to perform tasks. To do this, various optimization algorithms have been used, which we have already described in Section 2.5. In practice, the optimization function (`fmincon` in Matlab and `scipy.optimize.minimize` in Python) takes a vector of pairs/sets of references of the control variable, for a number of steps equal to $\frac{t_f}{\mathrm{d}t}$ and at each iteration varies depending on the optimization algorithm (SQL for example) these references and evaluates the cost function as this input changes, and finally returns the value related to the minimum error.

4.5 Tollerance and Repeatability

Before continuing our work it is good to evaluate which tolerance we want to set in order to decide whether the result is satisfactory. First of all it would be advisable to carry out several experiments giving the robot the same input in order to be able to evaluate what its actual repeatability is. With the data we had, it was possible to evaluate the average error in terms of Euclidean distance and its variance. The results tell us that the average error for the Euclidean distance is approx

$$\text{Mean Error (m)} = 0.0064$$

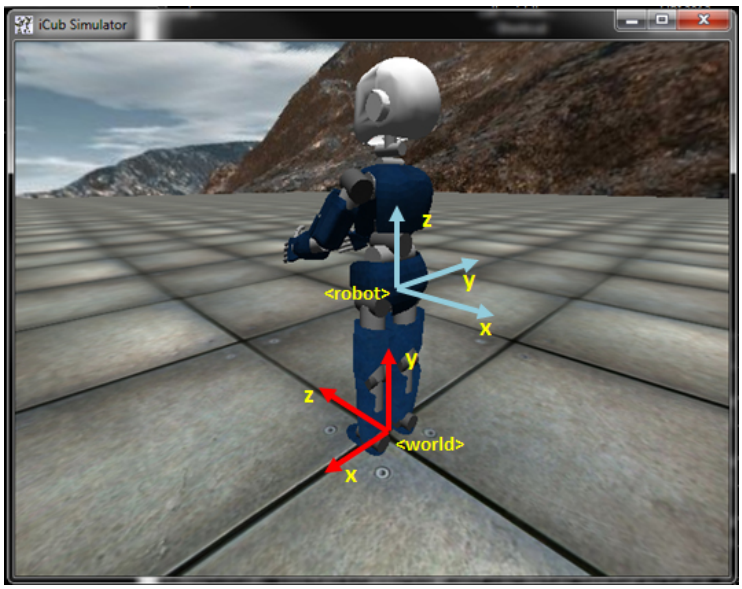

Figure 4.6: iCub Cartesian reference

and its variation:

$$\text{Standard Deviation (m)} = 0.0024$$

For this reason we decided to select a tolerance T = 0.009m

4.6 Reaching Results

For our reaching experiment, we chose 10 points, 9 of which were random within the task space and the Setup point, and three different time horizon t_f (500ms, 1sec, 2sec). Motor trajectories to reach these points have been generated using the optimization procedure, as described in Section 2.5.1, with a value of β equals to for 0.005 for the NARX and 0.001 for the LSTM.

From the Figure 4.6 it is possible to see the references of the robot, we will always refer to the references of the world. Table 4.1 shows the selected target points and the relative distance from the setup point. This distance is shown both in terms of all three Cartesian coordinates X, Y, Z, but it is to be noted that we do not have any actuator capable of acting directly on the coordinate X, in fact the shoulder pitch, the wrist pitch and pitch elbow allow to directly change only the position on the Y-Z plane, the movement on the X axis is in fact due to the usual mechanical limitations to which each rigid robot is subjected.

The other tables show the final position of the end effector at the end of each experiment. The first value in the table indicates the associated experiment and therefore a change of the reference target. The second row recall the target

$Targets$	X	Y	Z	$AmpMoveXYZ$	$AmpMoveYZ$
SETUP	0.1141	0.5582	0.2639	0	0
02	0.1172	0.5698	0.2724	0.0147	0.0143
03	0.1155	0.5792	0.2783	0.0255	0.0254
04	0.1212	0.5846	0.2806	0.0320	0.0312
05	0.1024	0.6130	0.2991	0.0661	0.0651
06	0.1300	0.6322	0.2893	0.0798	0.0782
07	0.0972	0.6572	0.3188	0.1144	0.1132
08	0.1080	0.6752	0.3041	0.1238	0.1237
09	0.1110	0.6919	0.3090	0.1411	0.1411
10	0.1284	0.7003	0.2972	0.1466	0.1459
	[m]	[m]	[m]	[m]	[m]

Table 4.1: Cartesian Reference

$SETUP$	STEP	X	Y	Z	$dXYZ$	dYZ
TARGET		0.1141	0.5582	0.2639	0	0
NARX	50	0.1183	0.5562	0.2619	0.0051	0.0028
LSTM	50	0.1167	0.5477	0.2557	0.0135	0.0132
NARX	100	0.1165	0.5530	0.2597	0.0056	0.0054
LSTM	100	0.1153	0.5426	0.2515	0.0198	0.0198
NARX	200	0.1167	0.5579	0.2633	0.0027	0.0006
LSTM	200	0.1151	0.5445	0.2528	0.0176	0.0176
	[0.01 s]	[m]	[m]	[m]	[m]	[m]

Table 4.2: Table setup point

of the experiment and the amplitude of the movement. The following rows indicate the network used by the optimization algorithm and the associated time horizon t_f, where the value in the column 'STEP' indicates the number of control instants lasting $0.01s$. The Cartesian coordinates associated with that experiment are indicated below the X Y Z columns, while $dXYZ$ indicates the Euclidean distance considering all three coordinates, while dYZ consider only Y and Z coordinates.

02	STEP	X	Y	Z	$dXYZ$	dYZ
TARGET		0.1172	0.5698	0.2724	0.0147	0.0143
NARX	50	0.1191	0.5711	0.2718	0.0024	0.0014
LSTM	50	0.1177	0.5571	0.2626	0.0159	0.0159
NARX	100	0.1178	0.5732	0.2730	0.0035	0.0035
LSTM	100	0.1175	0.5555	0.2615	0.0179	0.0179
NARX	200	0.1196	0.5685	0.2693	0.0040	0.0033
LSTM	200	0.1167	0.5481	0.2556	0.0273	0.0273
	[0.01 s]	[m]	[m]	[m]	[m]	[m]

Table 4.3: Table point 02

03	STEP	X	Y	Z	$dXYZ$	dYZ
TARGET		0.1155	0.5792	0.2783	0.0255	0.0254
NARX	50	0.1188	0.5779	0.2758	0.0042	0.0027
LSTM	50	0.1175	0.5639	0.2677	0.0186	0.0185
NARX	100	0.1195	0.5759	0.2742	0.0065	0.0052
LSTM	100	0.1177	0.5589	0.2640	0.0249	0.0248
NARX	200	0.1207	0.5723	0.2715	0.0109	0.0096
LSTM	200	0.1170	0.5547	0.2609	0.0301	0.0301
	[0.01 s]	[m]	[m]	[m]	[m]	[m]

Table 4.4: Table point 03

04	STEP	X	Y	Z	$dXYZ$	dYZ
TARGET		0.1212	0.5846	0.2806	0.0320	0.0312
NARX	50	0.1233	0.5843	0.2783	0.0031	0.0023
LSTM	50	0.1194	0.5710	0.2713	0.0164	0.0163
NARX	100	0.1208	0.5898	0.2815	0.0052	0.0052
LSTM	100	0.1186	0.5787	0.2765	0.0075	0.0071
NARX	200	0.1228	0.5849	0.2776	0.0034	0.0030
LSTM	200	0.1196	0.5702	0.2708	0.0174	0.0174
	[0.01 s]	[m]	[m]	[m]	[m]	[m]

Table 4.5: Table point 04

05	STEP	X	Y	Z	$dXYZ$	dYZ
TARGET		0.1024	0.6130	0.2991	0.0661	0.0651
NARX	50	0.1077	0.6149	0.2988	0.0056	0.0018
LSTM	50	0.1071	0.6206	0.3011	0.0091	0.0078
NARX	100	0.1089	0.6179	0.3007	0.0084	0.0050
LSTM	100	0.1094	0.6179	0.3003	0.0086	0.0050
NARX	200	0.1074	0.6200	0.3021	0.0090	0.0076
LSTM	200	0.1030	0.6000	0.2930	0.0144	0.0144
	[0.01 s]	[m]	[m]	[m]	[m]	[m]

