ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA
CESENA CAMPUS

School of Engineering and Architecture
Second Cycle Degree/Two-year Master in
Computer Science and Engineering

Novel robotic skill synthesis with Conditional Neural Movement Primitives

Master's thesis in
INTELLIGENT ROBOTIC SYSTEMS

Supervisor
Prof. Andrea Roli

Co-Supervisor
Prof. Emre Uğur
(Boğaziçi University, Istanbul)

Candidate
Igor Lirussi

Academic Year 2022-2023

Abstract

Humans are capable of executing a wide variety of complex tasks, based on prior experience. Often, they accomplish them by breaking them down into minor actions that are composed together one after the other to achieve the goal. These actions are not always learned directly but adapted from previous similar experiences to the current context.

In this study, we propose a computational model that is biologically inspired and aims to integrate into robotics the human ability to adapt movements and combine them to achieve high-level skills.

A novel approach to high-level skill synthesis is explored by leveraging movement primitives learned through Conditional Neural Motion Planning (CNMP) models.

The research introduces two methods for generating and composing new actions based on demonstrated ones. In the first approach, trajectories are blended by utilizing the task interpolation capabilities of the neural network and a developed mathematical system for parameterization. Additionally, two alternative architectures for the CNMP model are proposed, both achieving results comparable to the original model while accommodating partial information. The second approach achieves action synthesis through the concatenation of primitives, spatial interpolation, and the network's ability to encode multidimensional data to embed the environment representation.

Both proposed methods are finally showcased in several experiments with real robots.

To my family and to the time we have.

Acknowledgements

First of all, I would like to thank my advisors, Prof. Emre Uğur and Prof. Andrea Roli, for their support and guidance during this research.

I want to thank the University of Bologna, the teachers, and the personnel for making this journey of academic and personal growth possible.

I extend my gratitude to all members of CoLoRs Laboratory in Boğaziçi University for their advice and kindness, especially Yigit and Alper.

I am thankful to my foreign friends, Bahadir, Naz, Ipek, and Zeynep, for their moral and logistical help abroad.

I also thank my Italian friends for their affection, which enables me to explore distant lands far away from home without feeling lost.

I am grateful to all the pilots of the 2nd Tactical Transport Squadron of the 46th Air Brigade for their support.

A heartfelt thank you to Cyrus for his immense support.

I also express my gratitude to Beppino for always being present.

Finally, I want to express my deepest gratitude to my family.

I have been able to persevere in my education thanks to their unconditional love and support.

Contents

Abstract **iii**

1 Introduction **1**
1.1 Overview . . . 1
1.2 Challenges . . . 3
1.3 Objectives . . . 4
1.4 Thesis Structure . . . 5

2 Related Work **7**
2.1 Gaussian Processes . . . 10
2.2 Conditional Neural Processes . . . 11
2.3 Conditional Neural Movement Primitives . . . 13

3 Platforms **15**
3.1 Baxter Robot . . . 15
3.2 UR10 Robot . . . 17
3.3 3F Robotiq Gripper . . . 18
3.4 FT 300-S Force Torque Sensor . . . 19
3.5 Frameworks . . . 20

4 Design **23**
4.1 Partial Skill Combination . . . 24
4.1.1 CNMP model with task-parameter only in condition . . . 29
4.1.2 CNMP model with task-parameter only in query . . . 31
4.1.3 Comparison of the previous models . . . 33
4.1.4 CNMP changing task in time with one conditioning point . . . 35
4.1.5 CNMP changing task in time with multiple conditioning points 38
4.2 End-To-End Skill Concatenation . . . 42
4.2.1 CNMP for skill concatenation . . . 44
4.2.2 CNMP embedding environment representation . . . 46

5 Implementation **49**
5.1 Partial Skill Combination 52
5.1.1 CNMP model with task-parameter only in condition 52
5.1.2 CNMP model with task-parameter only in query 53
5.1.3 Comparison of the previous models 54
5.1.4 CNMP changing task in time with one conditioning point . . 55
5.1.5 CNMP changing task in time with multiple conditioning points 55
5.2 End-To-End Skill Concatenation 59
5.2.1 Building the graph of concatenated trajectories 59
5.2.2 Pruning the graph of concatenated trajectories 59

6 Validation and Testing **61**
6.1 Partial Skill Combination 62
6.2 End-To-End Skill Concatenation 66

7 Conclusions **71**
7.1 Future work 71

List of Figures

2.1 In ProMPs the distribution of basis functions is often represented using a probabilistic framework, but can lead to some failures. . . . 9
2.2 Gaussian Processes are probabilistic models that define a distribution over functions. 10
2.3 The structure of a CNP network, with three main blocks: Encoder, Aggregator, and Decoder. 11
2.4 CNP allows precise predictions for targets sampled from a distribution conditioned with observations 12
2.5 CNMP model, it is conceived to work with temporal relations between sensorimotor data and different task parameters 13

3.1 Baxter Robot platform 16
3.2 UR10 Robot platform 17
3.3 3F Robotiq Gripper platform 18
3.4 FT 300-S Force Torque Sensor 19

4.1 Trajectories for two different skills demonstrated, in different colors. The generated ones in grey color, from the black conditioning points. 24
4.2 Examples of failing approaches with stitching in CNMPs. On the left, simple combination of parts. On the right, stitching the right part using a conditioning point at the end of the left part. 25
4.3 CNMP modeling uncertainty for a condition. In the parametrized CNMP, uncertainty is solved, and skills are encoded with task parameters 1 for ascending and 2 for descending. 26
4.4 Plot of the interpolation abilities of CNMPs also in the task-parameter dimension. 27
4.5 Influence of task parameter from the conditioning points and from the queries. 28
4.6 Architecture of the CNMP model proposed without the original task parameter in the queries. 29

4.7 Interpolation comparison of CNMP model vs CNMP model with TP only in the condition. . . . 30
4.8 Architecture of the CNMP model proposed without the original task parameter in the observations. . . . 31
4.9 Interpolation comparison of CNMP model vs CNMP model with TP only in the query. . . . 32
4.10 Interpolation comparison of the 3 CNMP models analyzed for a different dataset and multiple tasks. . . . 34
4.11 Transition in time of task through task parameter shifting 35
4.12 Different functions for changing task parameter in the CNMP network 36
4.13 A conditioning point that varies task parameter emulates two different conditioning points of different task parameters. . . . 37
4.14 Procedure to individuate multiple conditions for multiple shifts in the task parameters. . . . 38
4.15 Graph and full path in time of the condition point used to transition multiple tasks parameters 39
4.16 Final results of multiple tasks transition and comparison with traditional CNMP network. . . . 41
4.17 Artificial dataset of different skills to concatenate. On the left, there are the demonstrations. On the right, there is the validation. . . . 42
4.18 An example of failing concatenation of skills due to abrupt jumps. 43
4.19 An example of using interpolation with CNMP model to generate new consecutive smooth trajectories. . . . 44
4.20 An example of a recursive concatenation of movement primitives using interpolation abilities of CNMPs. . . . 45
4.21 An example of filtering the skills generated based on extrapolation limitations. . . . 46

6.1 Some moments from learn by demonstration. On the left, action to overcome obstacles. On the right, action to pass under a tunnel. . 63
6.2 The 6 dimensions of a recorded trajectory on a real robot and the resulting mixed trajectory in grey generated by the network. . . . 64
6.3 Some instants from the playback of the trajectory generated where the robot successfully completes the combination of the two actions. 65
6.4 Teaching by demonstration three different actions. From left to right: move from the ground to the green block, move on to the yellow block, and overcome the blue obstacle. . . . 68
6.5 Results of the concatenation method given a starting point in front of the blue box and end point on the yellow box 69

6.6 Results of the concatenation method given a starting point in the back of the blue box and end point on the green box 70

List of Tables

4.1 Comparison Table of errors of original CNMP vs CNMP with TP only in condition 29
4.2 Comparison Table of errors of original CNMP vs CNMP with TP only in query 32
4.3 Full Comparison Table of errors of CNMP vs CNMP with TP only in condition vs CNMP with TP only in query 33
4.4 Example of CNMP ability to embed environment representations . 47

List of Symbols

t	Time
r	Representation
x	Target point (query point)
$SM(t)$	Sensorimotor data at time t
E	Encoder network
Q	Decoder network
O	Set of observation points (conditioning points)
D	Set of demonstration trajectories (expert demonstrations)
T	Set of target points x desired
α	Blending parameter *or* scale
$\beta_t(i)$	Backward variable
γ	External parameter (Task parameter) (TP)
θ	Parameter set of Decoder Network
ϕ	Parameter set of Encoder Network
σ	Standard Deviation
μ	Mean
τ	Trajectory

List of Acronyms/Abbreviations

1D	One Dimensional
2D	Two Dimensional
3D	Three Dimensional
IR	Infrared
ROS	Robot Operating System
RGB	Red Green Blue
RGBD	Red Green Blue Depth
SL	Supervised Learning
LfD	Learning from Demonstration
ML	Machine Learning
DL	Deep Learnign
GMM	Gaussian Mixture Model
HMM	Hidden Markov Model
MSE	Mean Squared Error
ReLU	Rectified Linear Unit
GP	Gaussian Processes
NP	Neural Processes
CNP	Conditional Neural Processes
DNP	Dynamic Movement Primitives
Pro-MP	Probabilistic Movement Primitives
CNMP	Conditional Neural Movement Primitives
TP	Task Parameter
YOLO	You Only Look Once model
UR10	Universal Robot 10
DOF	Degrees of freedom
IMU	Inertial measurement unit
RPC	Remote Procedure Call
CPU	Central Processing Unit
GPU	Graphic Processing Unit

Listings

5.1 CNMP model code 50
5.2 CNMP model forward function 51
5.3 CNMP model loss function for trainig 51
5.4 CNMP model architecture change for TP only in conditions 52
5.5 CNMP training code change for TP only in conditions 53
5.6 CNMP model architecture change for TP only in query 54
5.7 CNMP training code change for TP only in query 54
5.8 Extract of code for comparison of models 55
5.9 Extract of code for task shift with one observation 56
5.10 Extract of code for task shift with multiple observations, couple of observation finding 57
5.11 Extract of code for task shift with multiple observations, couple of observation interpolating 57
5.12 Extract of code for task shift with multiple observations, using the matrix obtained 58
5.13 Extract of code for building the graph of skills concatenated 59
5.14 Extract of code for filling the graph of skills concatenated 60
5.15 Extract of code for filling the graph of skills concatenated 60

Chapter 1

Introduction

1.1 Overview

Humans have a remarkable ability to achieve complex goals in a wide variety of tasks. A person is usually exposed to different scenarios during the day, starting from the home environment to the commute, work, mealtime, and so on. The versatility of our species is a key factor, and the human cognitive flexibility has been appointed as a major driver in evolution [8], [27]. Some situations are more complicated than others; nevertheless, regardless of their difference, humans excel in meeting the different demands to solve the tasks desired.

All these scenarios present different small challenges to solve in order to accomplish the desired high-level goal. Humans switch contexts in a really flexible and natural way and constantly take care of the multitude of these small problems that are faced to complete the desired objective. For example, a general task can be divided into subtasks, which can then be further divided into smaller ones [33]. The strategy of breaking down intricate objectives into smaller, manageable, simpler activities is the most widely used heuristic to solve problems [10].

Many of these sub-challenges require an interaction with one or more objects. For example, the action of opening involves a door to pass through it while moving or to access the fridge for cooking. The reaching action can imply an object like a pen in the office to write or a glass of water to drink. To push as an action often implies a button to enable a device in the workplace, or to turn on a car to commute, or the stove to heat a meal. Objects have undoubtedly strong importance in the small actions performed to achieve a goal, and their affordance is still the object of research in humans [35], [38] and machines [15].

As seen, many different movements and sub-actions, often involving objects, are executed in daily life. Furthermore, they are also adapted to accomplish the current desired goals. The adaptation can involve a simple difference of position

with respect to the previous location, both of the object or the executor, or can involve a completely different context to which the action learned is transferred. These skills are learned and discovered at the beginning, and then the knowledge of the action is abstracted and adapted to different purposes.

Moreover, a person builds sequences of actions naturally to achieve the objective and, as discussed, adapts them to the environment. The skills are often combined together one after the other, based on the scenario but also based on the result and position of the previous execution. Occasionally, it can happen that part of an action is used and part of another action, mixing previously learned movements if the situation requires it. This results in the creation of new combinations and compositions of previously known activities.

Lastly, dissecting complex challenges requires also decision-making under uncertainty, which is essential for achieving high-level goals since the sequence of activities is not always clear in advance. Often, the goal changes mid-way in response to the environment, or the initial assessment is sub-optimal or incorrect, forcing a change in planning and a new decision on what subsequent action to take. So it's worth noting that online decision under dynamic circumstances and change of skill executed allows a person to navigate the complexities of daily scenarios with success.

The human mind's capacity for abstraction, planning, and execution is still a remote objective for robotics [32]. This level of adaptation to the environment and building of compounded behaviors is still a hard challenge to solve nowadays.

For this reason, robots currently are not pervasive in society like other technologies. Humanoid robots have little if no presence and, despite the potential different uses, are relegated to mainly interaction and exhibition duties. The majority of robots work in a controlled environment, like factories, where the surroundings are specifically designed for them. The actions taken are repetitive, fixed, and in contact with a simple, defined set of objects.

Furthermore, even if some robots are able to integrate into semi-structured environments (for example, the robotic vacuum cleaners for homes or lawnmowers for gardens), they are specialized to a single task in a single scenario. Multi-purpose robots require a more human-compatible design and a higher degree of intelligent behavior [9], but versatile humanoid robots are still not pervasive in the current status of society.

In this study, we propose a computational model that is biologically inspired. Our approach consists in the use of mathematics and artificial intelligence to emulate human abstraction and adaptation capabilities in the execution of a series of primitive actions. We want to prove how demonstrating basic movements to a robot and composing them together with flexibility may lead to achieving complex tasks of various natures. Specifically, movement primitives are reused and

combined differently for different goals, avoiding explicit teaching of multiple objectives. The trajectories for the skills learned are adapted to the environment and partially composed thanks to the interpolation abilities of Conditional Neural Movement Primitives (CNMP) networks [53]. Lastly, the approach has been implemented and tested on an anthropomorphic robot and on an industrial collaborative robot.

1.2 Challenges

Robotics dominates many fields, but as discussed, often the environment is controlled, designed to help the robot in its task, and not human-friendly. If the purpose is to integrate robots into the human environment, robots must adapt to humans, not vice-versa. All environments in which humans are present are not organized or predictable, and this means one issue is that robots have to accommodate for these conditions. A challenge is definitely to introduce the machine to an unstructured environment, and this implies many sub-issues.

Having surroundings that might change forces the machines to have a great amount of perception. The system has to be extremely aware of the objects and people around it to operate in a safe and meaningful way. This translates into equipping many sensors and using real-time data from all available sources. Moreover, the machine cannot rely on these detection instruments mounted on the external world since a humanoid robot is expected to be mobile. Having a multipurpose system that can act in different scenarios implies, indeed, a self-contained arrangement of sensors.

The perception brings, in cascade, the necessity of storing this information and creating an internal copy of the surroundings that works as a base for planning and future predictions. Creating a digital twin for the environment is not essential for all the actions since some of them can be executed in real-time, but it is required to plan their effects and combine results together. For example, if a sponge is needed to clean a table, it would be faster to have the knowledge of its last position, but it can also be researched on demand and used while observing the effects in real-time till the table is clean. On the other hand, complex actions that combine multiple primitives need a future prediction of their effects on the environment, so its internal representation is required.

With changing surroundings, it is possible also that the expected position of objects is no longer consistent with the representation. This forces the system to find an alternative or explore the environment till the object is found. Other kinds of exploration possible are the exploration of the action space to infer new actions and results, or the exploration of objects' capabilities to learn new affordances and usages. [1]

Another factor worth taking into consideration is the subject of planning. Plans have to be structured in a meaningful way otherwise an incorrect sequence won't just produce an incorrect result but might bring the system further away from the final goal. The combinations of actions generated usually have importance in the order of execution, so the product of the skills has to be considered carefully.

Furthermore, objects and tools are usually designed for humans, so their capabilities might vary depending on the machine used and might influence the actions in the planning phase. Giving meanings to the objects, both in terms of affordances and representations, is still a tricky challenge in robotics [24] and partially involves the previously investigated challenges of planning and exploration.

Also, obstacle avoidance, whenever there is an object in the trajectory of movement, is a factor to take into consideration. The robot is required to be aware of the surroundings and itself, not to collide, hurt, damage them, or just fail the designated goal. Humans adapt previously known actions whenever an obstacle or an impediment is present.

Part of the adaptation challenge is also being able to transfer the skills known to new locations and scenarios. For example, learning how to turn a key for the door and use the action for the key of the car or the knob to turn on the stove. This is an essential capability that is difficult to implement in a machine.