Table 4.6: Table point 05

06	STEP	X	Y	Z	$dXYZ$	dYZ
TARGET		0.1300	0.6322	0.2893	0.0798	0.0782
NARX	50	0.1300	0.6202	0.2887	0.0116	0.0116
LSTM	50	0.1244	0.6243	0.2933	0.0104	0.0088
NARX	100	0.1303	0.6348	0.2921	0.0041	0.0040
LSTM	100	0.1275	0.6303	0.2937	0.0053	0.0047
NARX	200	0.1329	0.6446	0.2940	0.0140	0.0135
LSTM	200	0.1297	0.6241	0.2903	0.0081	0.0081
	[0.01 s]	[m]	[m]	[m]	[m]	[m]

Table 4.7: Table point 06

07	STEP	X	Y	Z	$dXYZ$	dYZ
TARGET		0.0972	0.6572	0.3188	0.1144	0.1132
NARX	50	0.1073	0.6536	0.3123	0.0060	0.0046
LSTM	50	0.1081	0.6556	0.3126	0.0126	0.0064
NARX	100	0.0990	0.6453	0.3135	0.0132	0.0130
LSTM	100	0.1095	0.6623	0.3155	0.0137	0.0060
NARX	200	0.0965	0.6346	0.3106	0.0241	0.0241
LSTM	200	0.1091	0.6456	0.3115	0.0182	0.0137
	[0.01 s]	[m]	[m]	[m]	[m]	[m]

Table 4.8: Table point 07

08	STEP	X	Y	Z	$dXYZ$	dYZ
TARGET		0.1080	0.6752	0.3041	0.1238	0.1237
NARX	50	0.1139	0.6700	0.3107	0.0048	0.0044
LSTM	50	0.1138	0.6794	0.3125	0.0110	0.0094
NARX	100	0.1197	0.6824	0.3116	0.0123	0.0087
LSTM	100	0.1219	0.6578	0.3036	0.0223	0.0174
NARX	200	0.1246	0.6428	0.2999	0.0353	0.0327
LSTM	200	0.1259	0.6351	0.2958	0.0447	0.0409
	[0.01 s]	[m]	[m]	[m]	[m]	[m]

Table 4.9: Table point 08

09	STEP	X	Y	Z	$dXYZ$	dYZ
TARGET		0.1110	0.6919	0.3090	0.1411	0.1411
NARX	50	0.1141	0.6799	0.3131	0.0114	0.0114
LSTM	50	0.1150	0.6742	0.3104	0.0181	0.0176
NARX	100	0.1203	0.6992	0.3125	0.0114	0.0079
LSTM	100	0.1207	0.6942	0.3074	0.0101	0.0028
NARX	100	0.1207	0.6780	0.3098	0.0161	0.0137
LSTM	200	0.1238	0.6503	0.3000	0.0443	0.0424
	[0.01 s]	[m]	[m]	[m]	[m]	[m]

Table 4.10: Table point 09

10	STEP	X	Y	Z	$dXYZ$	dYZ
TARGET		0.1284	0.7003	0.2972	0.1466	0.1459
NARX	50	0.1147	0.6766	0.3117	0.0217	0.0212
LSTM	50	0.1102	0.6948	0.3162	0.0222	0.0159
NARX	100	0.1328	0.7029	0.2991	0.0072	0.0041
LSTM	100	0.1294	0.6882	0.2990	0.0122	0.0122
NARX	200	0.1359	0.6940	0.2969	0.0117	0.0063
LSTM	200	0.1317	0.6785	0.2976	0.0219	0.0217
	[0.01 s]	[m]	[m]	[m]	[m]	[m]

Table 4.11: Table point 10

4.6.1 Error comparison of the end effector position for Reaching

Through the figures of this section we want to show how the trend of the error between the target and the position of the end effector at the end of the experiment varies depending on both the network used to predict the motor commands and the selected time horizon. All the figures have on the vertical axis the amplitude of the error in meters, while on the horizontal axis the index of the experiment is shown. We want to remember that the index of experiments increases with the increase in the amplitude of the movement.

The Figures 4.7, 4.8 show how the error varies for each of the two artificial neural network models as the time horizon changes, while from the following Figures 4.9, 4.10, 4.11 we observed the trend of the error by comparing the two networks and keeping the time horizon fixed for each figure.

From Figure 4.7 it can be seen how fixed a time horizon of $0.5s$ and $1s$ and selected the NARX model the experiments are successful with an error value within the tolerance set for small movements while for a greater control horizon the result is not always accurate. As the range of motion increases, we see that for some experiments the error grows excessively. The Figure 4.7 is associated with a Table 4.12 in which it is marked with a symbol ✓when the error of the experiment is less than the tolerance T, while with a symbol $\times$ when this is greater. This table also shows the success rate of the ten experiments we have done, and we can see that the higher percentage is associated with time horizon $t_f = 1s$ with a success rate of 90%. From the Figure 4.8, on the other hand, it can be seen that the experiments having selected the LSTM model have not actually reached the target position with precision, except on a few occasions. Consequently, in the Table 4.13 it can be seen that the success rate remains better for a time horizon $t_f = 1s$, but with lower percentages.

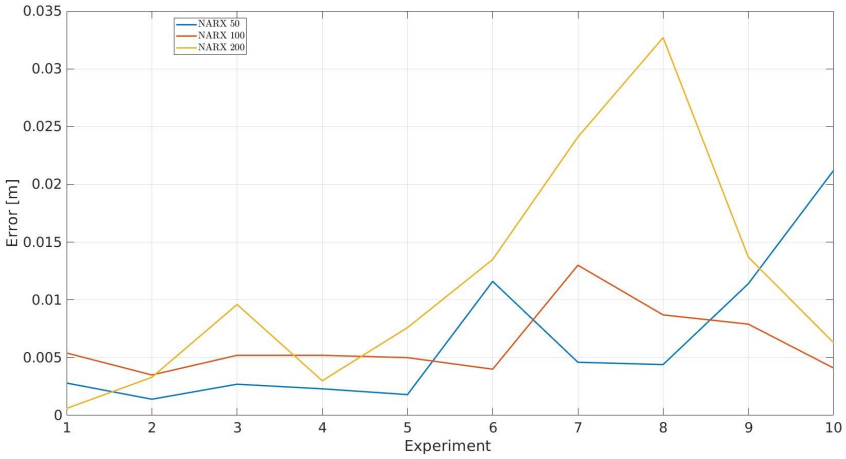

Figure 4.7: Comparison of the NARX error as the time horizons vary

NARX	1	2	3	4	5	6	7	8	9	10	% Succ
50	✓	✓	✓	✓	✓	×	✓	✓	×	×	70
100	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	90
200	✓	✓	✓	✓	✓	×	×	×	×	✓	60

Table 4.12: Error - NARX

From the Figure 4.8, on the other hand, it can be seen that the experiments having selected the LSTM model have not actually reached the target position with precision, except on a few occasions. Consequently, in the Table 4.13 it can be seen that the success rate remains better for a time horizon $t_f = 1s$, but with lower percentages.

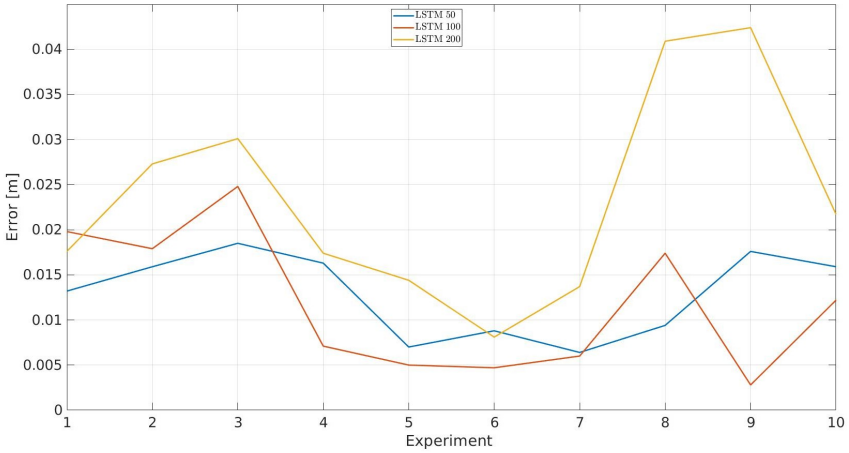

Figure 4.8: Comparison of the LSTM error as the time horizons vary

LSTM	1	2	3	4	5	6	7	8	9	10	% Succ
50	×	×	×	×	✓	✓	✓	✓	×	×	40
100	×	×	×	✓	✓	✓	✓	×	✓	×	50
200	×	×	×	×	×	✓	×	×	×	×	10

Table 4.13: Error - LSTM

The Figures 4.9, 4.10, 4.11 show a comparison of the error of the two networks fixed the time horizon, while in the tables there is a symbol on the line associated with the network that had a better result for the experiment, if the symbol is an × it means that although the result is better the error of the experiment exceeds the set tolerance. The results show that NARX is better than LSTM in most cases regardless of the time horizon that is set.

Now it is interesting to see the results not only of the final position of the end effector so in the next section we present how it behaved throughout the control trajectory.

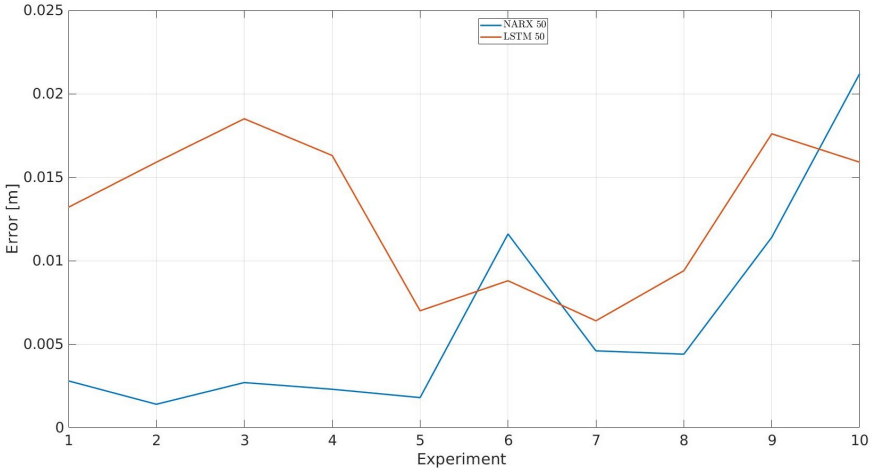

Figure 4.9: Comparison of NARX and LSTM fixed the time horizon $t_f = 0.5s$

50	1	2	3	4	5	6	7	8	9	10	% Succ
NARX	✓	✓	✓	✓		✓	✓	✓	×		80
LSTM					✓					×	20

Table 4.14: Error NARX - LSTM $t_f = 0.5s$

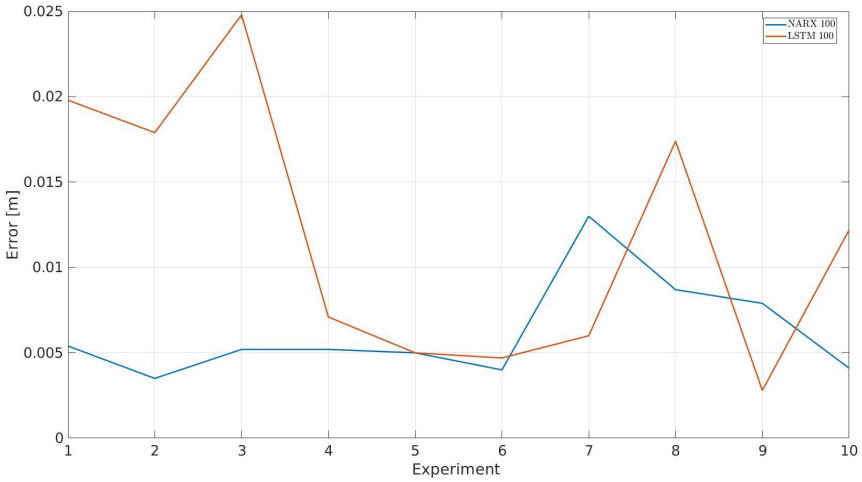

Figure 4.10: Comparison of NARX and LSTM fixed the time horizon $t_f = 1s$

100	1	2	3	4	5	6	7	8	9	10	% Succ
NARX	✓	✓	✓	✓	✓	✓		✓		✓	80
LSTM							✓		✓		20

Table 4.15: Error NARX - LSTM $t_f = 1s$

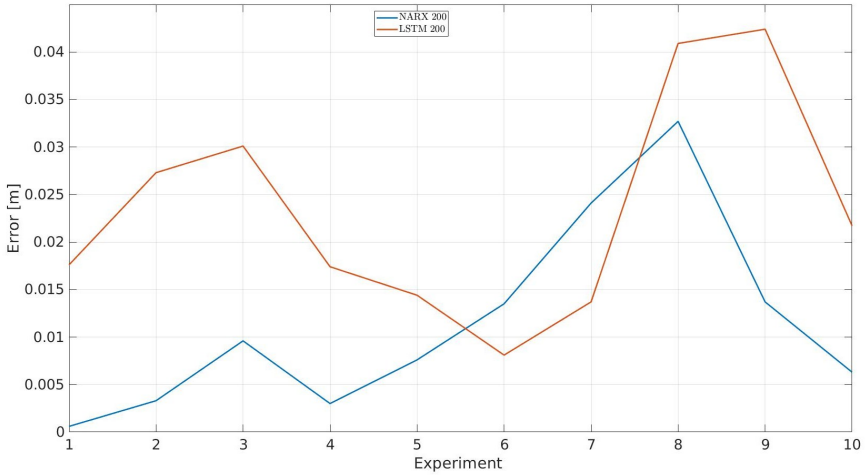

Figure 4.11: Comparison of NARX and LSTM fixed the time horizon $t_f = 2s$

200	1	2	3	4	5	6	7	8	9	10	% Succ
NARX	✓	✓	✓	✓	✓			×	×	✓	80
LSTM						✓	×				20

Table 4.16: Error NARX - LSTM $t_f = 2s$

4.6.2 NARX - End effector behavior

The analysis of the previous section shows us that by using NARX model in many experiments we can obtain good results while in others the error is greater,therefore it is interesting to see if the real robot behaves as predicted by the network using the same estimated motor commands.

The Figures 4.12, 4.13 show the behaviors of the network and the iCub having as input the same vector of motor control commands obtained by the optimization algorithm. As can be seen from the Figure 4.12, in this experiment and with this configuration of β(a factor that penalizes the weight of the motor effort in the cost function), time horizon t_f and selected target point, the behavior of the network and the robot is almost the same. In the Figure 4.13 we will instead graph the difference between the two lines of the previous figure.

The figures 4.12, 4.13 show that the error behavior is not linear, and this may be due to both a modeling error and the noise that characterizes every experiment on a real robot, but in any case the modulus of this difference always remains below 6mm and this is considered an acceptable result.For our purpose, and in view of the error compensation algorithm, we have set a final end effector distance $T <= 0.009m$ as tolerance.