Another more hidden challenge is how the actions are merged among them. Usually, humans, when they pass from one action to another, apply a smooth transition. This means that the movements don't have to fully start and end as they are learned, or the result will be artificial and sub-optimal.

Furthermore, object handling, grasping, and manipulation present some issues that are the object of research. How to pick the item desired, where, with which grasp, and with which force intensity are issues that can undermine the final result.

Lastly, another challenge that will be encountered is the recognition when the action is completed. Being aware of the right final state is essential for successfully matching the expectations for the goal requested.

These challenges discussed are crucial aspects to consider, but not all of them will be addressed in this project, and some will also be simplified. Nevertheless, it's worth noting the scope and limitations of this work and the boundaries within which the research operates.

1.3 Objectives

The aim of this research is to investigate novel skill generation by combining previously taught ones with the use of CNMPs [53]. The research aims to be applied to robotics scenarios involving trajectories for object manipulation and high-level goal achievement. The generation of new combinations of skills will be performed

by connecting skill segments that the robot learned by demonstration. The amount of demonstrations given should be reasonable for the system to be applied in real life by a human. The combination of actions will be investigated in both the concatenation of trajectories end-to-end and the use of parts of them. The ultimate goal is to create a system that allows a robot, given some demonstrations, to reuse the skills acquired to complete different objectives whose trajectories were never taught explicitly. Furthermore, the adaptation should be acceptable in different configurations of the environment and, ideally, in different scenarios.

1.4 Thesis Structure

Accordingly, the remainder of this thesis is structured as follows.

Chapter 2 discusses the background of the topic, the current advancements in the field, and the related research with a literature review.

In chapter 3 the instruments and frameworks used in this research are listed and analyzed to be able to understand the initial setup and replicate the results.

The chapter 4 explains the design and architecture of the proposed method. In order to understand the logic, the conceptual passages and mathematical background.

The chapter 5 analyzes the key points of the implemented solution through the explanation of the most important passages in the code developed.

The chapter 6 shows the final results and the testing on real-life robotic platforms.

Finally, Chapter 7 concludes this thesis by summarising its main contribution and future work.

Chapter 2

Related Work

In this chapter, we will proceed with the literature analysis and state-of-the-art methods related to the topic. Furthermore, some basic notions regarding the previous research relied upon will be explained.

Trajectory generation for robotics is a topic closely related to manipulation and navigation. Every movement can be described as a trajectory, defined as the composition of the path taken by an agent (or its joints) in time [4]. In this discussion, we will focus more on the manipulation part, being the most clear example of how combining different trajectories can produce novel robotic skills. Manipulation is one of the most distinctive capabilities of robots, since their main objective is to perform physical tasks in the real world. Any process that presents the necessity of a human to engage with a physical object meaningfully can solely be automated by robot manipulation. [47]

Creating robots capable of directly interacting with the world around them is still a key challenge in robotics, and manipulation is central to this. [33] Nevertheless, the ability to solve high-level goals in robots is increasing [14], [54] thanks to the recent advances in artificial intelligence. Some approaches may follow natural language instructions to achieve complex sequences of actions [16], but according to the research objective, a certain degree of autonomy is desired. This implies typically giving only the final goal and not the step-by-step instructions.

Recently, a lot of research has invested in deep reinforcement learning to map sensor inputs of a robot directly to motor torques [28]. These approaches provide independence, due to not hard-coded behaviors, and versatility by leveraging recent advancements in training deep networks. However, they encounter difficulties because of the demand for a huge amount of samples and underlying complexity. Reinforcement learning embraces the full complexity of these problems by requiring both interactive, sequential prediction as in "learning from demonstration" (LfD) and complex reward structures with only "bandit" style feedback on the actions chosen [30]. For this reason, recent research aims to minimize what must

be learned and to support sequential composition [47]. However, the collection of trajectory samples in the real world requires too much time, so a framework for robotic simulation is often used to simplify the setting.

A commonly used technique in robotics is Learning from Demonstration (LfD) [3] [50], and then [43]. It allows for solving a wide variety of robotics problems by imitating an external agent. The demonstrator, often a human or another system, provides examples (expert demonstrations) of how to perform a task, and the learning agent generalizes from these demonstrations to acquire the ability to perform the task later independently.

Famous learning from demonstration research includes statistical modeling [5], dynamic systems [51], and their union in [56]. In Dynamic Movement Primitives (DMPs) [51] [22], a trajectory is represented with a set of differential equations and learned with as little as one shot. DMPs demonstrated a good capability in learning a remarkable variety of dynamic behaviors [36]. Thanks to the "point attractor" mechanism, it guarantees reaching a point even under perturbations. DMPs have successfully been utilized in difficult manipulation tasks such as in-hand manipulation and flipping boxes using chopsticks [40]. On the other hand, additional tuning is needed to determine the number of basis functions. Moreover, the motor learning problems that are most intriguing often involve a high number of dimensions [31], and DMPs still struggle to be integrated with high-dimensional, multi-modal data [49]. Finally, their approach is not designed to learn from multiple trajectories and, therefore, cannot encode the important parts of multiple demonstrations [53].

In the Probabilistic Movement Primitives (ProMP) [39], instead, the distribution of basis functions is often represented using a probabilistic framework, typically a Gaussian Mixture Model (GMM) or a similar probabilistic model. Probabilistic models allow to capture the variability across demonstrations and different Degrees of Freedom and, furthermore, enable variable conditioning in which predictions can be refined through new observations.

Historically, Gaussian Mixture Models [37] have been prominent among various probabilistic approaches since they provide adaptable solutions to the challenge of modeling trajectories. On the other hand, GMMs involve estimating many parameters, especially when dealing with high-dimensional data or a large number of components. This can make training and inference computationally expensive, particularly when the dataset is extensive, and if not done correctly, can lead to some failures, Figure 2.1.

Another probabilistic model, like GMMs, often used in statistical modeling techniques is Hidden Markov Models (HMMs) [34]. HMMs were successfully applied to learn multi-modal models from temperature, pressure, and fingertip information for exploratory object classification tasks [6].

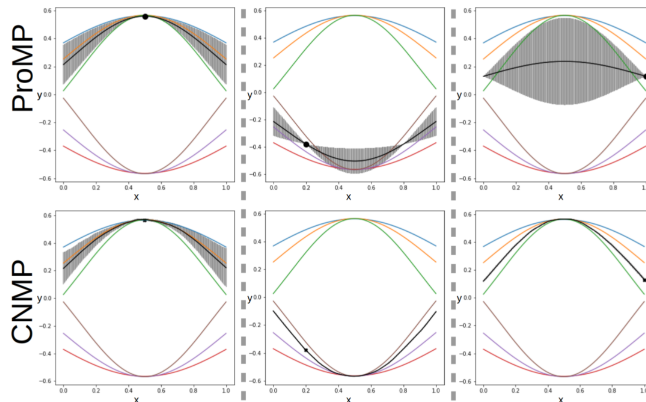

Figure 2.1: In ProMPs the distribution of basis functions is often represented using a probabilistic framework, but can lead to some failures.

A recent model developed in robotics called Conditional Neural Movement Primitives (CNMP) [53] also learns from demonstrations, but instead of using GMMs, they use neural networks to model the mapping from conditions to trajectories directly. Neural networks allow the model to scale better and offer robust data approximation via gradient descent. Until recently, neural networks in deep learning were trained to approximate a single-output function. However, when data is a distribution, the single function cannot approximate the underlying model. So, the network can be modeled as a probabilistic approximator that can predict the distribution parameters, mean, and variance. This makes CNMPs well-suited for tasks with complex, high-dimensional state spaces. It allows one to learn skills in tens, rather than thousands, of real-world interactions and interpolate among them.

Based on the above-mentioned observations, the proposal is as follows. In the following research, the ability of CNMPs to interpolate the trajectories demonstrated is exploited to synthesize new complex skills. The model is based on Gaussian Processes (GP) [52], Neural Processes (NPs) [12], and Conditional Neural Processes (CNPs) [11]. For context, an explanation in detail of these above-mentioned methods will follow.

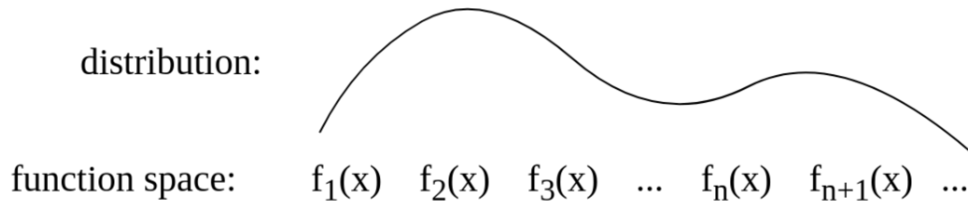

Figure 2.2: Gaussian Processes are probabilistic models that define a distribution over functions.

2.1 Gaussian Processes

Gaussian Processes [52] are probabilistic models that define a distribution over functions, Figure 2.2. This means that they leverage pre-existing knowledge about a set of functions and infer during test-time specific functions that fit the data provided. Given a set of observed points, there are infinite possible functions that pass through them.

Gaussian processes provide an elegant solution to this challenge by assigning a probability to each of these potential functions. The mean of this probability distribution then represents the most probable characterization of the data given the observation points [23].

This is called regression and is used, for example, in robotics or time series forecasting. Gaussian processes are not limited to regression, and they can also be extended to classification and clustering tasks [26] [29]. Many supervised learning problems can be seen as function approximations since a dataset of observations $\{x_i, y_i\}_{i=0}^{n-1}$ is basically a number n of evaluations $y_i = f(x_i)$ of an unknown function f. A supervised learning algorithm returns an approximated function g. The goal is to minimize the loss between the real function f and the predicted one g. The evaluation is carried out on unlabelled data points x_j.

On the other hand, the disadvantages of Gaussian Processes are prior selection and training time for large datasets. Scaling issues with GPs have been addressed in [55]. The limited expressivity from functional restriction was addressed with DeepGPs in [7] [48].

Overcoming these issues and attempting to combine Deep Learning (DL) with GPs was proposed in [59], but the approach remains close to GPs since the network is used to learn more expressive kernels to use with GPs.

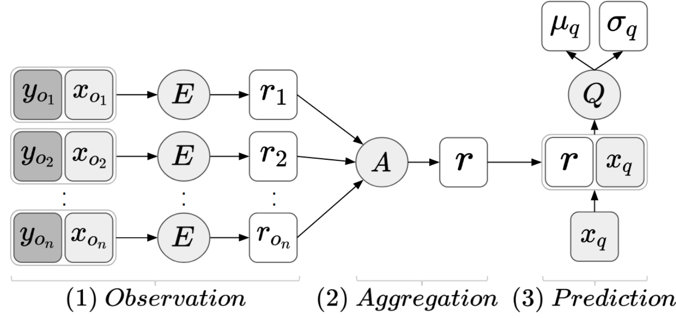

Figure 2.3: The structure of a CNP network, with three main blocks: Encoder, Aggregator, and Decoder.

2.2 Conditional Neural Processes

In [11], the authors propose a novel research in which the inference potential of Gaussian Processes and the performance of neural networks are blended together. Neural networks are extensively employed as approximators of functions and have demonstrated considerable efficacy but often require large datasets for training. In CNPs, the prior knowledge is directly derived from the data, allowing them to infer the underlying function distribution based on observations. CNPs are built with three main blocks: Encoder, Aggregator, and Decoder. Encoder E and Decoder Q are typically Multi-layer perceptrons. The model structure is shown in Figure 2.3. The model scales with complexity $O(n+m)$ for making m predictions from n observations, while GPs scale with $O(n+m)^3$. CNPs don't require the specification of a kernel cause they learn it from the data provided in training. The tradeoff is that the representations of the observations have fixed dimensionality.

The work is based on the previous research of Neural Processes (NPs) [12]. NPs are suggested as a means to manage the substantial computational demands of GPs while leveraging their flexibility and efficiency with data. NPs help create different predictions by learning a shared hidden representation. However, they have trouble with long sequences because they automatically pick certain points. Building on NPs, Conditional Neural Processes (CNPs) are strong models that make training more efficient by allowing explicit conditioning. The approach allows precise predictions for targets sampled from a distribution conditioned with observations, Figure 2.4. Given a varying number of observations O, a neural network E is utilized as an encoder to generate a fixed-size representation r_i.

Observations fed to the network don't have an order, following the stochastic processes, because subsequently, they are aggregated with the average operation to obtain a single representation r. Any commutative operation is valid and usable. The resulting representation r contains the conditioning information and is fed

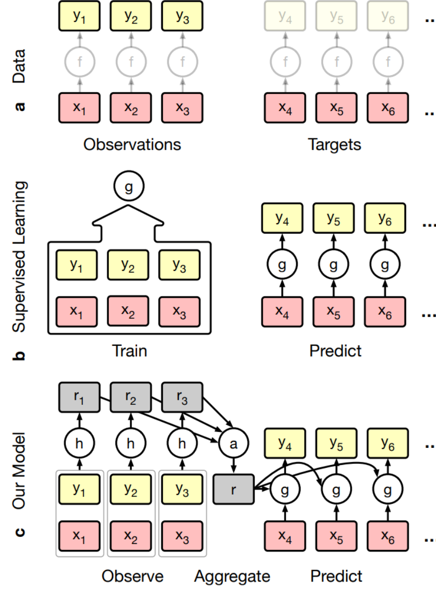

Figure 2.4: CNP allows precise predictions for targets sampled from a distribution conditioned with observations

to a decoder network Q along with the desired target x_j to query. The decoder network Q has parameters θ. For all the targets $x_j \in T$ the decoder outputs the mean and standard deviation.

The formulation of the encoding of each observation is:

$$r_i = E_\phi(x_i, y_i), \quad \forall (x_i, y_i) \in O \tag{2.1}$$

and the following commutative operation between the encodings to create a single one:

$$r = r_1 \oplus r_2 \oplus ... \oplus r_i, \tag{2.2}$$

The commutative operation expressed by $\oplus$, can be summation, average, product, and so on.

The vector generated is concatenated with the target variables. The merged representation is passed to the decoder to obtain the output as:

$$\phi_j = Q_\theta(x_j, r), \quad \forall x_j \in T \tag{2.3}$$

Where the output is:

$$\phi_j = (\mu_j, \sigma_j^2) \tag{2.4}$$

which are the mean and the standard deviation of the output variable.

In summary, the CNP model, with averaging operation, can be formulated as:

$$\mu_j, \sigma_j^2 = Q_\theta \left(x_j \oplus \frac{\sum_i^n E\phi((x_i, y_i))}{n} \right) \tag{2.5}$$

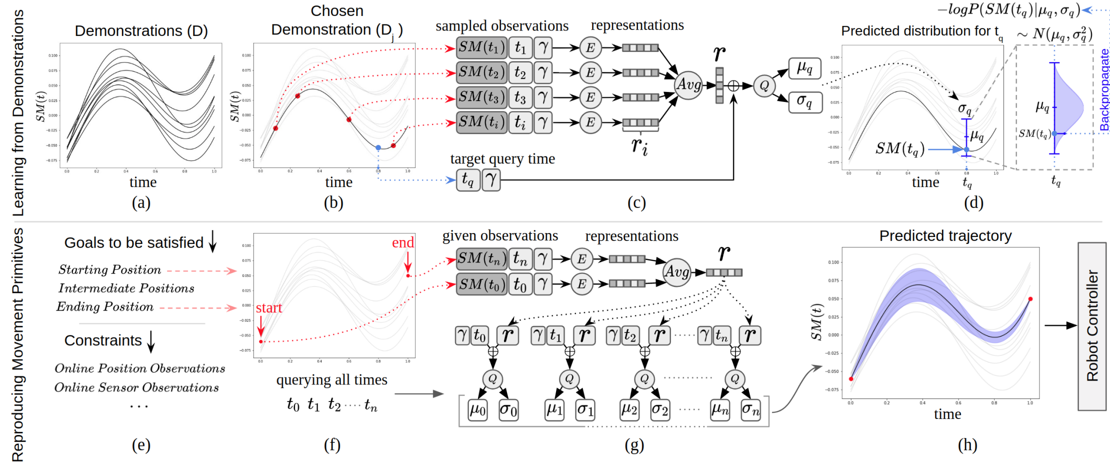

Figure 2.5: CNMP model, it is conceived to work with temporal relations between sensorimotor data and different task parameters

2.3 Conditional Neural Movement Primitives

Finally, in [53], Conditional Neural Movement Primitives are proposed as a model. CNMPs, as the name suggests, are an extension of CNPs and are particularly well suited for the robotics domain. The model illustration can be seen in Figure 2.5, from [53]. The "learning from demonstration" framework can learn non-linear relationships between trajectories and reproduce them in joint or task space. In this case, a trajectory is formally defined as a temporal function, $\tau = \tau(t)$, where the sensorimotor data in time describes how a robot moves. So, each trajectory τ is a list of ordered sensorimotor values:

$$\tau = \{SM(t_1), SM(t_2), ..., SM(t_T)\} \tag{2.6}$$

where $SM(t_i)$ is the sensorimotor data at an instant of time t_i. So, the challenge of trajectory generation becomes figuring out a series of commands $SM(t_i)$ that creates the movement desired [13]. Finally, with a set of observations O, the model has to learn the function $\tau = f(t|O)$, using N expert demonstrations, $D = \{\tau_1, \tau_2, ..., \tau_N\}$.

CNMPs are conceived to work with temporal relations t and different task parameters γ. CNMPs maintain the permutation invariance of CNPs over observations O and queries T. Furthermore, to make the model time-invariant, the sensorimotor trajectories are often scaled in the interval [0,1]. The task parameter γ effectively adds one or more dimensionalities to the network's input, and it's passed to both the encoder and the decoder. An observation becomes the concatenation of $SM(t_i)$, t_i, and γ. The dimensionality of $SM(t_i)$ depends on

factors like the Degrees of Freedom of the robot joints and the number of variables corresponding to the actuators. For the aggregation of the representations, the averaging operation has been chosen. During training, a random trajectory τ_i is selected from the D set of expert demonstrations. Next, a random number n of random observation points are selected from the trajectory τ_i. The encoder takes the n observations and produces n representations. The final representation is obtained by averaging the representations produced by the encoder fed with all the observation points. The target data is predicted using the representation and the query time t concatenated to the task parameter γ

The encoder and the decoder are trained jointly with the error calculated from the following loss function:

$$L(\theta, \phi) = -logP(y_i|\mu_i, softmax(\sigma_j)) \tag{2.7}$$

using both mean and standard deviation produced by the network. As a note, the uncertainty of the prediction provided by the variance is useful for the model's active exploration to choose wisely where the next observations are needed. Moreover, the capacity of CNMPs to deal with high-dimensionality input can also be used to input images in the model. Image completion indeed can be seen as a regression task. Leveraging the interpolation capabilities of CNMPs, our approach will investigate novel synthesis by combining and concatenating previously taught ones.

Chapter 3

Platforms

In this chapter, all the physical and digital platforms utilized will be explained to give a proper understanding of the initial architecture and a more comprehensive idea of the environment of the experiments. The first part will state the devices and their setup, capabilities, and configuration used. Subsequently, the frameworks and libraries employed will be listed and described.

3.1 Baxter Robot

The Baxter robot is an industrial robot built by Rethink Robotics in 2011 [58]. The platform [Fig. 3.1] has two robotic arms with interchangeable grippers at the wrist (End Effectors). The robot is 180cm tall and, with its pedestal, weighs 140 kg. The arms have 7 degrees of freedom (DOF), which implies they have seven joints each. This makes it kinematic redundant, meaning that for some points reached in space, multiple pose configurations of the arms are possible. These two factors combined allow the robot to have an impressive capability in manipulation. The robot can be equipped with suction caps or two-jaw parallel grippers; we chose the last ones in our configuration. The gripper, in addition to its position, also offers information regarding the force applied while grasping an object.

The robot was designed with attention to collaborative tasks with humans. For this reason, it eases teaching by demonstration by having integrated two touch sensors in the wrists that unlock the motors of the arms, allowing the user to move them easily and record the trajectories executed. The robot helps the movement with a feature called "Zero-g mode", in which the weight of the joints is neutralized actively by the motors. This enables the teaching expert to demonstrate movements in a similar environment without gravity, without having to carry the instrumentation load in time constantly. Furthermore, the robot has many input buttons and LEDs, they are present on the hands, arms, and chest, and they allow

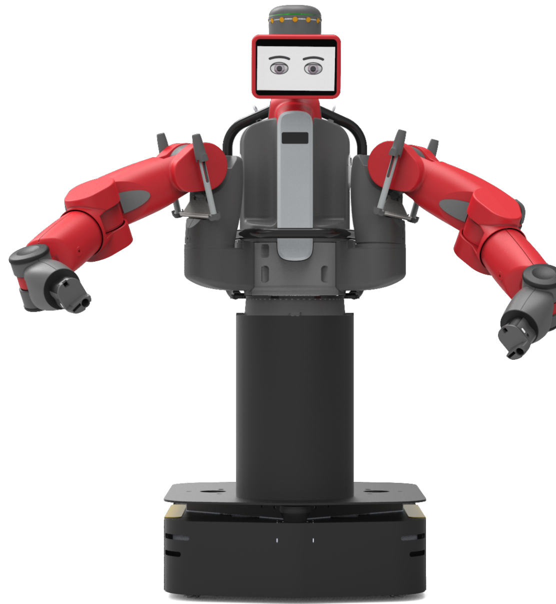

Figure 3.1: Baxter Robot platform

the programming of custom behaviors. They are especially useful in retrieving and giving instructions to the robot without reaching a computer, like closing the gripper at the desired moment, getting feedback, or starting the trajectory recordings.

Another feature worth mentioning is the increased safety of operating around humans. Thanks to active and passive safety systems equipped in the platform, it doesn't require a cage for protection. On the other hand, making the robot less hazardous comes with the cost of precision. A motor driving a spring that drives Baxter's arm instead of just a direct motor impacts the precision of movements, sometimes in terms of centimeters. This doesn't make the robot perfectly suitable for industrial applications, but especially appropriate for research and for the adaptability in our project.

The head of the robot includes a ring of sonar sensors for people detection, a wide-angle camera, and a movable display that acts like a face. Another benefit of the robot lies at the end of both hands. Immediately next to the attachment for the tools, an infrared (IR) sensor provides data on the distance from a solid object (i.e., a table) and an inertial measurement unit (IMU). Moreover, an embedded RGB camera is also present, allowing to see closely the object approached or to change the point of view on it without additional external cameras.

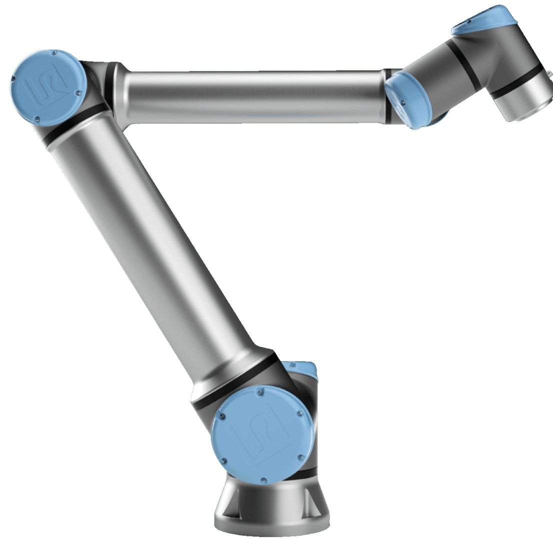

Figure 3.2: UR10 Robot platform

3.2 UR10 Robot

UR10 is a single industrial robot arm that shines in reliability and precision [Fig. 3.2]. It has been manufactured by Universal Robots and combines long reach with a high payload [57]. It is intended for medium-duty tasks, so it's compact in its overall dimensions compared to a fully intended industrial robot. It can reach an impressive height of 2.3m on its pedestal. In our experiments, it was mounted on a pedestal 0.85m tall to increase the reachability at table level.

The arm has a reaching radius of 1,3m from the mounting point, which implies a workspace of approximately 5,3 square meters at the base level. The robot has 6 Degrees of Freedom, with six rotating joints. It is able to reach any point in its reaching radius but has no kinematics redundancy, meaning only one position is possible for any given point. The total payload that can be carried is 10 kg.

Like the previous robot, this one is designed to work collaboratively with humans. It features built-in safety features, such as force/torque sensors, to detect and respond to external forces or unexpected events. A button to release the motor breaks is present on the floating touch screen and allows the robot's motion by hand. This robot also doesn't require a safety cage around for protection, but the emergency stop button always has to be within easy reach.

The robot design emphasizes modularity, making it easier for users to customize and adapt it for different tasks or use various end effectors. We coupled it with a three-finger gripper described in the next paragraph.

Figure 3.3: 3F Robotiq Gripper platform

3.3 3F Robotiq Gripper

At the end of the UR10 Robotic Arm, a Robotiq 3-Finger Adaptive Robot Gripper was mounted [Fig. 3.3]. The gripper has a human-inspired design and has three fingers with three joints each [44]. The physical platform was chosen for its precision and safety, and it pairs well with the UR10 capabilities. The gripper offers different grip modes; the ones available are: "basic", "wide", "scissor" and "pinch". Each is appropriate for distinctive objects to grip; the basic one is the most versatile, but the wide one has more stability for big or long objects, and the pinch one is the best for small objects. The "scissor mode" closes together the two fingers on the same side, for high-precision manipulation. We mainly used the "pinch" setting.

The gripper has a mass of 2.3 kg in contrast with a grip payload of 10 kg. The grip force applied can range from 30 to 70 N, depending on the grip mode selected. The precision declared is up to 0.05 mm.

This platform was designed as well for collaborative robotic applications, allowing it to work safely alongside human operators. It incorporates safety features to detect and respond to external forces, stopping in case of high forces applied. The torque and speed of gripping data are available and exposed through a dedicated ROS topic. Speed and torque are also adjustable for the intended use. It is possible to control and retrieve data for each finger individually.

Figure 3.4: FT 300-S Force Torque Sensor

3.4 FT 300-S Force Torque Sensor

Between the UR10 robot and the 3F Finger Gripper, an FT 300-S Force Torque Sensor [Fig. 3.4] was mounted to increase precision and repeatability.

The sensor offers high-resolution real-time measurements regarding the force and torque applied to the three space dimensions and improves the capabilities of the robot. This device makes the UR10 able to detect the payload carried or the amount of pressure between the object or gripper and the static environment (i.e., the table).

The device was built for compatibility with the Universal Robot series and has an IP65 rating. It also enables precise object placement such as alignment, indexing, and insertion [45].

It exposes the six readings of the forces and torques in the three axes through a port opened in the computer connected to it. In our setup, it has been connected straight to the UR10 computer, allowing any user to retrieve the necessary data.

The FT 300-S is commonly used in tasks where force and torque sensing are critical, but we used it to increase the reliability of the payload measurements and the safety of the operations.

3.5 Frameworks

ROS The Robot Operating System (ROS) [46] is a framework to standardize the deployment of robot applications. The system is a set of software libraries and tools that combine the state-of-the-art drivers for the most common robot interfaces and contain the most used algorithms for robotics.

Since robotics programming is a complex challenge, the idea behind ROS is to use the "divide et impera" approach and split it into multiple sub-problems. This division requires a distributed strategy, and distribution implies communication among parts. One of ROS's main objectives is to standardize communication. For this reason, ROS acts like a middleware framework, allowing the ease of the dialog between software and the robotic hardware. It is widely used in research and industry, from mobile robots to manipulators. In our research, we used ROS version 1.

It's platform-independent and open source, so it is possible to create a custom robotics device compatible with it, and it's possible to develop personalized libraries. Its modular architecture allows the creation or use of a series of executable pieces of code called nodes, which run on a single or even on many computers. The nodes communicate among themselves with messages in the form of data structures previously defined. The distributed system allows to decuple the computation of heavy tasks, like vision, 3D reconstruction, and navigation, from the robot's hardware.

Communication occurs through "topics" and "services". Nodes running on any computer ping the "master node" designated to retrieve all the possible topics and services exposed from other nodes in the network. A node can be a "publisher" or "subscriber" to a topic, sending messages to it or receiving messages from it. Communication with topics is not blocking and it is many-to-many: multiple nodes can publish, and simultaneously, multiple nodes can subscribe to a topic. Services work in a similar way, but the communication blocks the computation till the data requested is retrieved. Their mechanism is analogous to Remote Procedure Calls (RPCs).

ROS is available with two commonly used programming languages: Python and C++. We used the Python version with the "rospy" package in the experiments with both robots.

Pytorch Pytorch is a Deep Learning framework [41] that focuses on speed and usability with an imperative and Pythonic programming style. The Python library [42] offers a wide variety of models and building blocks for constructing neural networks. By design, it eases the debugging for the user with a rich ecosystem of dedicated tools. It works on the CPU and on hardware accelerators like GPUs. For this reason, PyTorch provides a multi-dimensional array called a tensor, which

is similar to NumPy arrays. Conversions among both of them will be present in the code implementation.

Jupyter Notebook Jupyter Notebook is an open-source interactive web application [25]. It supports multiple programming languages and offers a cell-based environment where code and description/graphical results can be blended. Jupyter Notebook integrates seamlessly with popular Python libraries, such as NumPy, Pandas, Matplotlib, and scikit-learn; some of them will be described later. It was used occasionally in our experiments to provide a fast and interactive coding experience with Python. The notebooks can be easily shared, and the process is clearly visualized. It was specifically useful in plotting multiple graphs during the training stages of neural networks or debugging operations on multi-dimensional arrays.

Anaconda and Python Libraries Anaconda is an open-source software that contains open-source tools and packages for data science, machine learning, and scientific computing [2]. It has been used to track the packages utilized in the robotic platforms and in the models' development and training. The Conda package manager was the most used tool to easily install, update, and manage various software packages and dependencies. Some of the most important libraries are listed below.

MatplotLib Matplotlib is a comprehensive 2D plotting library for Python that generates high-quality charts, plots, and visualizations. It has been widely used to double-check the quality of the training or plot the trajectories recorded with the robots.

Numpy NumPy is a package for numerical computing in Python. It supports large, multi-dimensional arrays and matrices and a collection of mathematical functions to operate on these arrays. It has been used for array slicing, normalization, and smoothing data.

Chapter 4

Design

In the previous chapters, it has been presented the necessary knowledge about the background research (chapter 2) and the platforms utilized in this work (chapter 3). This chapter discusses the conducted research from the design perspective, giving the main points and emphasizing the key ideas without going into the implementation details as the next chapter (chapter 5).

As analyzed in the Intro (chapter 1), humans have remarkable cognitive flexibility and use it to achieve complex goals across various daily scenarios. These goals are often broken down into smaller sub-tasks (skills), and this is one objective of this research. This combination of trajectories also requires an appropriate shift among them, in both cases, the combination being end-to-end or partial.

Being biologically inspired, this research aims to emulate this human flexibility in skill execution and task composition with the previously explained CNMP networks. Movement primitives that define skills are combined differently for different goals and are adapted to the environment and context thanks to the interpolation abilities of CNMP networks.

The following discussion will proceed with analyzing before the partial skill composition with CNMPs, modeling how it is possible to go from one movement primitive to the other. The motivation of this order is that this part enables the partial combination and is used as well in the transition moment of end-to-end skill concatenation.

Next, it will be analyzed how it is possible to concatenate them one after the other to achieve meaningful execution in order to reach the final goal position or even goal-state.

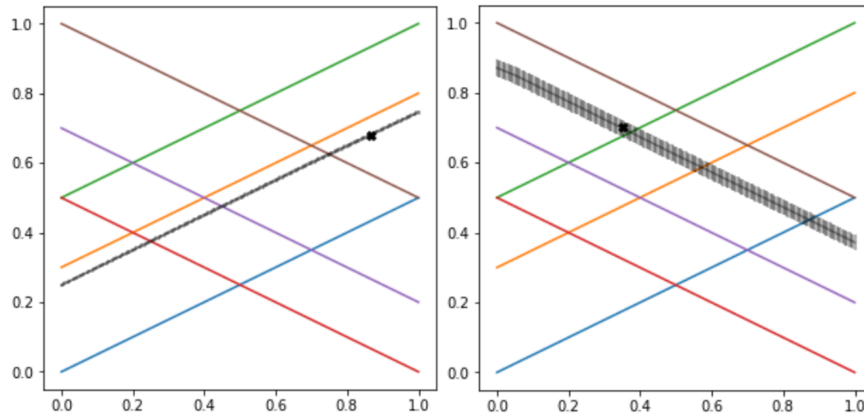

Figure 4.1: Trajectories for two different skills demonstrated, in different colors. The generated ones in grey color, from the black conditioning points.

4.1 Partial Skill Combination

In order to synthesize a new skill by combining parts of others, it is required to teach the model at least two different types of movement primitives. Subsequently, the robot has to be able to pass from one action (skill) to the other.

The moment of this transition has to be arbitrarily decided according to the necessities and not specifically crafted at teaching time.

The challenge of this apparently straightforward problem resides in the moment of the transition, because it has to be executed in a meaningful, natural, and safe way. These simple three requirements will lead the following examination through different approaches till the one designed.

For the purpose of a better understanding of the design and its steps, from now on, a simplistic example (fig. 4.1) will be used to further clarify the explanations given. Nevertheless, the simplification doesn't exclude the possibility of more elaborate tasks that require complex movements or sensorimotor data.