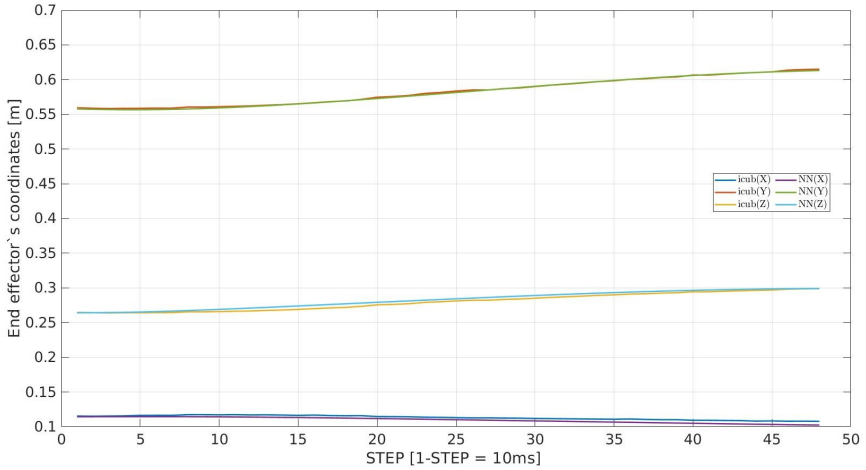

Figure 4.12: Comparison between predicted NARX output and real robot movement [$t_f = 0.5s$, experiment 05]

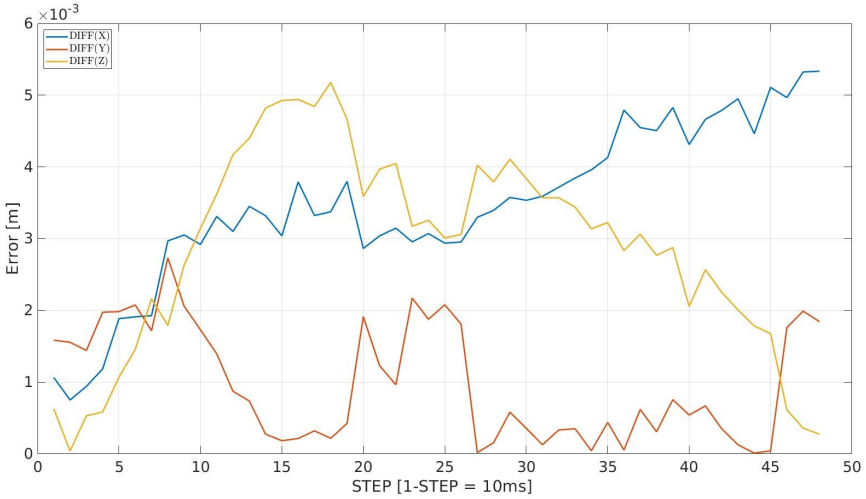

Figure 4.13: Difference between predicted NARX output and real robot movement [$t_f = 0.5s$, experiment 05]

From the tables, however, it can be seen that in some points this difference actually appears greater, so we decided to show for the point at a greater distance (Table 4.10) what happens in detail.

As you can see from the Figures 4.14, 4.15, 4.16, 4.17, also for these configurations the results are satisfactory even if you can see how for longer sequences the error appears greater.

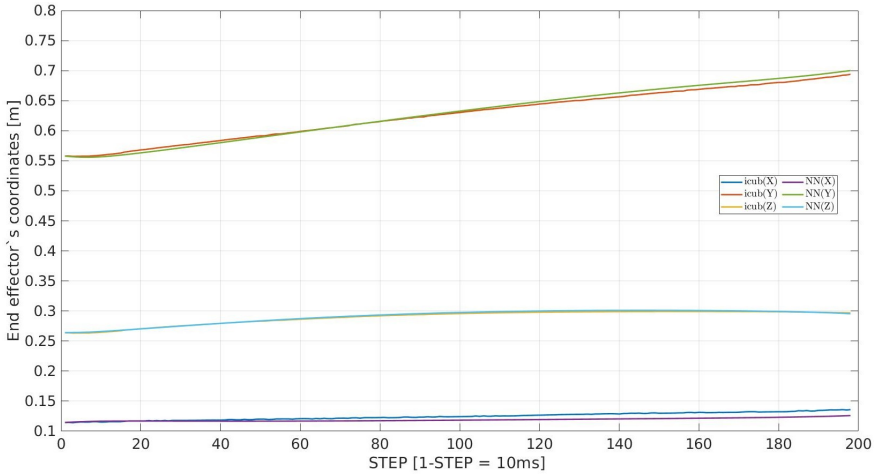

Figure 4.14: Comparison between predicted NARX output and real robot movement [$t_f = 2s$, experiment 10]

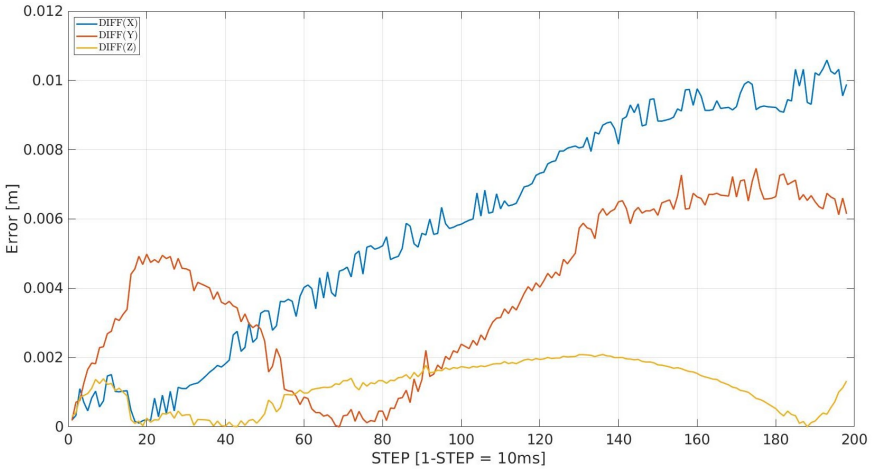

Figure 4.15: Difference between predicted NARX output and real robot movement [$t_f = 2s$, experiment 10]

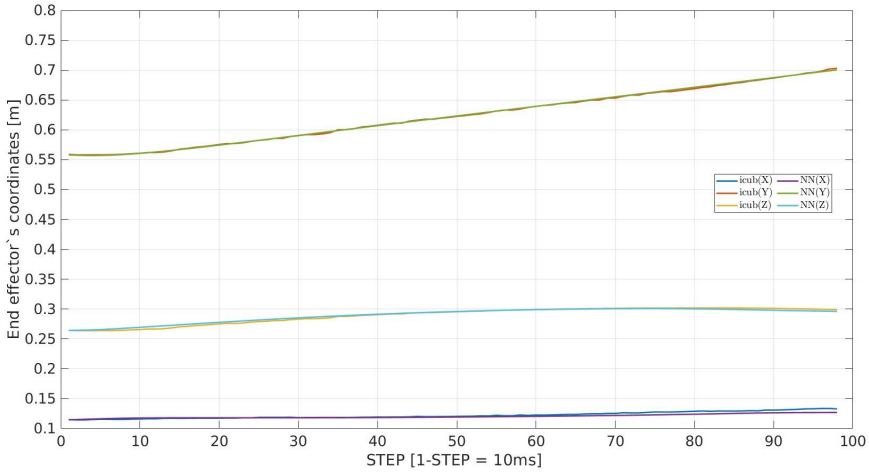

Figure 4.16: Comparison between predicted NARX output and real robot movement [$t_f = 1s$, experiment 10]

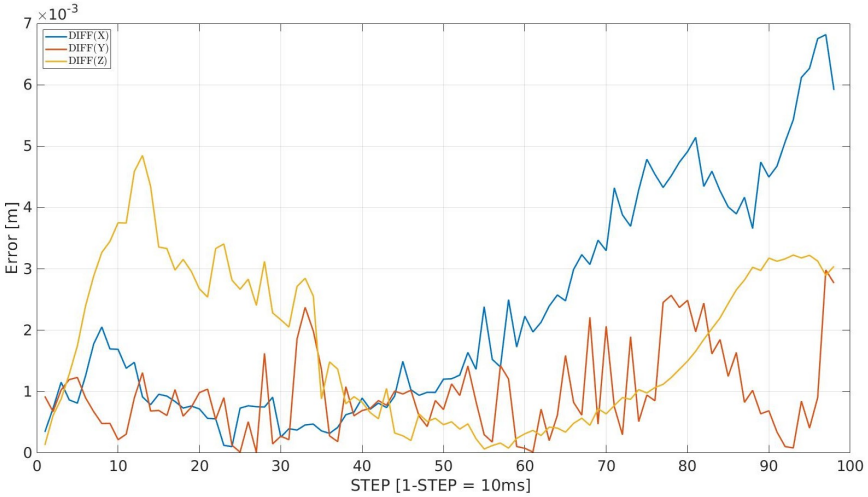

Figure 4.17: Difference between predicted NARX output and real robot movement [$t_f = 1s$, experiment 10]

It is interesting to analyze what happened for the same experiment with an experiment of shorter duration [0.5s].