The two actions discussed will be a simple movement upwards and a similar movement downwards. They can represent, for example, a trajectory in real life, a manipulator movement, or a joint trajectory. This allows a discussion that starts from a single 1D dimension and keeps the understanding manageable at the subsequent introduction of new dimensions.

Some demonstration trajectories are given for both skills, in the fig. 4.1 are visible as the three colorful thinner lines ascending and three colorful thinner lines descending. The two triplets differ to some extent, so the network has enough knowledge to create as well new trajectories never seen. This will prove later that the method designed also works with newly generated movement primitives.

The thicker points in the fig. 4.1 are the conditions, these force the model to

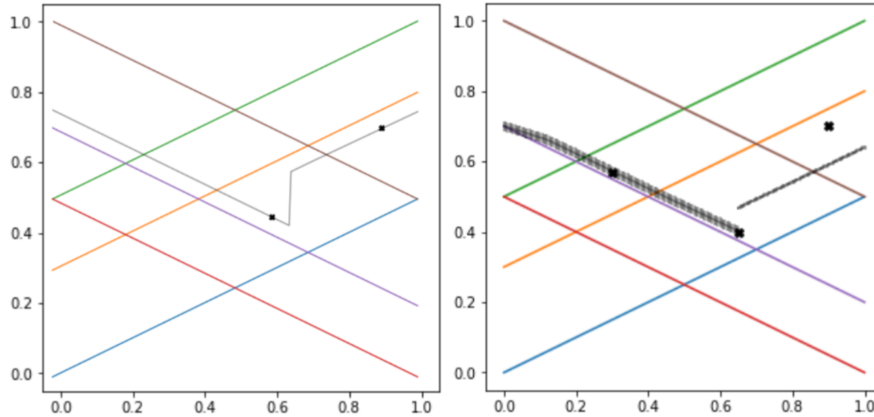

Figure 4.2: Examples of failing approaches with stitching in CNMPs. On the left, simple combination of parts. On the right, stitching the right part using a conditioning point at the end of the left part.

generate a trajectory that passes through that state. The movement primitive created is denoted as the grey line passing through the condition point, along with the uncertainty of the prediction in every timestep as its width.

The shift can not be abrupt, so it's not sufficient to directly stitch together two parts of the trajectories collected. The simplest solution of executing one trajectory till a certain desired timestep and executing the second one after that moment will create an abrupt jump in the execution for the majority of timesteps where the trajectories don't perfectly intersect (fig. 4.2). A jump in the movement primitive will lead to an unnatural fast change of pace and position of the robot during the execution, which is not clearly referable to human behavior. Moreover, moving from one position to a completely different one in the next moment is an unsafe behavior. It might lead to damage to the robot itself and its surroundings, harm to people, or activate the safety stop measures of the robot due to the high speed and torque applied.

Being able through the CNMP model to generate trajectories from previous demonstrations allows a certain flexibility, so a second approach might suggest generating a second trajectory starting from the end points of the first one. This solution only delays the problem subsequently because the condition point(s) on the second trajectory will be dependent on the first one, requiring further calculation, and won't be solely used for the effective purpose of making a trajectory reach the desired state. In fig. 4.2, right plot, the first part of the trajectory is generated with the left-most condition ($time = 0.3$). Subsequently, the second part is generated using a conditioning point at the end of the left part plus the desired one. These two conditioning points will likely not be on the same trajectory gener-

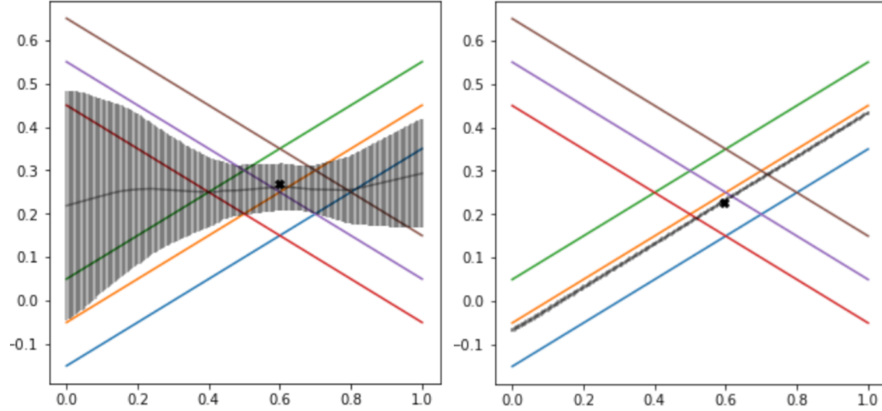

Figure 4.3: CNMP modeling uncertainty for a condition. In the parametrized CNMP, uncertainty is solved, and skills are encoded with task parameters 1 for ascending and 2 for descending.

ated: let's recall the robot arms generally have 6 or 7 DoF, chapter 3, and even the cartesian space has three dimensions plus the 4D quaternion for the orientation, so the probability of trajectories intersecting in spaces of at least six dimensions is minimal. This will lead the CNMP model to average the two trajectories obtained by the two conditions on different points. In fig. 4.2, right plot, we see the second trajectory part passing in the middle of the two right-most observations that would generate independently two different movement primitives. To pass from one to another of these last two trajectories, a shift will still be required, and this raises again the same problem. So, even using conditioning points, there is a tradeoff between jumping and not meeting the conditions for the second trajectory.

Since the stitching of parts is not a feasible option, the approach selected implies giving the same network the two skills and obtaining a coherent output.

The CNMP model is capable of storing different movement primitives of different skills without requiring multiple networks. The selection of the right trajectory is due to the conditioning points (observations) previously explained. The observations are indeed useful for both finding the interpolated trajectory from the demonstrations passing in a new state and also for identifying the correct skill. In fig. 4.1, the conditioning points identified uniquely the trajectory to generate. However, as it happens in fig. 4.3, the conditions might be uncertain, so the standard CNMP model averages the outputs, creating a misleading trajectory.

To eliminate uncertainty, a task parameter γ (TP) can be included in both the input and the query of the network, as in fig. 2.5. The task parameter is a full-fledged new dimension of the network, like input time t and the output value. In the case of the task parameters, the dimension added is an input dimension,

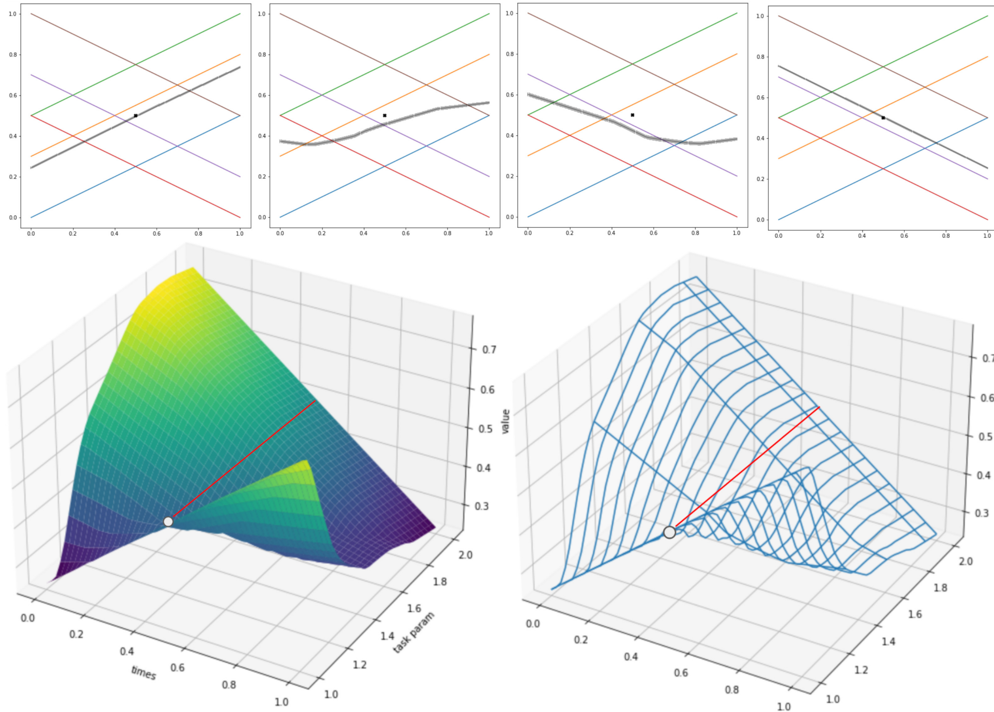

Figure 4.4: Plot of the interpolation abilities of CNMPs also in the task-parameter dimension.

which excludes uncertainty. Now, the network, given a condition, can uniquely identify the movement primitive.

How adding a dimension to the input is interpreted by the networks and what's in between the two parameters has been researched to understand better how to shift among them. In order to do this, a 3D plot is required. This is achieved by leveraging the previously considered weakness of the model of having uncertain conditions. These points are shared by parameters among and maintain the same other dimensions while the external parameter γ varies. This doesn't introduce any bias while the condition changes.

The visualization of a continuous change between different task parameters is present in fig. 4.4. As it's possible to observe, the network interpolates nicely among the functions also in the tasks-parameter space. The movement of the conditioning point in time is depicted in the 3D plot with a red line ending in its final position.

It is worth noting that to obtain the graph in fig. 4.4, both task parameters in the observations and in the query had to vary together. As visible in fig. 2.5, the original network design of CNMPs implies the presence of this parameter (as γ) for the conditioning points and for the query t.

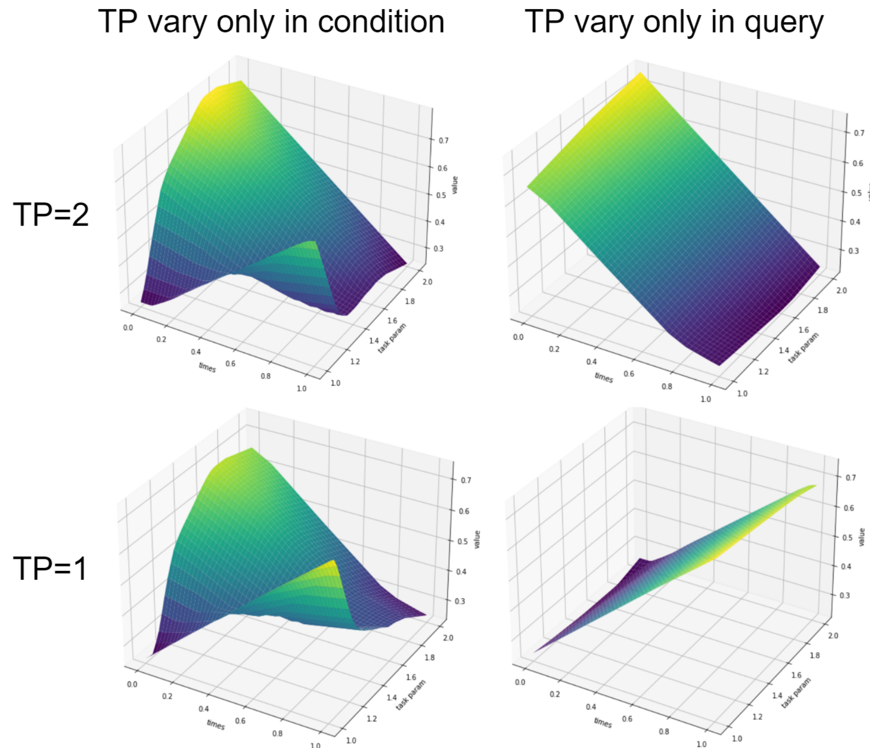

Figure 4.5: Influence of task parameter from the conditioning points and from the queries.

Successively, the difference in variating only one at a time has been researched. In the first place, for all the time steps, only the task parameter of the condition was changed, maintaining the task parameter of the query constant. Next, the opposite was performed to understand how they influence the network result differently.

The plots comparison is visible in fig. 4.5. On the top row, for task parameter 2, the first graph shows how the result changes according to the variation of the task parameter in the condition. Meanwhile, the second graph shows how, for every time queried, keeping the parameter fixed to 2 in the conditions and changing it in the query doesn't produce a significant variation in the output. The second row repeats the procedure with the other task parameter to crosscheck the results. It is clearly emerging how the parameter in the observation seems to have a stronger influence than the one in the query, which doesn't seem to contribute significantly.

The following subsections will investigate the structure of the network and the design of two different architectures. This work proposes before a network with task parameters only in condition and, next, the model with task parameters only in the query. These two architectures are respectively built and examined below.

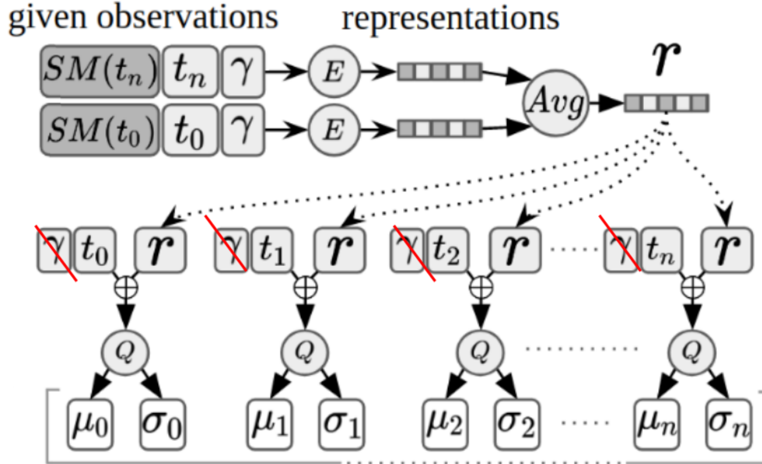

Figure 4.6: Architecture of the CNMP model proposed without the original task parameter in the queries.

4.1.1 CNMP model with task-parameter only in condition

This part investigates the architecture of the CNMP model altered to be able to infer the results having the task parameters (TPs) only in the input of the conditions. This design allows querying afterward the network only with the time t. This architectural choice is motivated by the previous analysis in which the task parameter in the query seemed to have little if no importance for the results. This also feels naturally more sensible for constant tasks as the request is only for the value at a time step, and the task parameter would remain constant anyway.

In fig. 4.6, the design changes can be compared to the original model. The neural network now feeds the decoder only with the time value to query and the constant representation of the conditions. As a result, the decoder has an input dimension less than the original model.

Table 4.1: Comparison Table of errors of original CNMP vs CNMP with TP only in condition

MSE Error	TP	CNMP	CNMP TP condition
on demonstrated trajectory	1	2.200240773631327e-06	0.0005239668753240399
	2	2.852831969051669e-08	3.317327413240996e-05
on interpolated trajectory	1	2.8834989373212024e-06	0.0004749418782444545
	2	2.598636844065303e-07	3.6395583963202646e-06

The new model has been trained on the same dataset and verified with the same validation set.

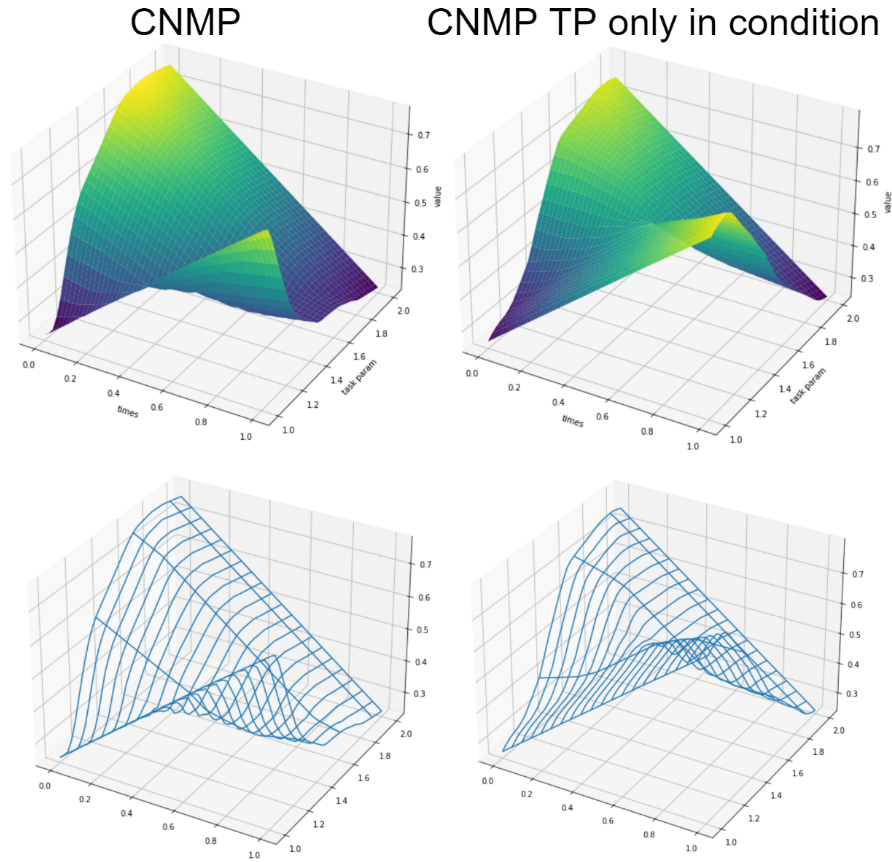

Figure 4.7: Interpolation comparison of CNMP model vs CNMP model with TP only in the condition.

The quantitative comparison results in table 4.1 show the average mean square errors (MSE) for every task on demonstrated trajectories and interpolated trajectories. The performance of the modified architecture exhibits an increase in the errors that, while present, is not significantly detrimental, suggesting a promising level of robustness in its overall functionality.

The qualitative results of the interpolation can be seen in fig. 4.7, and although some interpolation differences are visible, they still clearly maintain a sufficient degree of correctness.

The new network architecture maintains the same interpolation capabilities, but allows the user to query only the time, without worrying about the task parameter, which will be inferred constant from the observations.

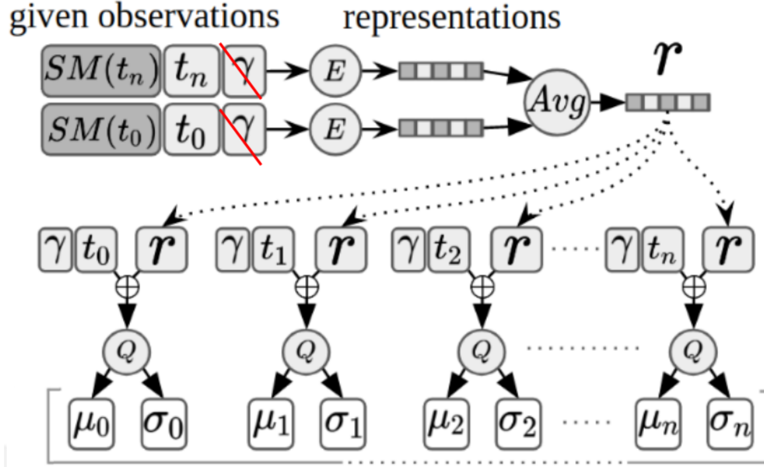

Figure 4.8: Architecture of the CNMP model proposed without the original task parameter in the observations.

4.1.2 CNMP model with task-parameter only in query

This subsection analyses the opposite alternative to the previous investigation. The architecture of the CNMP model is altered to generate the results using the task parameters (TPs) only in the input of the query.

This design allows having conditioning points that are parameter-less and querying the network subsequently with the time t and the task parameter γ.

This architectural choice seems to be more appropriate for possible changes at run-time of skill by the model. However, it's clearly more challenging since the information is provided later in the pipeline.

Moreover, conditioning points are responsible for the final position at every time t, and feeding the network only subsequently with the task forces it to infer the γ of the conditions.

In fig. 4.8, the design changes can be compared to the original model. The neural network now feeds the encoder with observations that don't have any parameter, delaying the γ inference to the decoder. As a result, the encoder has an input dimension less than the original model.

The second new model has been trained on the same dataset and verified with the same validation set as the previously discussed ones.

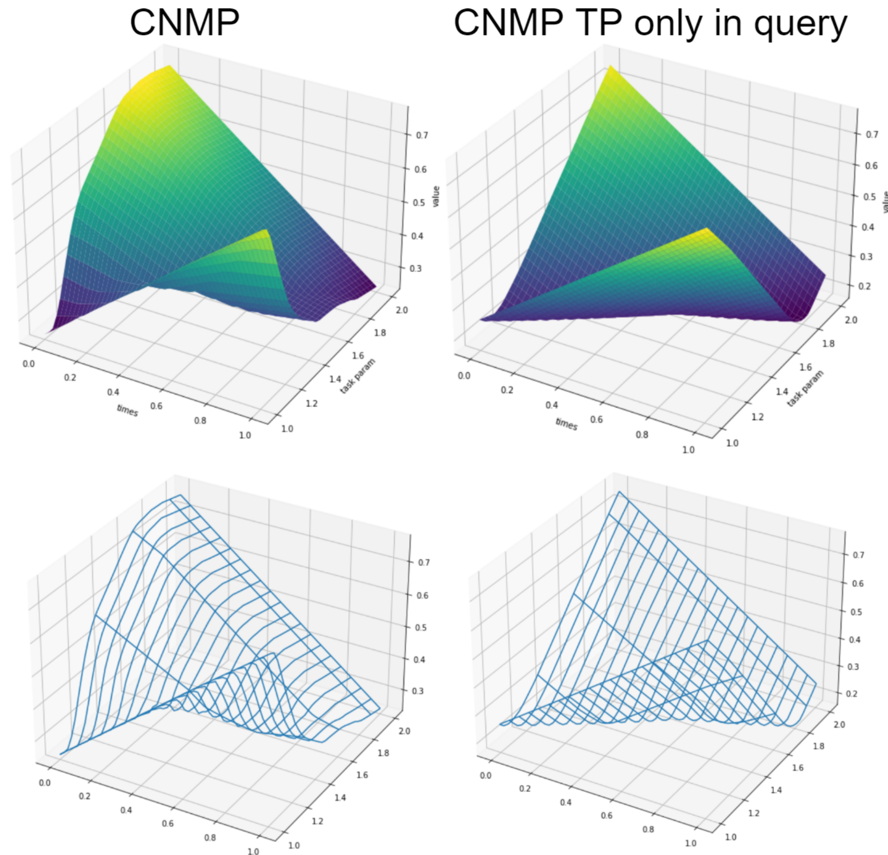

Figure 4.9: Interpolation comparison of CNMP model vs CNMP model with TP only in the query.

Table 4.2: Comparison Table of errors of original CNMP vs CNMP with TP only in query

MSE Error	TP	CNMP	CNMP TP query
on demonstrated trajectory	1	2.200240773631327e-06	0.00015134105003480186
	2	2.852831969051669e-08	1.5758800492775196e-05
on interpolated trajectory	1	2.8834989373212024e-06	0.0008166800702431865
	2	2.598636844065303e-07	0.00020075239802309004

The quantitative comparison results in table 4.2 show the average mean square errors (MSE) for every task on demonstrated trajectories and interpolated trajectories. The modified architecture again demonstrates a rise in errors compared to the original model, but these errors, though now noticeable, do not compromise the usability of the model. Not the same can be said for the qualitative results

of the interpolation. The comparison that can be seen in fig. 4.9 shows a remarkable drop in the values in the interpolated area between the two functions. This will clearly impact the correctness of a transition among them. The inaccuracy is probably due to the fact that the parameter information goes through an inferior network depth compared to the full model. Nevertheless, these results leave room for further research on how to condition with one state parameterless and let the network, queried with different tasks, pass through that state.

4.1.3 Comparison of the previous models

A comparison of the three previously discussed models is presented below to better evaluate the most capable and cross-validate the results. A different dataset was built with different parametric functions: linear, parabolic, and sinusoidal. This also implies the presence of three different tasks to feed to a single CNMP network. For each skill, different trajectories are present in the dataset to enable new trajectory generation for conditioning points of unseen values.

Moreover, the transition among all tasks is now represented with three values, so the first task is encoded as [1][0][0], the second as [0][1][0], the third as [0][0][1]. This avoids passing through the middle one as in the case of a single parameter [1..2..3] encoded network. The transition is performed by decreasing one value and increasing the other simultaneously.

Table 4.3: Full Comparison Table of errors of CNMP vs CNMP with TP only in condition vs CNMP with TP only in query

MSE Error on	**TP**	**CNMP**	**CNMP TP condition**	**CNMP TP query**
demonstrated trajectory	1	2.200240773e-06	0.0005239668753	0.00015134105003
	2	2.852831969e-08	3.317327413e-05	1.5758800492e-05
interpolated trajectory	1	2.8834989373e-06	0.0004749418782	0.0008166800702
	2	2.598636844e-07	3.6395583963e-06	0.0002007523980

In fig. 4.10, it is possible to observe all the possible combinations for interpolating the different tasks on which the network has been trained. Even with more than two tasks and three dimensions added to the input the CNMP model performs sufficiently well in all three skills.

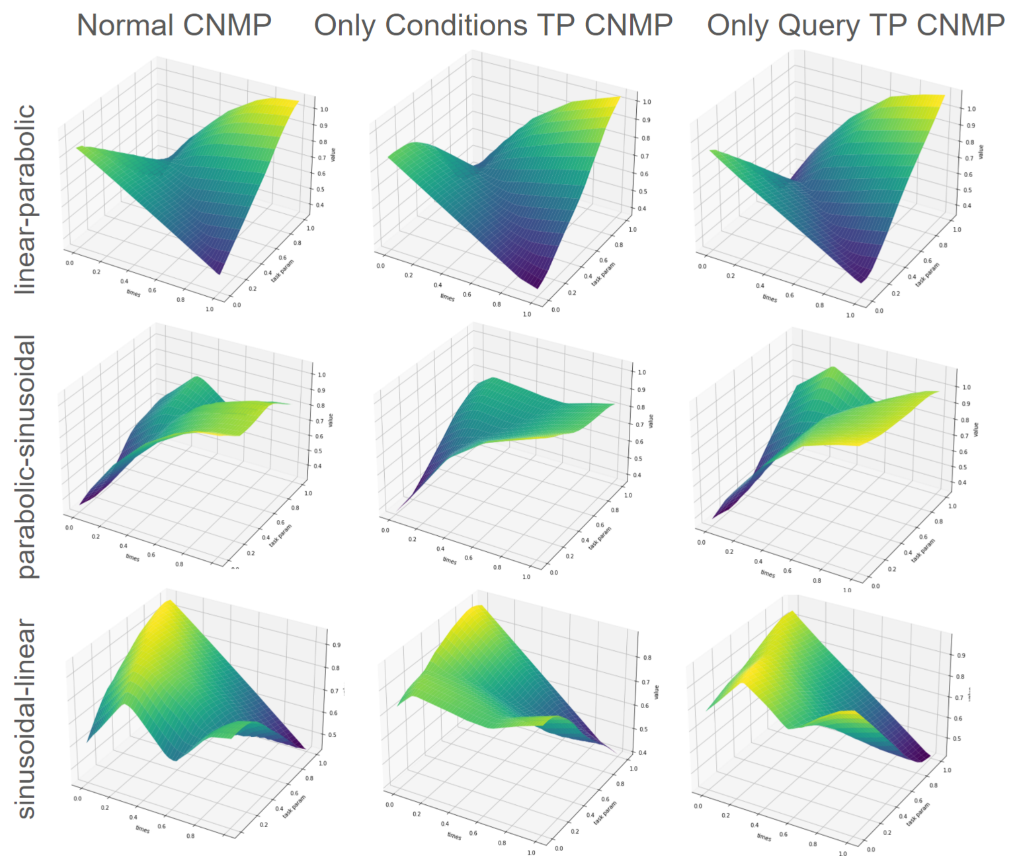

Figure 4.10: Interpolation comparison of the 3 CNMP models analyzed for a different dataset and multiple tasks.

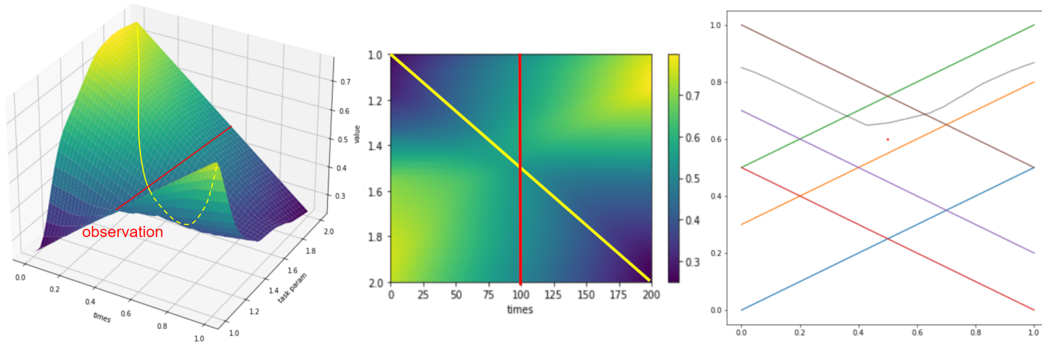

Figure 4.11: Transition in time of task through task parameter shifting

4.1.4 CNMP changing task in time with one conditioning point

Analyzed the interpolation capabilities of the CNMP networks also on the task dimension, a possible method to shift the execution of a skill in time is proposed below.

In order to build the interpolation visualizations, the network was always queried for every time-step with a constant γ. However, since the queries are independent, it's possible to query the time and parameter singularly as required.

Moreover, from the previously constructed plots, it is evident how it is possible to change tasks in time via querying the network with TPs linearly changing with time.

In fig. 4.11, the path of the queries of time and γ is depicted in yellow. As time goes on, left to right, the action moves from descending (back of the graph) to ascending (front of the graph). This achieves a smooth change between task parameters using the interpolation space provided by the model.

In this case, math leverages the neural network's hidden capabilities to wisely input the required parameters to generate the desired output.

It's worth noting that this method works with any CNMP model variation previously discussed, as long as the interpolation space is built correctly. Further research could include better training to improve the results in the interpolation dimensions.

The mixing of the two different skills is indeed performed through time, and in this case, the change is linear. Different functions analyzed produced different results, among them the most famous ones like sigmoid, logarithmic, and others. The model is capable of different transition speeds and periods depending on the selected function. The linear function is selected given that it produces the smoothest transitions since inclination is minimal in all the timesteps.

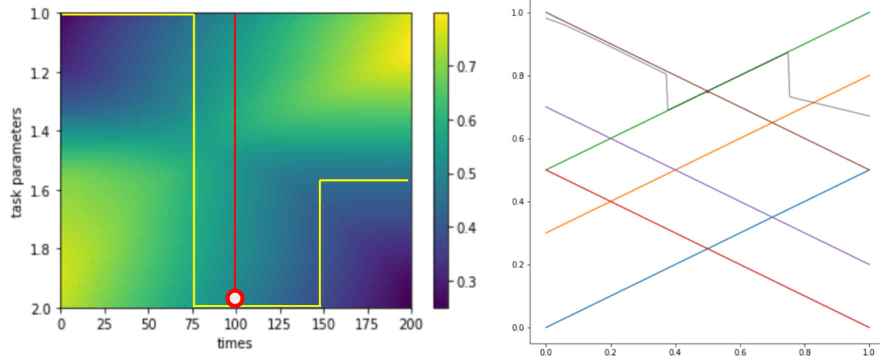

Figure 4.12: Different functions for changing task parameter in the CNMP network

On the opposite, a sharper function, like a step function in fig. 4.12, produces a shift that is immediate and full-fledged a stitch among two parts.

In fig. 4.11, it is possible to see that the final trajectory result is not passing through the conditioning point. This occurs since, at that moment $t = 0.5$, the transition between two different task parameters is halfway ($\gamma = 1.5$) and not fully on one γ. The interpolated space varies in quality depending on the network and its training, and this means the error would be higher compared to the values on γ on which the model was trained.

Surprisingly, it's not actually a problem because the conditioning point is nothing else than a point that symbolizes the transition between two conditioning points with different γ located on different positions. The conditioning point is just a point that allows the transition without introducing any bias because it's part of both trajectories. In fig. 4.13, it is possible to see the red conditioning point emulating two different conditions with different TP, respectively descending for the left one and ascending for the right one.

This means that the conditioning point for transition is not meant to be anywhere, but it has to be at the intersection of the two desired functions with different TPs. This will grant the resulting function to pass at the initial time through the first conditioning point, since its TP is not mixed, and at the final time through the second condition for the same reason.

In the example fig. 4.13, at time $t = 0$, the function passes through the first point because the task parameter is fully descending. This is guaranteed because the red observation chosen is on the line produced by that initial point. In the same way, at time $t = 1$, it's guaranteed to pass through the rightmost point because the observation is also on the function that is produced by that condition. For this reason, the observation point is not meant to be independent but derived from the two points to emulate.