From the Figure 4.19 it can be seen that the error on the Y axis grows rapidly and then stabilizes. We tried the test again and got a very similar result (Figure 4.20, which allowed us to rule out the fact that it was an unlucky experiment. A possible explanation could be that for such a large movement, a different value of β is needed to reduce the intensity of the variation of the motor commands. Further tests will be done in the future.

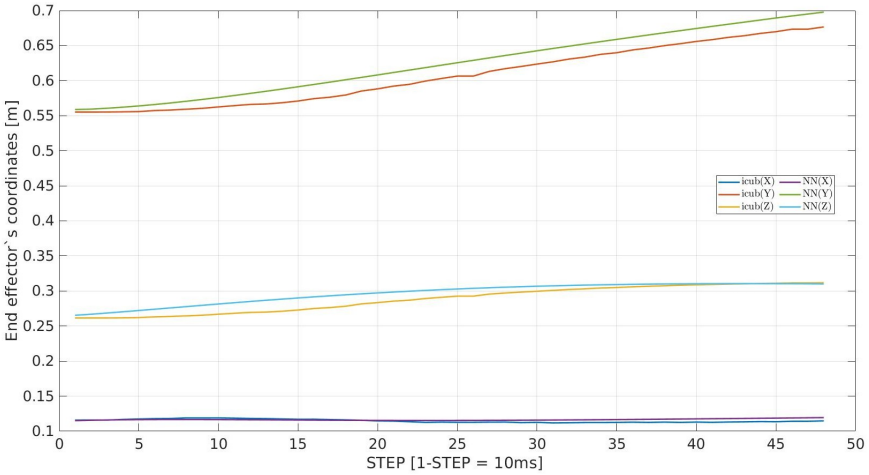

Figure 4.18: Comparison between predicted NARX output and real robot movement [$t_f = 0.5s$, experiment 10]

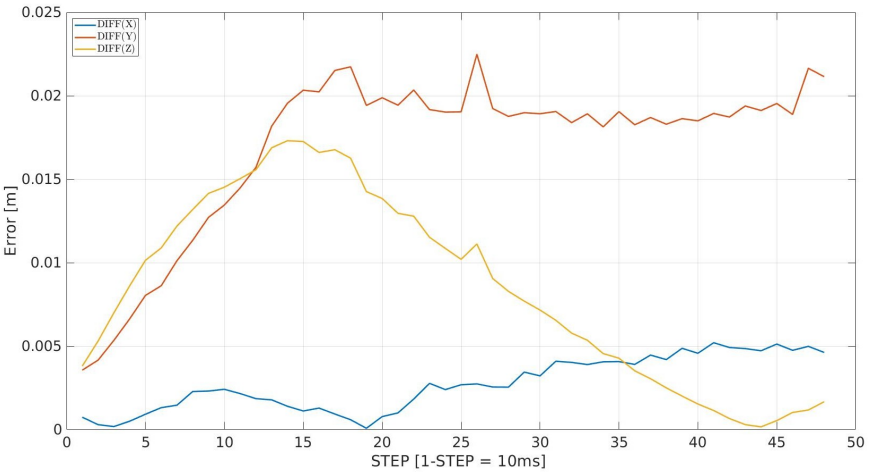

Figure 4.19: Difference between predicted NARX output and real robot movement [$t_f = 0.5s$, experiment 10]

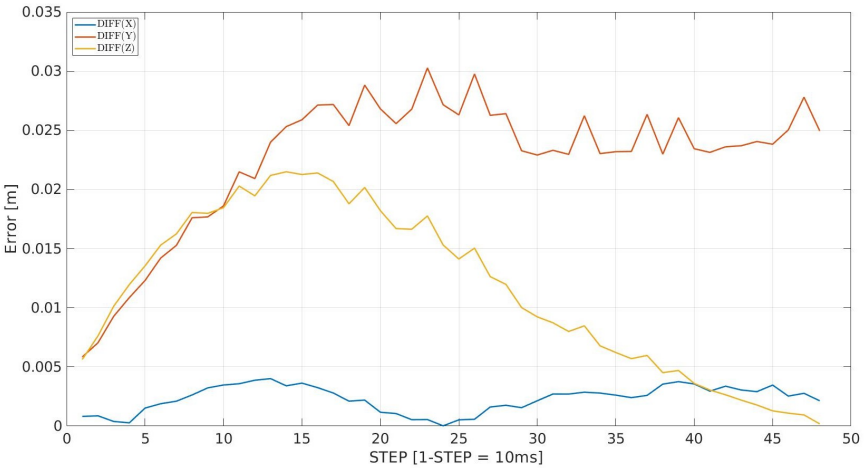

Figure 4.20: Difference between predicted NARX output and real robot movement [$t_f = 0.5s$, experiment 10-B]

4.6.3 LSTM - End effector behavior

The analysis of the section 4.6.1 shows us that the results using the LSTM are worse than those using the NARX, let's look at how the robot behaves having as input the predicted motor commands using the LSTM.

From the Figures 4.21, 4.23, 4.25 you can see that actually the optimization algorithm has obtained values of motor commands that try to bring us to the target, but they seem being of insufficient intensity, in fact, the robot is almost never able to reach the point respecting the set tolerance.

This could be due to an insufficiently accurate model selection, as can be seen from the training chart of the LSTM 4.5 the minimized loss function has irregularities that could be a sign of not very good learning by the network.

From the Figures 4.22, 4.24, 4.26, the figures where the trend of the error is shown, you can see how the main increase of the error is in the initial phase, and this could suggest a greater weight of β of the cost function, a sudden change of reference of the motor could bring a greater error in the initial phase.

In the future we will certainly try to collect more data to investigate these critical issues.

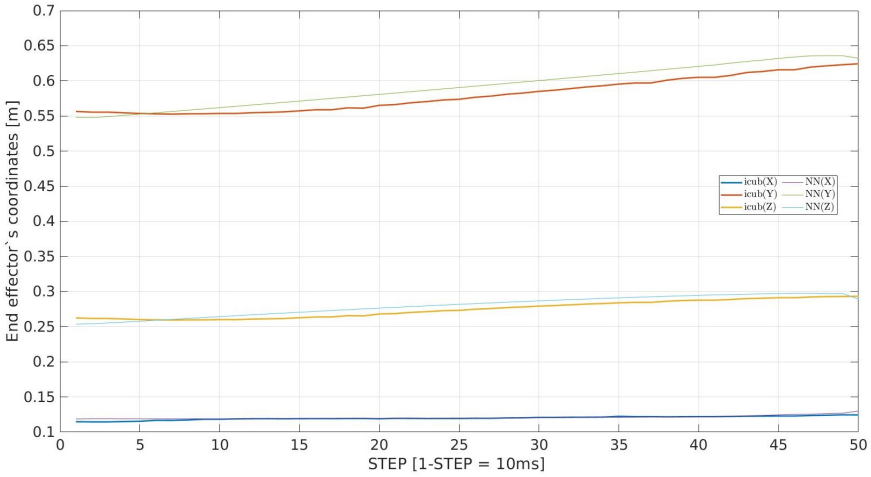

Figure 4.21: Comparison between predicted NARX output and real robot movement [$t_f = 0.5s$, experiment 06]

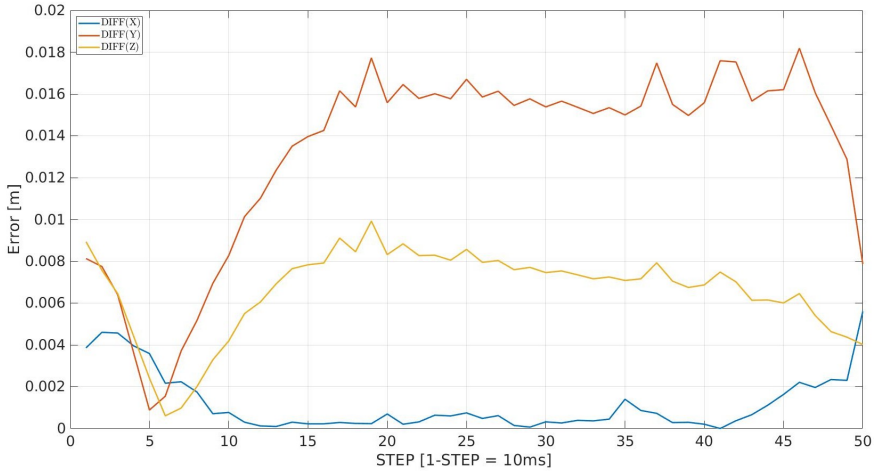

Figure 4.22: Difference between predicted NARX output and real robot movement [$t_f = 0.5s$, experiment 06]