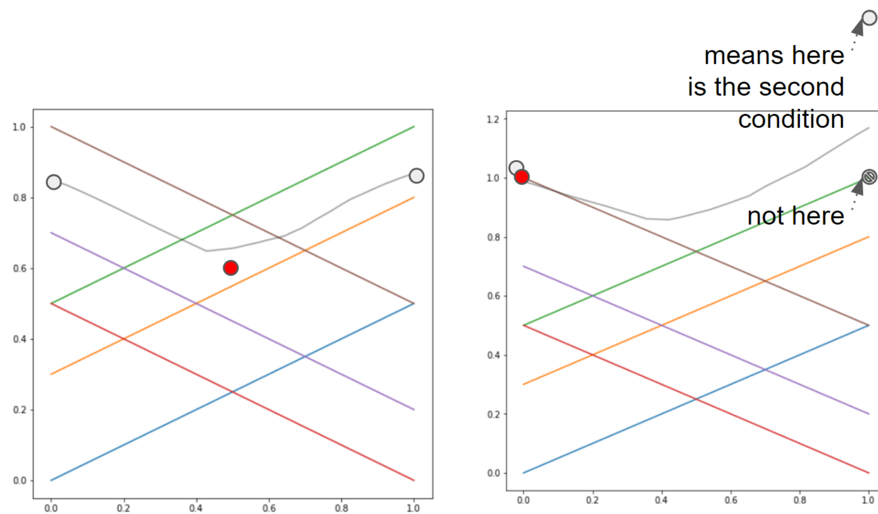

Figure 4.13: A conditioning point that varies task parameter emulates two different conditioning points of different task parameters.

The best strategy to find the conditioning point for the transition among two conditions of different γ seems to place it on the intersection of the two trajectories generated by these conditions. However, it is not always granted that the trajectories will intersect. As discussed above, in multidimensional spaces, the probability of this event decreases significantly.

When two functions don't intersect, the best possible solution to not bias the interpolation is, while changing the γ, to shift as well the position of the condition in time. In the series of queries to the network, instead of giving the same context point and changing its task parameter, the position also changes. The optimal change of position is from a point on the first function generated by the first condition to the closest point on the function generated by the second condition. This guarantees the series of points will stay on the interpolation of the two functions and generate the desired interpolation surface area.

Extending the concept, the algorithm defined looks for the closest points among the two functions, and it transitions among them. The closest couple of points is looked only in the time span in which the network was trained, since it is a computation of cost t^2. If there is an intersection, the change on the conditioning point will be only on the task parameter. If there is no intersection, the change of the conditioning point will also be a change in position from the closest point of the first function to the closest of the second one.

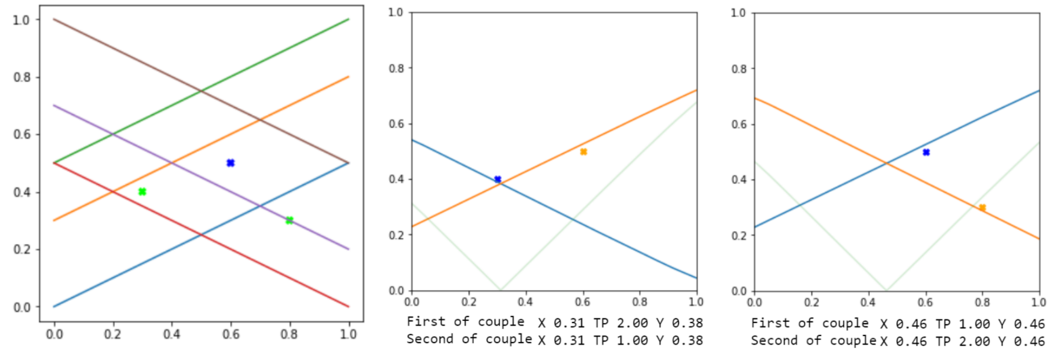

Figure 4.14: Procedure to individuate multiple conditions for multiple shifts in the task parameters.

4.1.5 CNMP changing task in time with multiple conditioning points

The system developed enables a single transition using a single conditioning point. The shift occurs completely from an initial time that is $t = 0$ to the final time of the network training.

It's a remarkable success because the bare normal CNMP model is completely incapable of changing the task coherently and continuously among its predictions. If fed with two conditions of different TP, the network outputs a completely unusable trajectory. This trajectory is a rough average of all the different ones generated by the conditioning points. This is due to the fact that the model, by definition, doesn't have an attention mechanism and doesn't lose the conditioning power in time.

However, this research extends the method further to multiple conditions with different task parameters. This enables the full control and customization of the predicted output.

Furthermore, it achieves the shift among the observations in a desired time span not restricted to the full training time length.

Once the interpolated surface is built, the time constraints are resolved with a fast transition from the first condition timestep to the second condition timestep. The resulting output sequence will still reside on the interpolated surface but, having less time, will be faster.

The multiple transitions are achieved by coupling them one by one in temporal order, so the first one determines the task parameter till its time. Then, between the first and the second one, the task is shifted. At the second observation, the task parameter will be fully on its task, but subsequently, will start to be merged with the third one, and so on. This guarantees that at the observation's times,

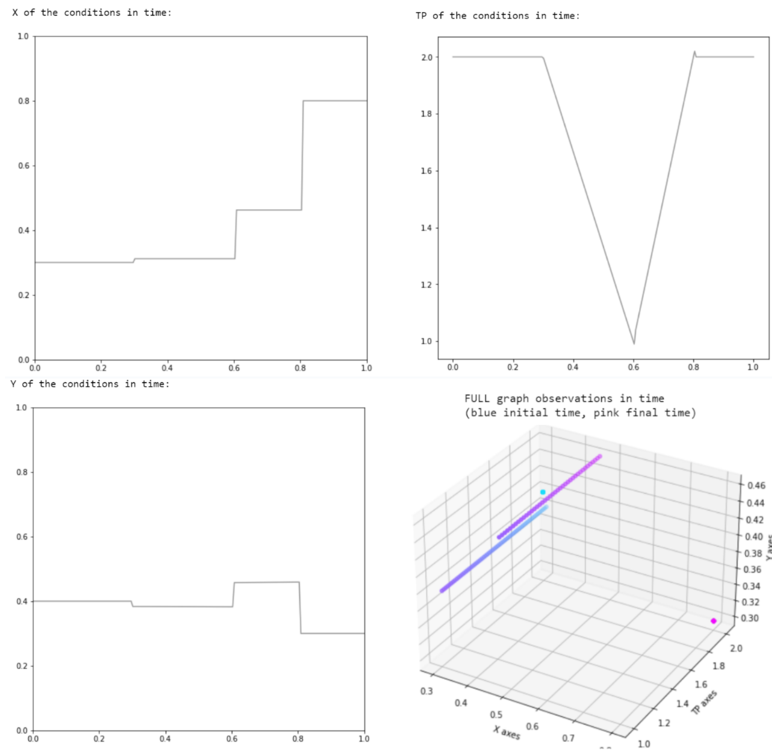

Figure 4.15: Graph and full path in time of the condition point used to transition multiple tasks parameters

the output function will pass through them, but in between, the task change will occur.

In fig. 4.14, it is possible to view the initial observations of different parameters, two green conditioning points for the descending task and one blue in the middle for the ascending task. The first couple in temporal order is selected. At this point, the two functions each condition will independently create are generated. Those are visible in the fig. 4.14 in the second plot.

The closest points between these functions are selected, the absolute distance among the points is plotted as the green line at the bottom of the plot. In this case, since there is an intersection, the couple selected has the same starting and ending position, which only differs for the γ parameter. During the shift time, the conditioning point of the network will transition from the first identified to the other one, changing TP. If the two closest points were in different locations, it would also change location, along with task parameter. The network is queried at the appropriate time with the condition designated.

The process repeats for the second couple and so on to find the conditioning points to feed the network to obtain the desired result. The final full path of the

conditioning point is visible in fig. 4.15 where it changes position in time according to the place where it will not bias the current transition. Meanwhile, the changing of the TP will occur during the time span designated.

It's worth noting that the conditioning point will also change its time, independently from the time queried. For this reason, in the 3D graph in fig. 4.15 the fourth dimension is introduced as color. The x axes of the graph corresponds now to the time of the condition, while the time of the queries corresponds to the change in color.

Finally, the results shown in fig. 4.16 demonstrate how this method can achieve trajectory predictions that are coherent across multiple shifts of tasks. The normal CNMP model with the same conditioning points is reported aside for comparison.

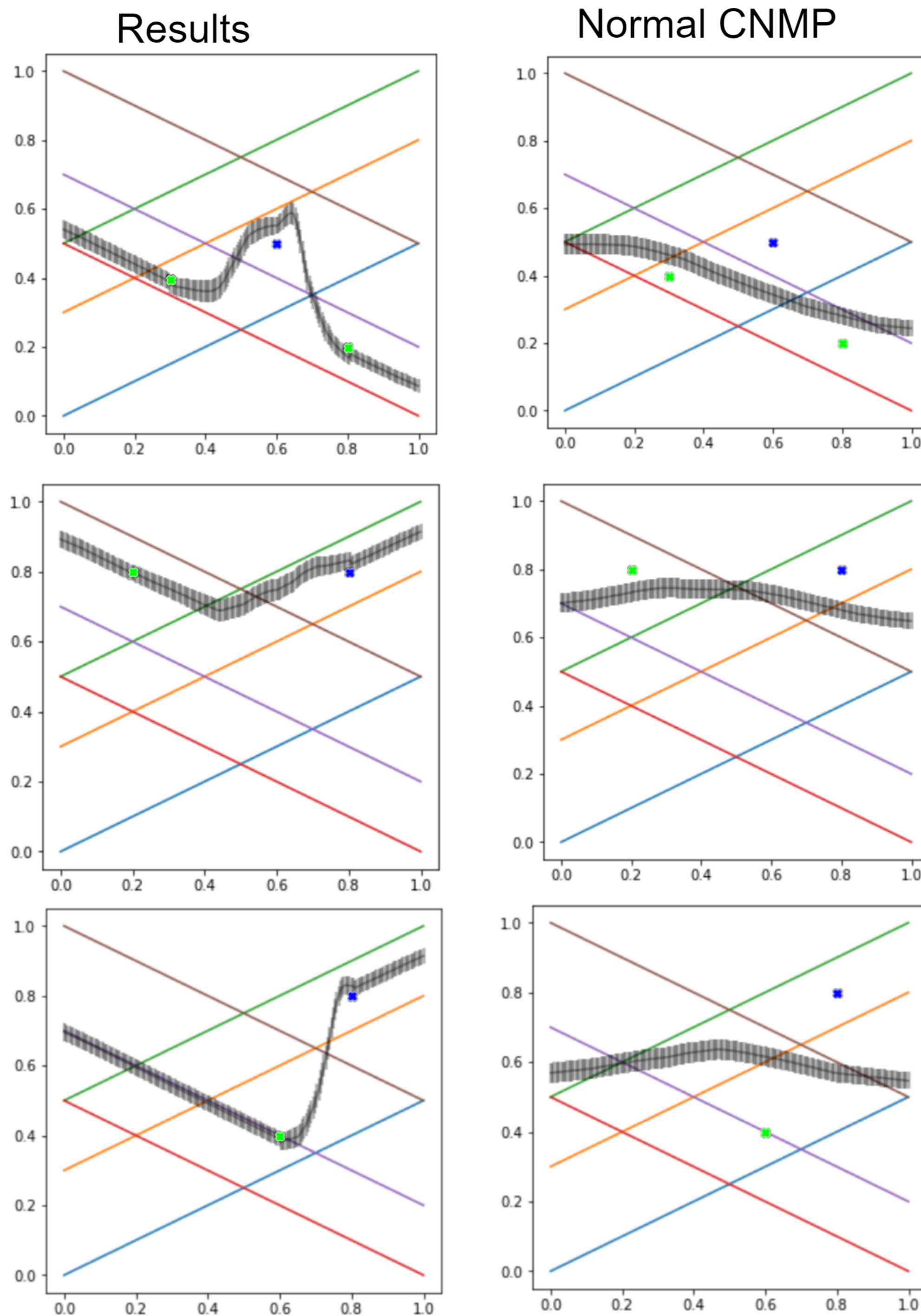

Figure 4.16: Final results of multiple tasks transition and comparison with traditional CNMP network.

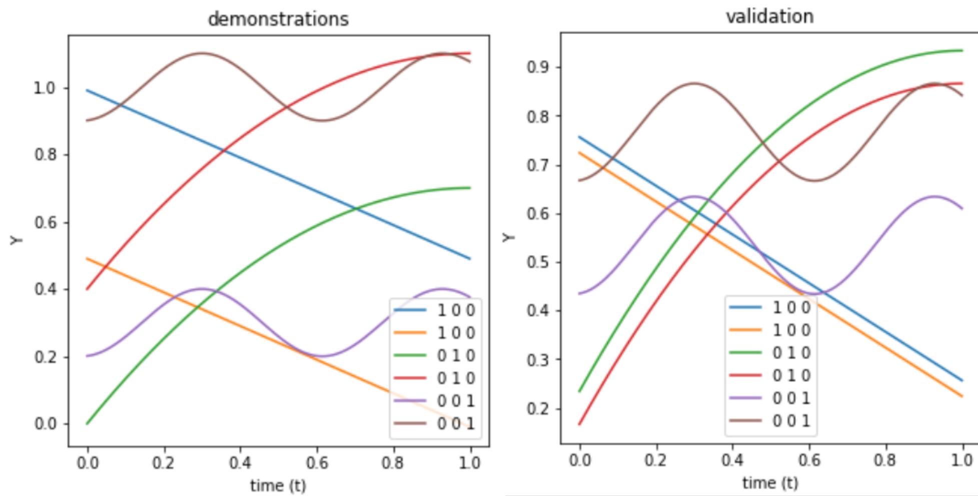

Figure 4.17: Artificial dataset of different skills to concatenate. On the left, there are the demonstrations. On the right, there is the validation.

4.2 End-To-End Skill Concatenation

The proposed method in the previous chapter would be really limited if not coupled with the ability to compose the skills in succession. Being able to combine parts is indeed useful, but it will eventually end in the time span of one action, even if we use parts of other ones. To really enhance the potentiality of the network and exponentially enlarge the capacities of the robot, a way to join the skills end-to-end is required.

In this part, a method is proposed with which it is possible to concatenate skills one after the other to achieve a goal or reach the final target position or desired state.

In order to synthesize a series of actions, the demonstrations have to be initially collected. Subsequently, the method presented will combine them end-to-end to reach a desired objective.

An immediate challenge presented consists of the concatenation of the movement primitives without incurring in jumps when passing from one to the other. Abrupt jumps in the final movement execution, as previously analyzed, can be dangerous and look unnatural.

The requirements, also for this research subpart, are indeed that the motions have to be executed with meaning, they have to seem natural, and they have to be safe.

Another challenge to face is the different lengths of the actions in time and space. Some actions might be faster than others because they are more easily executable, for example, picking an object compared to pouring water. Further-

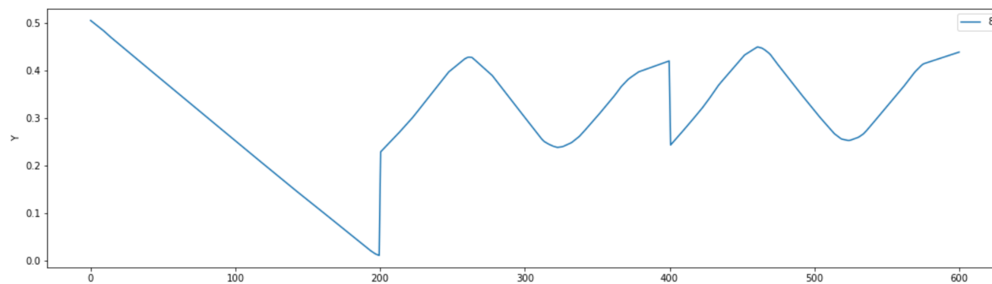

Figure 4.18: An example of failing concatenation of skills due to abrupt jumps.

more, the same actions might last less if performed in different points of space, for example, placing an object in a container next to it or further away.

Another possible challenge is choosing the right action sequence to perform. Having all the possibilities from a starting point means that a meaningful choice has to be made. Every action has an effect on the environment, and some of them have a specific order that can not be permutated. Deciding the right order is a complication to solve,

For simplicity, the example that will follow will refer to a simple dataset fig. 4.17 that eases the comprehension of the concepts. Nevertheless, this method can still be extended to multiple dimensions and more complex scenarios.

In the dataset artificially generated, two trajectories are provided for each skill. They can resemble any task or movement in time, for example, push or shake an object or move to a position. This dataset enables a clear visualization later on. More realistic but more difficult-to-understand data can be found in the chapter 6.

The different skills, in this case, are given to the same network with the task-parameter option seen in the previous chapter, but it's not necessary. The task parameters are, this time, three, one for each task, and they are boolean, so it's clearly visible which task is queried to the network. The task parameters are visible in the legend of the fig. 4.17.

As mentioned in the previous chapter, for every skill, not one but multiple different trajectories are collected. This is not mandatory, as the network can learn with as little as one demonstration. However, having multiple expert demonstrations allows the network to generate new unseen ones from interpolation. This extra capability is fundamental since it allows the extension of the effectiveness of the actions from a single point to a whole area or volume. Furthermore, having a network that generates new unseen trajectories will allow the method proposed to be validated later for these cases.