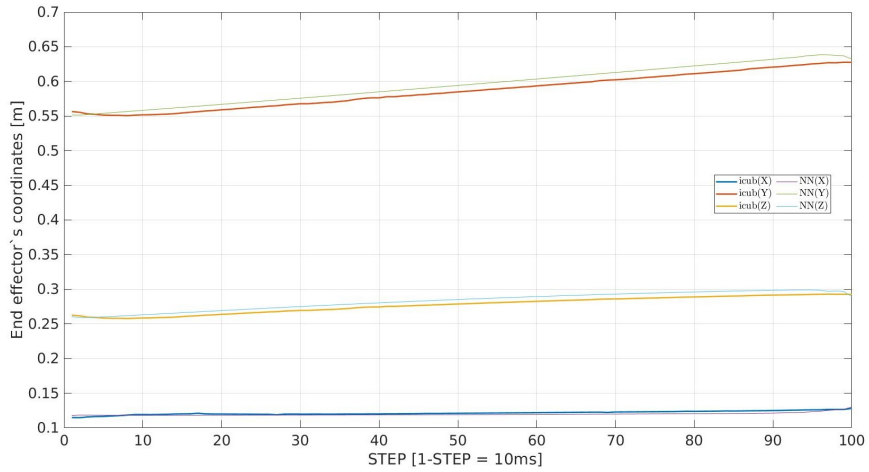

Figure 4.23: Comparison between predicted LSTM output and real robot movement [$t_f = 1s$, experiment 06]

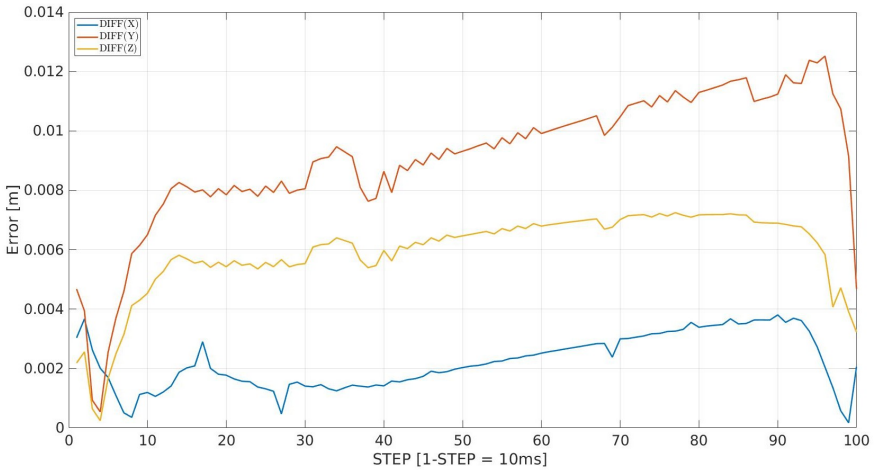

Figure 4.24: Difference between predicted LSTM output and real robot movement [$t_f = 1s$, experiment 06]

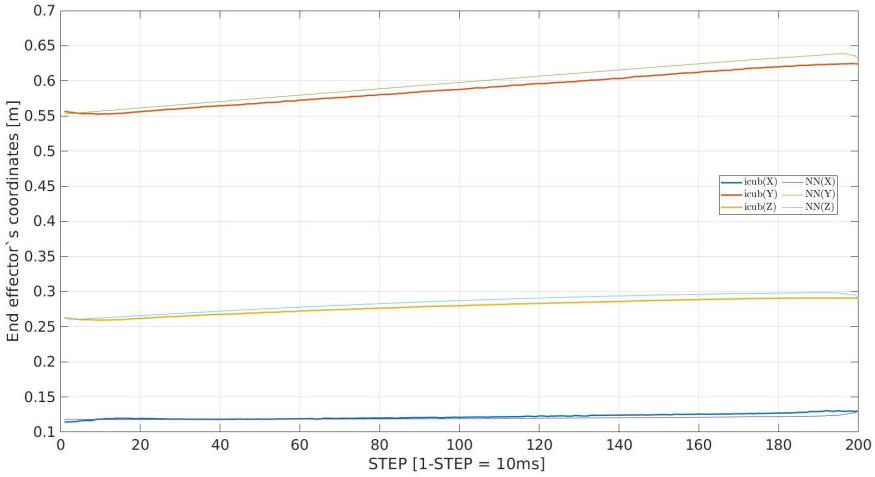

Figure 4.25: Comparison between predicted LSTM output and real robot movement [$t_f = 2s$, experiment 06]

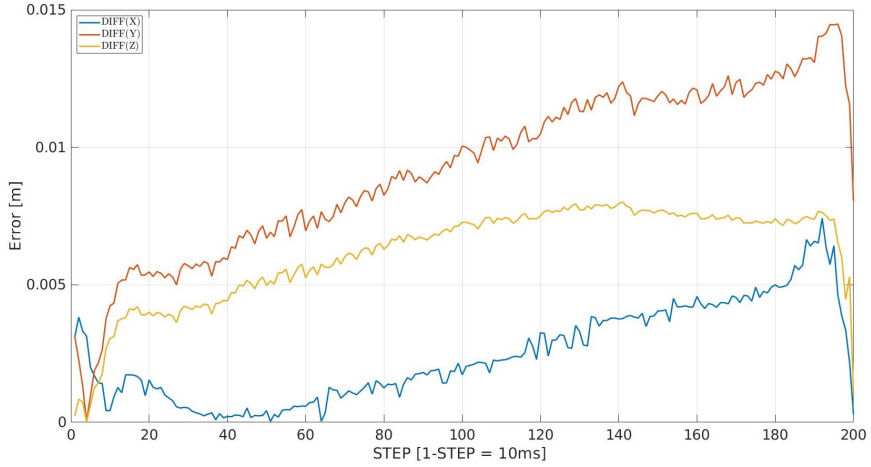

Figure 4.26: Difference between predicted LSTM output and real robot movement [$t_f = 2s$, experiment 06]

4.7 Preliminary Throwing Results

To do the throwing experiments we had to face a new challenge, namely to solve the problem of grasping the ball to be thrown. The robotic arm of the iCub we used also allows the use of a robotic hand that can be controlled in position. We therefore found a configuration of the fingers able to keep the ball steady during movement and an opening configuration in order to release the object. As can be seen from the Figures 4.27 due to the geometry of the ball we are

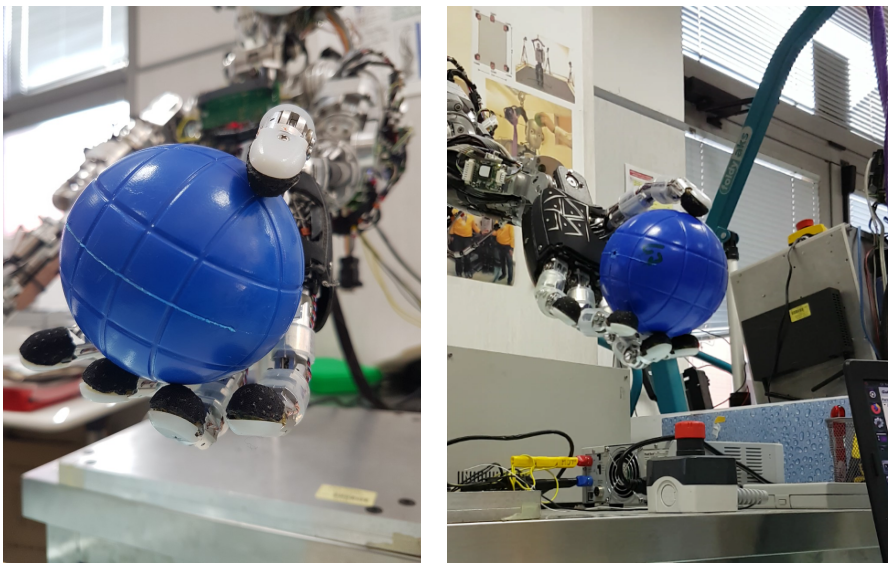

Figure 4.27: Examples of closed hand setup

unable to set the contact point of the object exactly at the position where the end effector is estimated to be, i.e. the center of the hand. This problem can be easily solved by adding an offset to the throwing model.