The interpolation abilities of the network are validated on the trajectories visible in the right plot. These trajectories enable the code to verify if the movement primitives generated by the network have a minimum error or not.

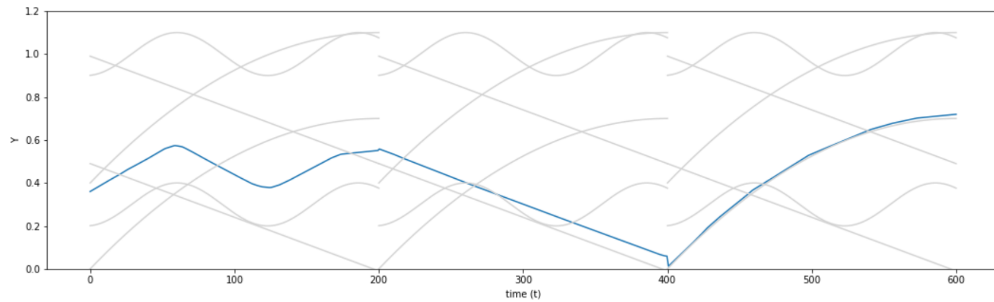

Figure 4.19: An example of using interpolation with CNMP model to generate new consecutive smooth trajectories.

The joining of multiple trajectories can't be a simple stitch of the previously recorded ones since it will also create abrupt jumps, fig. 4.18. We want to avoid this event for the safety and human-like movements as required but also for a deeper further reason.

Since the network can deal easily with multidimensionality, some dimensions might be added in the future to represent the state of the world. Jumps in the final trajectory must be avoided because the network would allow unmotivated internal world representation changes without being the author of them. However, this is an option that we will explore at the end of this chapter.

4.2.1 CNMP for skill concatenation

To overcome part of the concatenation abrupt changes problem, the ability of CNMP of interpolation is used again. Once the final point of an action is reached, the model is queried to generate the next skills with a conditioning starting point matching it. The model, conditioned on that initial position, will generate the actions required that start from that state, avoiding big discrepancies in the final trajectory generated.

In fig. 4.19, it is possible to see that the first trajectory is the result of the interpolation of the two sinusoidal demonstrations of the dataset depicted in grey. Furthermore, the end of the first trajectory is used as a condition to generate the second one. The second is the interpolation of the two linear functions, also depicted in the background.

It's worth noting that in the example, the output is monodimensional, and the interpolation is simple, but in multidimensional inputs, the CNMPs can interpolate and generate output primitives that are able to interpolate in the whole 3D or joint space. This means that the trajectories of every joint, or 3D axis, won't have abrupt jumps.

Using a real-life example, if the final position of an action is reached and no

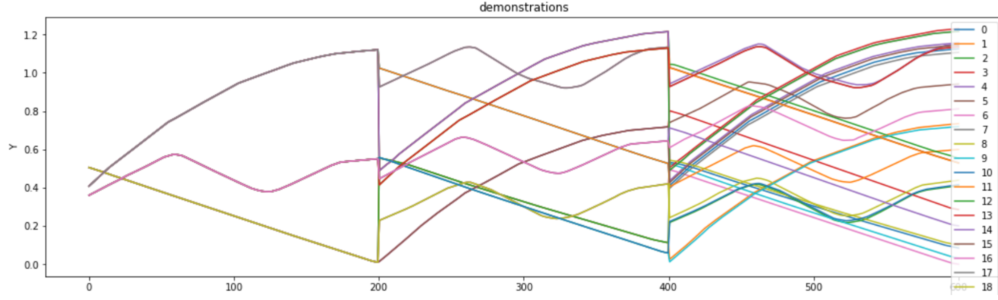

Figure 4.20: An example of a recursive concatenation of movement primitives using interpolation abilities of CNMPs.

other action demonstrated starts from there, the model will use the demonstrated trajectories that start from other points and combine them to generate a skill that starts from that position.

At this point, given a starting point, the method proposed recursively builds the graph of all the possible actions, fig. 4.20. This means that all the actions are evaluated from the starting position, then for all the positions reached, all the actions are generated, and so on.

The cost of this action is exponential with time; the cost is $O(n_a^d)$, where n_a is the number of actions available and n_t is the depth number of subsequent actions. The cost is high but can be easily reduced since the network can be queried only for the prediction of the final step, and the queries take milliseconds.

Furthermore, mechanisms for pruning the tree if a jump is detected can significantly reduce the number of possibilities. Moreover, the building of the tree can stop once a viable sequence is found to reach the final goal. Lastly, more informed research is possible with heuristic algorithms that can reduce the steps to find a desired goal. However, the scope of this research remains to demonstrate the validity of the method, so the optimizations are left for future work.

Even with the interpolation ability, the graph still contains some jumps, and we want to avoid them for the reasons explained previously. This is due to the fact that repeating some actions over and over again will bring the state out of the demonstration range provided to the network.

The CNMP model shines in the interpolation but lacks the ability to extrapolate from demonstrations. So, if a conditioning point is positioned out of the area between expert demonstrations, the network will simply output the demonstration that is the closest to that point without going further.

This problem actually helps to find the actions that are simply not feasible. For the sake of safety and meaningfulness, it is not actually reasonable to extrapolate the actions demonstrated to the whole space around the agent. For example, concatenating too many push actions might, in real life, exceed the reachability

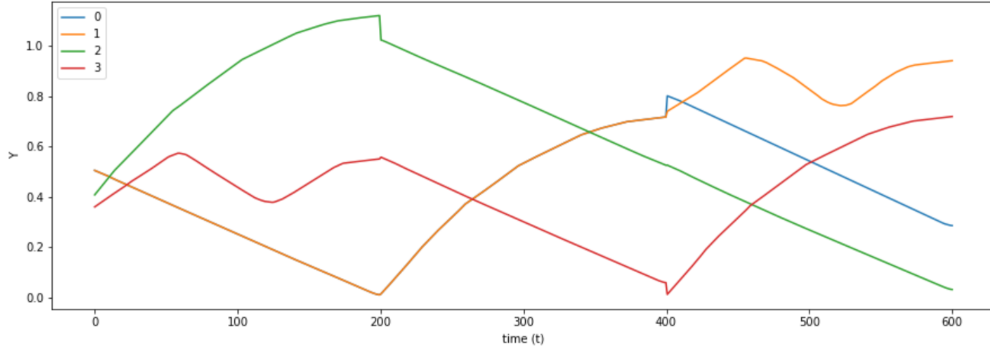

Figure 4.21: An example of filtering the skills generated based on extrapolation limitations.

area of the robot or its arm length.

For this step the assumption taken is that the expert gives demonstrations that are at the limits of the capacity of the robot. This allows the method to derive what is feasible and what is not.

Consequently, to remove the actions that are not feasible from the option set, it is sufficient to leverage the limitations of the model and just look for big data variations in the sequences generated. After finding them, it's possible to prune the tree from these unfeasible action concatenations.

In fig. 4.21 are presented the selected trajectories generated by the network, with interpolation to avoid jumps, but without extrapolation to avoid the impractical ones. As it is possible to see in the picture, among the four trajectories chosen, it's also present in red the one previously shown in fig. 4.19 and created using interpolation.

All the actions presented in the example have the same temporal length. This is not mandatory but a choice for the sake of simplicity. An extension of this research could include skills that have different durations. Overall, since the majority of movement primitives are simple actions and last a few seconds, this complication was not addressed.

4.2.2 CNMP embedding environment representation

Trajectories generated in this way are executable and this means that the method works for spatial concatenation. On the other hand, the final goal in this way can be only a position in space or a joint state. Although it is really useful to make the robot or the agent reach new places using the previous knowledge, this is limited to the device or, at maximum, the object it can carry.

Upon observation, it's clear that many movement primitives involve an external

object or the interaction with the environment around them. Another step in complexity is to develop the method for goals that are not only as a point in time or robot position but also related to the external world.

The external world introduces a whole new degree of complication because it requires having a degree of knowledge about it. Without this internal representation of the environment, the robot would execute actions that are possible but not meaningful. For example, picking an object that is not present or placing a lid before the pot.

Since the CNMP model can deal with multidimensional data, the solution proposed is to embed the state changes of the environment in the network. Given a representation of the world, the network can learn how the actions performed change it and the final outcome. The representation can be as simple as a single x, y position of an object manipulated or the whole image of the environment.

Furthermore, the network's ability to interpolate will make it adaptable to new states, as long as they can be interpolated from demonstrations, and it will be able to predict the outcome of the actions on them.

Table 4.4: Example of CNMP ability to embed environment representations

	left arm	right arm
move arm to center	[**0**,1,2] → [**1**,1,2]	[0,1,**2**] → [0,1,**1**]
move arm to its side	[**1**,1,2] → [**0**,1,2]	[0,1,**1**] → [0,1,**2**]
pick and place from side to center	[**0**,**0**,2] → [**1**,**1**,2]	[0,**2**,**2**] → [0,**1**,**1**]
pick and place from center to side	[**1**,**1**,2] → [**0**,**0**,2]	[0,**1**,**1**] → [0,**2**,**2**]

A simplistic example is reported. The CNMP model can be trained on both arms of a robot and enable the movement of an object from a position that is reachable only from the left arm to a position that is reachable only to the right arm. This action requires the internal knowledge of the position of the object to find the right sequence and not to execute actions that are possible but not meaningful if the object is not in the desired position. The object position is encoded with a state $[1, 2, 3]$ depending on its position from left to right. We assume the left position is only reachable from the left arm, the central position from both arms and the right position from the right arm. The arms' positions are encoded in the same way, but by definition, the left arm can reach only the first two and the right arm only the last two. In a real-world scenario, the position of the arm will not be monodimensional but probably multidimensional, but the example is still valid. The world representation is finally $[PositionLeftArm, PositionObject, PositionRightArm]$. The CNMP model will learn the changes in time of both the arms and the environment representation. The possible demonstrations are depicted in table 4.4,

where pick and place means close the gripper, move to a position, and open the gripper.

Given a condition on the initial state where the object is on the left $[0, 0, 2]$, the method proposed will build the whole graph of possible actions and resultant environment states. At this point, the sequence for bringing the object from left to right, final state $[0, 2, 2]$ is:

1. $[0, 0, 2] \rightarrow [1, 1, 2]$ left arm pick and place from side to center
2. $[1, 1, 2] \rightarrow [0, 1, 2]$ left arm move arm to its side
3. $[0, 1, 2] \rightarrow [0, 1, 1]$ right arm move arm to center
4. $[0, 1, 1] \rightarrow [0, 2, 2]$ right arm pick and place from center to side

It's worth noting that the method doesn't require additional data structures, but it's all embedded in the network.

This method extension means adding the dimensions normally to the network's input and output. In this way, it's still possible to use the previously presented method to discard the actions that don't bring to a valid state. In the example, it is impossible to move one arm in the center while the other is there so the arms won't collide.

Moreover, the action possibilities can still be filtered by analyzing abrupt changes in the world representation to understand when an action's output will lead to a state where further actions are not possible. In the example, the object must be in the center to be picked and placed on the side. If not, the generated action will show an abrupt initial change in the environment representation.

While the presented method works out of the box for reaching goals in space, a possible extension for meaningful executions is presented here. Nevertheless, the scope of this work is not action planning, which is a complex topic in robotics beyond the scope of this work.

Chapter 5

Implementation

This chapter gives a more in-depth view of the implementation of the methods discussed before. This doesn't mean other implementations are not possible or the method designed can't be built in other programming languages and frameworks.

The platforms and tools presented in the chapter 3 are used. Python was chosen as a programming language for its ability to deal with data and machine learning but it can as well be used for robotics thanks to the "rospy" package.

The most important passages in the code developed and only the key points of the solution implemented will be explained below. They can be used as a reference for clarification, for another implementation, or for further improvements.

This code is available in a GitHub public repository, [19].

Following the structure of the chapter 4, this chapter is divided into two main subsections.

The first subsection explains the implementation of the partial skill combination. The parts of this section follow the previous chapter; the first is relative to the implementation of the CNMP model with the task parameter only in condition, and the second is relative to the CNMP model with the task-parameter only in query. The third part is a comparison of all three models. Lastly, the last two parts will explain how CNMP changing task executed in time with one conditioning or multiple conditions are developed.

The second subsection shows the implementation of the previously explained End-To-End Skill Concatenation method.

Since the CNMP model has been used in both subsections, the structure is briefly reported in listing 5.1. The encoder and the decoder networks are multilayer perceptions consisting of three fully connected layers with the non-linear ReLU activation function. The dimensionality is fixed to 128 dimensions except for the first and last layers.

The encoder's first layer has a dimensionality equal to the sum of the input and output ones, since the observations have both of them. It will output the latent

Listing 5.1: CNMP model code

```
import torch
class CNMP(nn.Module):

    def __init__(self):
        super(CNMP, self).__init__()

        # Encoder takes observations which are (X,Y)
           tuples and produces latent representations
           for each of them
        self.encoder = nn.Sequential(
        nn.Linear(d_x+d_y,128),nn.ReLU(),
        nn.Linear(128,128),nn.ReLU(),
        nn.Linear(128,128)
        )

        #Decoder takes the (r_mean, target_t) tuple and
           produces mean and std values for each
           dimension of the output
        self.decoder = nn.Sequential(
        nn.Linear(128+d_x,128),nn.ReLU(),
        nn.Linear(128,128),nn.ReLU(),
        nn.Linear(128,2*d_y)
        )
```

representations to aggregate together with position-independent operations, as in equation 2.2.

The first layer of the decoder has a dimensionality equal to 128 for the latent representation plus the input query desired to concatenate. The decoder outputs double the output dimensions since for each one is returned the mean and standard deviation.

The forward function built in listing 5.2 passes the observations provided to the encoder, then generates the mean of all the 128 latent representations created. For all the queries, the representation is concatenated to them and passed to the decoder. The decoder produces the mean and the standard deviation for each target queried.

Lastly, the loss function implemented is visible in listing 5.3 is the implementation of the log probability loss explained in the chapter 2. The standard deviation

Listing 5.2: CNMP model forward function

```
1     def forward(self,observations,target_t):
2         r = self.encoder(observations) # Generating
             observations
3         r_mean = torch.mean(r,dim=0) # Taking mean and
             generating the general representation
4         r_mean = r_mean.repeat(target_t.shape[0],1) #
             Duplicating general representation for every
             target_t
5         concat = torch.cat((r_mean,target_t),dim=-1) #
             Concatenating each target_t with general
             representation
6         output = self.decoder(concat) # Producing mean
             and std values for each target_t
7         return output
```

Listing 5.3: CNMP model loss function for trainig

```
1 def log_prob_loss(output, target):
2     mean, sigma = output.chunk(2, dim = -1)
3     sigma = F.softplus(sigma)
4     dist = D.Independent(D.Normal(loc=mean, scale=sigma)
         , 1)
5     return -torch.mean(dist.log_prob(target))
```

values σ (”sigma” in the code at *line 3*) are passed through the Softplus activation function. Softplus is a smooth approximation of the ReLU function and ensures that the standard deviation remains positive.

The following line creates a PyTorch distribution object. It specifies that the distribution, with independent axes, is Normal with mean mean and standard deviation sigma.

Finally, the negative log-likelihood of the target values given the distribution (dist) is computed. This is used for backpropagation in network training.

Listing 5.4: CNMP model architecture change for TP only in conditions

```
        #Decoder takes the (r_mean, target_t) tuple and
           produces mean and std values for each
           dimension of the output
        self.decoder = nn.Sequential(
        nn.Linear(128+d_x-d_TP,128),nn.ReLU(), #edited
           here to pass only x without tp
        nn.Linear(128,128),nn.ReLU(),
        nn.Linear(128,2*d_y)
        )
```

5.1 Partial Skill Combination

In this section, the concept of task parameters was introduced. At the implementation level, it doesn't change the previously explained network structure since the dimension of the input will be increased, for example by one: $d_x = 2$, and the code will adapt the network structure.

An example of input for a query will be:
$tensor([[0.1357, 1.0]], dtype = torch.float64)$.

Finally, an observation will be in the form of $[0.6], [1.0], [0.45]$, where the two dimensions of the input (note the 1.0 as TP) are concatenated with the output desired.

Another crucial passage in the chapter 4 was to vary independently the two task parameters of the observations and the queries in order to build the visualization of their influence in fig. 4.5.

To achieve this, a simple matrix of observations and queries was built, where each row contains observations and a time query to the network. Subsequently, the matrix was edited with constant task parameters in the observations, letting the TP vary in the queries, in order to build the plot. The opposite was implemented in order to build the plot where the task parameters vary in the observations and are constant in the queries.

5.1.1 CNMP model with task-parameter only in condition

In this part, a new architecture is implemented, as proposed in chapter 4, based on the previously explained CNMP model.