The dynamics of the motors that are used for the hand and fingers are however much slower than that of the rest of the arm. For this reason, we decided to empirically evaluate the time required to perform this operation. The experiment consisted of changing the position reference from "closed hand" to "open hand" and assessing the time needed to see the ball release. From the experiments we have obtained that the command arrives to the motors in less than $0.005s$ and the actual time to see the hand open is $0.37s \pm 0.05s$ for a maximum time of $0.425s$. This parameter, which is characteristic of the specific open/closed configuration of the hand, requires us to give the motor command to release the object at least $0.46s$ before the release point obtained by optimization is reached. For this reason, for future throwing experiments it will be necessary to ensure that this time is actually respected. Finally, we show the only throwing experiment we have done on the robot using the same NARX model used for the reaching. In this experiment, the target point is no more a point in the robot's task space, but it is a point outside its operating region. The optimization as explained in the throwing model Section 2.5.2 tries to find the motor control commands and the release point necessary to throw the ball at the desired target point. Given the problems in using the hand, for now we only have data relating to the movement of the arm and not those relating to the final position that the ball reaches.

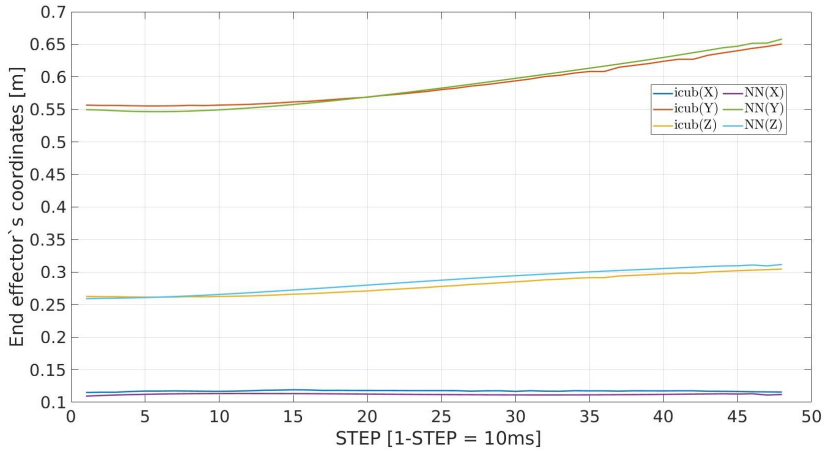

Figure 4.28: Comparison between predicted NARX output and real robot movement [$t_f = 0.5s$, Throwing]

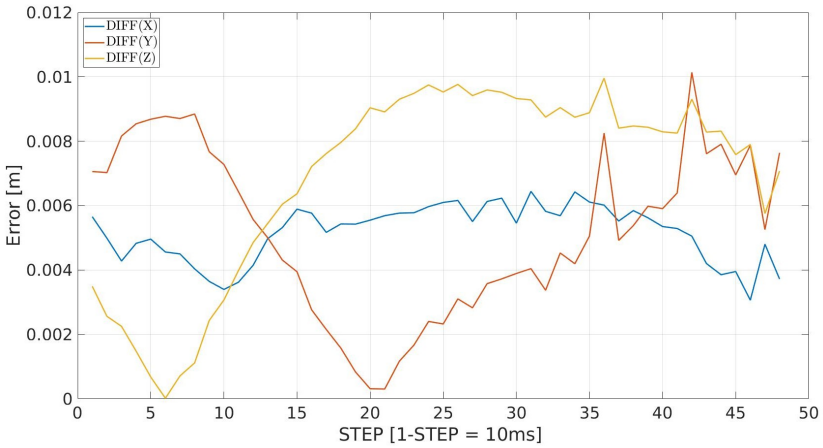

Figure 4.29: Difference between predicted NARX output and real robot movement [$t_f = 0.5s$, Throwing]

The Figures 4.28, 4.29 show us that the trend of the arm also for this type of task is similar to the predicted one, although it is possible to notice an initial error probably due to an imprecision of the initial setup perhaps caused by the gripping of the ball at the beginning of the test.

Chapter 5

Conclusions

In this work we proposed a cartesian model-free method, using recurrent neural networks, for the control of a humanoid robot arm. The controller was employed successfully for reaching tasks and preliminarily extended for object throwing tasks. The work is based on previous works of Thuruthel and colleagues [1], [2] which proposed a learning of the dynamic model for open loop predictive control of soft robotic manipulators and a stable open loop controller for the same robot.

By relying on this approach, we tested the dynamic model learning method, starting from a task space exploration algorithm using the arm of the iCub humanoid robot. We also performed a comparison between two RNNs (NARX and LSTM) used to learn the forward dynamic model of the robot arm.

The results of the reaching tasks performed as a combination of a RNN and the optimization algorithm demonstrate that the proposed approach allows the robot arm to produce reaching movements with a success rate of 90% in the best case.

The comparison of the two RNNs shows important differences in the performances. Specifically for NARX, we have observed that shorter trajectories guarantee better results, i.e. the final position error does not exceed the set tolerance (0.009m). In fact, for experiments with time horizon $t_f = 0.5s$ the success rate is 70%, for those with $t_f = 1s$ the success rate is 90% while for those with $t_f = 2s$ seconds the rate drops to 60%. This is because the prediction error of the learned model tends to increase as the time increases or for too short experiments. For the LSTM network, on the other hand, we observed a larger and more discontinuous error that tends to increase with the increasing prediction horizon, as for the NARX but in the best case fixed $t_f = 1s$ the success rate is 50%. We believe that LSTM performances could be improved with a more accurate model selection, in fact the Figure 4.5 shows a not perfect behavior in the learning phase.

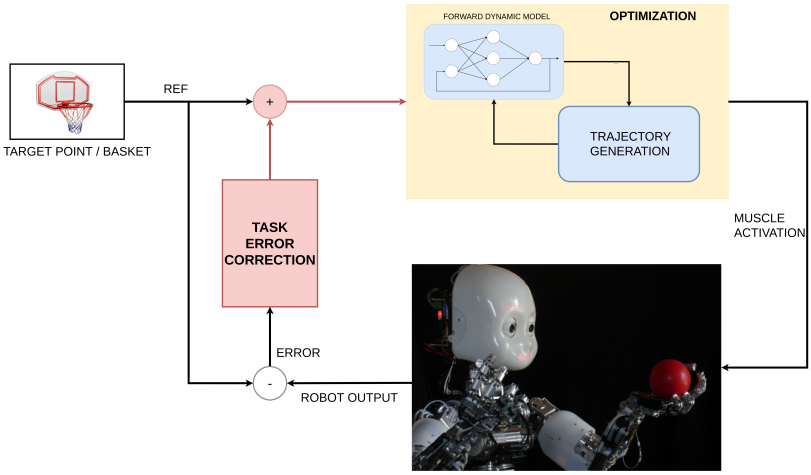

Figure 5.1: Control scheme architecture with error compensation

For the part relating to throwing we have not been able to collect data, but we have described and implemented a model for solving this problem (Section 2.5.2). The proposed model is based on the use of the same model trained for the reaching experiments, and by changing only the cost function that the optimization uses to find the motor commands it is possible to perform the throwing task.

To improve the robustness of the results, one could consider collecting new reaching experiments to refine the analysis, try to refine the model selection of the LSTM network and perform more throwing experiments. Another possible extension to the work, as shown in Figure 5.1, would be the introduction of a further phase of error compensation, both to improve the reaching experiments but above all to go to compensate for all the non considered errors, such as for example that due to friction of the air or the error introduced by the addition of weight on the robot hand, in the throwing experiment.

Finally, we believe that the proposed model can be useful in the modeling of robots of different nature such as musculoskeletal robots, where the reproduction of muscles is developed through the elongation and contraction of elastic elements and other elements that are not easily modeled or controllable.

Bibliography

[1] Thomas George Thuruthel, Egidio Falotico, Federico Renda, and Cecilia Laschi. Learning dynamic models for open loop predictive control of soft robotic manipulators. *Bioinspiration & Biomimetics*, 12, 08 2017.

[2] T. G. Thuruthel, E. Falotico, M. Manti, and C. Laschi. Stable open loop control of soft robotic manipulators. *IEEE Robotics and Automation Letters*, 3(2):1292–1298, April 2018.

[3] Fumiya Iida and Auke Jan Ijspeert. *Biologically Inspired Robotics*, pages 2015–2034. Springer International Publishing, Cham, 2016.

[4] Terrence Fong, Illah Nourbakhsh, and Kerstin Dautenhahn. A survey of socially interactive robots. *Robotics and autonomous systems*, 42(3-4):143–166, 2003.

[5] Brian R Duffy. Anthropomorphism and the social robot. *Robotics and autonomous systems*, 42(3-4):177–190, 2003.

[6] Comau. Amico. `https://www.comau.com/it/pages/this_is_comau/innovation/amico.aspx`, Accessed 2020-04-01.

[7] Jesús Retto. Sophia, first citizen robot of the world. *ResearchGate https://www. researchgate. net*, pages 2–9, 2017.

[8] Ichiro Kato. " the wabot-1" an information-powered machine with sences and limbs. *Bulletin of Science and Engineering Research Laboratory*, 62, 1973.

[9] Ichiro Kato, Sadamu Ohteru, Katsuhiko Shirai, Toshiaki Matsushima, Seinosuke Narita, Shigeki Sugano, Tetsunori Kobayashi, and Eizo Fujisawa. The robot musician 'wabot-2'(waseda robot-2). *Robotics*, 3(2):143–155, 1987.

[10] Giorgio Metta, Lorenzo Natale, Francesco Nori, Giulio Sandini, David Vernon, Luciano Fadiga, Claes Von Hofsten, Kerstin Rosander, Manuel Lopes,

José Santos-Victor, et al. The icub humanoid robot: An open-systems platform for research in cognitive development. *Neural Networks*, 23(8-9):1125–1134, 2010.

[11] Kuniyuki Takahashi, Tetsuya Ogata, Jun Nakanishi, Gordon Cheng, and Shigeki Sugano. Dynamic motion learning for multi-dof flexible-joint robots using active–passive motor babbling through deep learning. *Advanced Robotics*, 31(18):1002–1015, 2017.

[12] T. Hester, M. Quinlan, and P. Stone. Generalized model learning for reinforcement learning on a humanoid robot. *2010 IEEE International Conference on Robotics and Automation*, pages 2369–2374, May 2010.

[13] Y. Karayiannidis and Z. Doulgeri. Model-free robot joint position regulation and tracking with prescribed performance guarantees. *Robotics and Autonomous Systems*, 60(2):214 – 226, 2012.

[14] S. G. Khan, G. Herrmann, F. Lewis, T. Pipe, and C. Melhuish. A q-learning based cartesian model reference compliance controller implementation for a humanoid robot arm. *2011 IEEE 5th International Conference on Robotics, Automation and Mechatronics (RAM)*, pages 214–219, Sep. 2011.

[15] D. Braganza, D. M. Dawson, I. D. Walker, and N. Nath. A neural network controller for continuum robots. *IEEE Transactions on Robotics*, 23(6):1270–1277, 2007.

[16] V. Falkenhahn, A. Hildebrandt, R. Neumann, and O. Sawodny. Dynamic control of the bionic handling assistant. *IEEE/ASME Transactions on Mechatronics*, 22(1):6–17, 2017.

[17] M. Plooij, W. Wolfslag, and M. Wisse. Open loop stable control in repetitive manipulation tasks. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 949–956, 2014.

[18] N. Sugimoto, V. Tangkaratt, T. Wensveen, T. Zhao, M. Sugiyama, and J. Morimoto. Trial and error: Using previous experiences as simulation models in humanoid motor learning. In *IEEE Robotics Automation Magazine*, volume 23(1), pages 96–105, 2016.

[19] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. *Proceedings of Robotics: Science and Systems (RSS)*, 2019.

[20] TOYOTA. CUE-3. `https://global.toyota/en/newsroom/corporate/28595150.html`, Accessed 2020-04-01.

[21] GANGHUA SUN and BRIAN SCASSELLATI. A fast and efficient model for learning to reach. *International Journal of Humanoid Robotics*, 02(04):391–413, 2005.

[22] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural Computation*, 12(10):2451–2471, 2000.

[23] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[24] Ugo Pattacini, Francesco Nori, Lorenzo Natale, Giorgio Metta, and Giulio Sandini. An experimental evaluation of a novel minimum-jerk cartesian controller for humanoid robots. In *2010 IEEE/RSJ international conference on intelligent robots and systems*, pages 1668–1674. IEEE, 2010.

[25] Vadim Tikhanoff, Angelo Cangelosi, Paul Fitzpatrick, Giorgio Metta, Lorenzo Natale, and Francesco Nori. An open-source simulator for cognitive robotics research: the prototype of the icub humanoid robot simulator. In *Proceedings of the 8th workshop on performance metrics for intelligent systems*, pages 57–61, 2008.

[26] Enrico Mingo Hoffman, Silvio Traversaro, Alessio Rocchi, Mirko Ferrati, Alessandro Settimi, Francesco Romano, Lorenzo Natale, Antonio Bicchi, Francesco Nori, and Nikos G Tsagarakis. Yarp based plugins for gazebo simulator. In *International Workshop on Modelling and Simulation for Autonomous Systems*, pages 333–346. Springer, 2014.

[27] Lorenzo Natale, Ali Paikan, Marco Randazzo, and Daniele E Domenichelli. The icub software architecture: evolution and lessons learned. *Frontiers in Robotics and AI*, 3:24, 2016.

[28] G Metta, P Fitzpatrick, and L Natale. Towards long-lived robot genes, 2007.

[29] Francesco Nori, Silvio Traversaro, Jorhabib Eljaik, Francesco Romano, Andrea Del Prete, and Daniele Pucci. icub whole-body control through force regulation on rigid non-coplanar contacts. *Frontiers in Robotics and AI*, 2:6, 2015.

[30] Stefano Nardo. An empirical comparison of recurrent neural networks on sequence modeling. Tesi di laurea, Università di Pisa, 2018/2019.

[31] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

Ringraziamenti

I miei più cari ringraziamenti vanno ad Egidio e Lorenzo, che oltre ad avermi dato la possibilità di lavorare in laboratorio con loro mi hanno sostenuto fino alla fine soprattutto nei momenti più critici di questo periodo di tesi. Oltre a loro devo ringraziare tutti i ragazzi del laboratorio che oltre ad avermi fatto una grande compagnia sono sempre stati disponibili ad aiutarmi e a confrontarsi sul lavoro che stavo facendo. Un ringraziamento è dovuto anche al professor Matteucci per la fiducia che mi ha dato senza la quale questo periodo di lavoro non sarebbe stato possibile.

Un ringraziamento va anche a tutta la mia famiglia, a mio babbo, a mia mamma, le mie zie, mio fratello, le mie tre magnifiche sorelle, ai loro compagni e i miei nove nipoti, insomma a tutto l'asilo che mi ritrovo ogni volta che torno a casa. Senza il loro sostegno e il loro affetto non avrei trovato né gli stimoli né la forza per scoprire cosa la realtà ha da offrirmi tutti i giorni.

Non posso dimenticare tutti gli amici del Poli e in particolare Polifemo che ha sempre vegliato su di me e i PoliUltras, una banda di matti che mi ha totalmente rivoluzionato il cuore e mi ha mostrato che non c'è sconfitta nel cuore di chi lotta ma comunque Poli 1 - Catto 0.

Un ultimo e caro ringraziamento va a tutti i Regaz di Rimmini che da sempre sopportano tutto il mio sentimentalismo e animo polemico e che tra una piada e una cantata in piazzetta mi hanno insegnato che l'unico criterio nella vita è il cuore.