Among the edits in the code, one of the most important ones is the change of the model itself in the definition seen previously in listing 5.1.

Listing 5.5: CNMP training code change for TP only in conditions

```
predicted_Y,predicted_std = predict_model(np.array([np.
    concatenate((v_X[i,0],v_Y[i,0]))]), np.delete(v_X[i],
     -d_TP, axis=1), plot= False) #edited here to pass
    v_X only with first element and not tp
```

As is visible in the listing 5.4, while the encoder stays the same, the decoder has input dimensions reduced. The subtraction is at *line 3*, after the concatenation of the latent representation's 128 dimensions and the d_x input dimensions, where d_TP is the number of dimensions of the task parameters.

Another fundamental change in the process is the input at the prediction time. The listing 5.5, taken from the prediction during training, shows the normal conditions in the first part of the function. In the second argument, the queries are passed. These are in the form of a matrix with inputs and task parameters in the columns, and the rows are the times.

In the new version, it is possible to see how the matrix is deprived by the last d_x columns containing the task parameters. The input passed to the network corresponds indeed to the dimensionality of the decoder designed before in the model.

This allows to query the network simply with an array of times t desired.

5.1.2 CNMP model with task-parameter only in query

In this part, another new architecture is proposed based on the previously explained CNMP model, as the opposite of the one presented in the previous part.

To maintain the comparison between all the code developed, the most important changes presented here will follow in parallel the previous part.

One of the most important changes is again the model architecture itself, seen previously in listing 5.1.

As it is possible to see in listing 5.6, the decoder this time was untouched, but the encoder was modified to accommodate different dimensions in his input. The dimensionality of the first layer of the network was reduced by the number of task parameter dimensions. At *line 3*, the dimensions of the task parameter d_TP are subtracted from the dimensions of the observations $d_x + d_y$.

Similarly to the previous architecture, another fundamental change in the process is the input at the prediction time. The listing 5.7, taken from the prediction during training, shows this time the normal queries $v_X[i]$ in the second part of the function. In the first argument, though, the observations are passed in a different way. These are also in the form of a vector with inputs, outputs, and task

Listing 5.6: CNMP model architecture change for TP only in query

```
        # Encoder takes observations which are (X,Y) tuples and produces latent representations for each of them
        self.encoder = nn.Sequential(
        nn.Linear(d_x-d_TP+d_y,128),nn.ReLU(), #edited here to pass only x without tp
        nn.Linear(128,128),nn.ReLU(),
        nn.Linear(128,128)
        )
```

Listing 5.7: CNMP training code change for TP only in query

```
predicted_Y,predicted_std = predict_model(np.array([np.concatenate((np.delete(v_X[i,0], -d_TP, axis=0),v_Y[i,0]))]), v_X[i], plot= False) #edited here to pass only with first element and not tp
```

parameters concatenated in order.

In the model implemented, the observations are deprived of the last *d_x* columns of the input relative to their task parameters. Now, the observations passed to the network correspond to the dimensionality of the first layer of the decoder designed.

This allows conditioning the network simply with conditions that are parameterless.

5.1.3 Comparison of the previous models

For the comparisons among models developed, a brief extract of code is reported in listing 5.8. Using the *numpy* library, the real values are compared to the predicted ones.

Different metrics have been used. Initially, the Mean Squared Error (MSE) is calculated. It measures the average squared difference between the actual (*real_values*) and predicted values (*predicted_Y*). It penalizes larger errors more severely due to the squaring operation. Since the errors produced were minimal, this is the metric chosen and reported in table 4.3.

Subsequently, also the Mean Absolute Error (MAE) is computed at *line 3*. It provides the average absolute difference between the actual and predicted values.

Lastly, the Root Mean Squared Error (RMSE) is calculated on the basis of this

Listing 5.8: Extract of code for comparison of models

```
import numpy as np
mse = np.mean((real_values - predicted_Y)**2)
mae = np.mean(np.abs(real_values - predicted_Y))
rmse = np.sqrt(mse)
```

one since it is the square root of the Mean Absolute Error. These metrics were chosen as they are the most commonly used in regression analysis.

5.1.4 CNMP changing task in time with one conditioning point

In this part is present a possible implementation of the method proposed to shift the execution of a skill in time.

For the sake of simplicity, the model used was the proposed CNMP without task parameters in the query. This does not imply that the method does not work with the other CNMP architectures analyzed.

The method is a mathematical way to feed the model with the right conditions, so it is possible to implement it on other networks for custom needs or for better performance.

Implementations for the other architectures were also developed but, for simplicity and avoiding repetitions, only one will be explained.

As depicted in fig. 4.11, the shift implemented is linear, and, given an observation, goes from one task parameter to the other. In listing 5.9, a matrix of times linearly changing task parameters and values is created. The values column is empty and will be filled later.

Subsequently, the model is queried for every timestep, with the TP changing *line 7*. Finally, values predicted are inserted in the third column of the matrix for every time step at *line 10*.

At the end, the resulting matrix will have all the data required. The plotting process visible on the right in fig. 4.11 will not be described here cause it is not necessary, but it was achieved using the *Matplotlib* python library.

5.1.5 CNMP changing task in time with multiple conditioning points

Below are proposed key parts of the final implementation for changing multiple times tasks in the same prediction time span.

Listing 5.9: Extract of code for task shift with one observation

```
mixed_tp=graph[0].copy() #first column time
mixed_tp[:,1]=np.linspace(2,1,mixed_tp.shape[0]) #second
    column tp changing

for index, el in enumerate(mixed_tp):
    with torch.no_grad():
        print("now time %f and TP %f"%(el[0], el[1]))
        pred_y = model( torch.from_numpy(np.array([np.
           concatenate(([0.5],[el[1]],[0.6]))])) ,
        torch.from_numpy(np.array([[el[0]]])) ).numpy()
        print(pred_y[0][0]) #  value returned
        mixed_tp[index,2]=pred_y[0][0] # fill 3rd column
print(mixed_tp)
```

There are two key functions for achieving this result: the first is dedicated to building a matrix of data, and the second one has as duty the use of this matrix to generate the predictions.

Building the matrix of timestep and observation for the timesteps. The first key part is building a matrix that defines how the conditioning point will move in time and task space. This function is responsible for the results visible in the fig. 4.14 and fig. 4.15.

The function receives the observation list, with time as the first dimension, and returns the matrix with the observation for every timestep. Initially, the observations are sorted based on their time. The matrix is defined with queries of time (in the first position) concatenated to only one observation at that specific time. The matrix has dimensions $1 + d_X + d_Y$, recalling that d_X includes the d_TP dimensionality of the task parameters.

Subsequently, the action will be repeated for every timestep. The loop also keeps track of the current and next observation in time, starting from the first. Some optimizations were performed not to calculate the same couples of closest points every time, but won't be reported here for shortness.

If the current time is less than the observation time or this is the last observation, the function fills the current timestep of the matrix with the current observation. This takes care of the initial and final periods.

If the timestep is in between two observations, the core part happens. The code implementation should check the whole predictions in time of the two observations,

Listing 5.10: Extract of code for task shift with multiple observations, couple of observation finding

```
pred_1ob = model( torch.from_numpy(np.array([
   observations[curr_obs_num ,:]])) , torch.from_numpy(
   times_array.reshape(200,1)) ).numpy()
pred_2ob = model( torch.from_numpy(np.array([
   observations[next_obs_num ,:]])) , torch.from_numpy(
   times_array.reshape(200,1)) ).numpy()
diff = abs(pred_1ob[:,0]-pred_2ob[:,0]) #computes the
   couples distances on the y predicted
index_min=np.argmin(diff) # saves the closest couple
```

Listing 5.11: Extract of code for task shift with multiple observations, couple of observation interpolating

```
# calculate the fraction of interpolation
fraction = (curr_time - observations[curr_obs_num ,0]) /
   (observations[next_obs_num ,0] - observations[
   curr_obs_num ,0])
#interpolated_param = start_param + fraction * (
   end_param - start_param)
observation_timestep_matrix[time_index ,1+i]=start_param[
   i]+ fraction * (end_param[i] - start_param[i])
```

finding the two closest points. In the code extract, this part is visible in listing 5.10. This operation can be done once per couple of observations.

Once the closest couple of points is found in the trajectories, the points among them will not bias the task parameter change. For this reason, the start and end obtained will be interpolated in the time period of the transition. The final interpolated point obtained for this timestep is the desired observation with the non-biasing position and the mixed TP. In the listing 5.11, it is possible to see how, in the current timestep, the interpolation is executed among the two previously defined points. The resulting conditioning point is inserted in the matrix after the current timestep.

The plottings of the matrix depicted in fig. 4.14 and fig. 4.15 are achieved again with the *Matplotlib* python library. The relative code will not be reported as well because it's not necessary for the sake of the result but only to visualize the correctness of the procedure.

Listing 5.12: Extract of code for task shift with multiple observations, using the matrix obtained

```
for time_index, time in enumerate(matrix_times):
    prediction = model( torch.from_numpy(np.array([
        matrix_observations[time_index] ])) , torch.
        from_numpy(np.array([[time]])) ).numpy()
    predicted_Y[time_index] = prediction[:,:d_y]
    predicted_std[time_index] = np.log(1+np.exp(
        prediction[:,d_y:]))
```

Building the function to query the network with the matrix obtained. The second key part uses the matrix built before to query the network at every timestep with the changing observations. It is responsible for the final results obtained and visualized in fig. 4.16.

This part, instead of passing to the network the array of multiple observations, passes the computed array with a single observation of every time step. In the listing 5.12, it is possible to see how the model is fed with the observation at *time_index* and the current *time*. The matrix previously built is split in the array *matirx_time* (first column of the matrix), and the matrix of observations for every time step *matrix_observations*.

The predicted output value is saved in an array and plotted, as in the results in fig. 4.16.

Listing 5.13: Extract of code for building the graph of skills concatenated

```
num_traj=len(params_array)**(max_depth+1)
lengh_time=time_steps*(max_depth+1)
full_graph = np.zeros((0,lengh_time)) # initially empty
```

5.2 End-To-End Skill Concatenation

In this section, the key parts of the implementation for concatenating trajectories are introduced. The training of the network on the actions performed has already been explained, so it will be skipped.

There are two important parts worth highlighting: the first one is the construction of the graph of all possible actions, and the second one is the filtering of the ones that are not executable.

5.2.1 Building the graph of concatenated trajectories

The first key part is building the graph of the whole possible trajectories obtained from a starting point and combining multiple skills. This code is responsible for the visualization of the results in the fig. 4.20.

Initially, as it is possible to see in the listing 5.13, the empty structure is defined, and row by row will be added later. The final number of trajectories is calculated as explained in chapter 4 elevating the number of actions available to the power of the number of times they can be concatenated. This will also determine the full-time length of the execution.

Next, the recursive function will take care of filling the graph for each step. It's possible to see the most important lines in the listing 5.14.

The model is queried for every action given a conditioning point based on the result of the previous one. The starting point and end point of the actions predicted are visualized, and then the predictions are added at the end of the new line to add to the graph. Finally, for every action, the function will call itself again. The parameter new line this time now has all the action predictions in the end.

If the end time is reached, the new line containing the newly generated trajectory is added to the graph. No more recursive calls will be performed in this case.

5.2.2 Pruning the graph of concatenated trajectories

The second important part of the method is to remove the movement primitives that are unfeasible or not meaningful. To remove these trajectories, the jumps in

Listing 5.14: Extract of code for filling the graph of skills concatenated

```
# prevision, given a starting point
predicted_Ys,predicted_std=predict_model(np.array([np.
    concatenate(([0.0],params_array[i],[starting_pt]))]),
    time_queries)
print("%.2f to %.2f"%(predicted_Ys[0],predicted_Ys[-1]))
print("filling form %i to %i"%(current_depth*
    time_queries.shape[0],(current_depth+1)*time_queries.
    shape[0]))
#modify line
new_line[ 0, current_depth*time_queries.shape[0]:(
    current_depth+1)*time_queries.shape[0] ] =
    predicted_Ys.T
#recursive call to further modify
full_graph=add_trajectories(predicted_Ys[-1,0],
    current_depth+1, max_depth, full_graph, new_line)
```

Listing 5.15: Extract of code for filling the graph of skills concatenated

```
def remove_sequences_with_large_jumps(graph, threshold):
    valid_sequences = []
    for sequence in graph:
        diffs = np.abs(np.diff(sequence))
        if not np.any(diffs > threshold):
            valid_sequences.append(sequence)
```

the trajectories that exceed a certain threshold will make the trajectory discarded.

In listing 5.15, it is possible to see how the graph is passed to the function and a new empty one is built. For each sequence in the original graph, the differences among the elements are computed as an array.

Next, the array is converted to positive values with the absolute function.

If none of these differences exceed the threshold desired, the sequence is added to the graph of valid sequences.

Chapter 6

Validation and Testing

This chapter presents the validation and testing of the methods that were before designed and then implemented in this research. The accuracy of the results has already been discussed with the error comparison tables in the chapter 4. To demonstrate the applicability of this research, instead, the methods are tested on real-life robots, chapter 3.

To achieve these results, some minor adaptations had to be made. For example, the number of input or output dimensions of the network changed to accommodate the joint space of the robots since the monodimensional example given in the design is not enough.

Furthermore, the time scale for the movement primitives has been changed to a longer time period. The trajectories generated if in cartesian space, were passed to an inverse kinematic algorithm to retrieve the joint space.

Overall, these variations don't impact what has been discussed and implemented before.

6.1 Partial Skill Combination

To test the capabilities of the method developed, the robot selected is the UR10. It is worth noting that the research is not designed specifically for this robot and can be applied to any robot with different specifications.

The UR10 robot in the setup described in the chapter 3 has two main parts: the robot arm that executes the movements and the gripper that grasps the desired object.

UR10 robot interface For the sake of completeness, this paragraph briefly describes the Python interface developed to simplify the use of the robot in general and enable this experiment. This interface is available in a GitHub public repository [21]. The interface acts as a bridge between the user commands and the ROS topics exposed by the robot's computer.

It implements a series of ROS Subscribers to retrieve the desired data published by the robot and save it into buffer variables. Some functions are present to get the variables about the current status of the robot, this makes the read operations fast and not blocking.

Also, some ROS Publisher nodes are implemented to send commands and information to the robot. These nodes publish data to the robot's defined topics in order to move the device or set the desired settings. Functions are also present here to simplify the calls and send the ROS messages through the nodes implemented.

3F Gripper interface Another codebase repository used in this experiment has been developed to easily interface the gripper to the user and ease its operation.

This interface is available in a GitHub public repository [20]. The code has been developed separately since the gripper is a different entity from the robot and has its own ROS services and IP address. The interface also acts as a connecting link between the Python commands and the ROS services made available by the gripper integrated circuit.

In this case, since the gripper doesn't offer topics but services (see chapter 3) some ServiceProxies are implemented to simplify the operation of this last. The interface offers functions that ease the call of these services to open, close, set the aperture or force of the gripper, get the position, and so on.

Trajectories recorder and playback Finally, these two interfaces were used to build a code that allows recording the trajectory in time of the robot's joint positions. Furthermore, the trajectory recorded allows the recording of the cartesian position of the end effector and the gripper aperture.

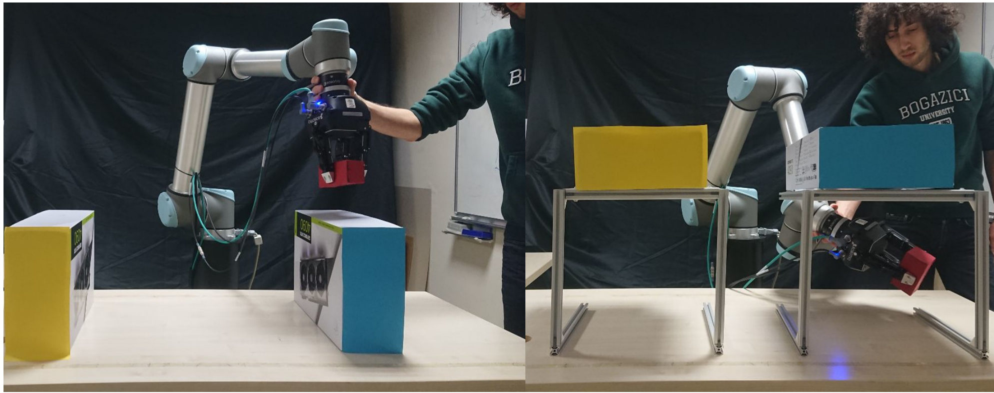

Figure 6.1: Some moments from learn by demonstration. On the left, action to overcome obstacles. On the right, action to pass under a tunnel.

Moreover, for our experiment, also the code for playing back these trajectories has been written. This work allows the testing in real life of the trajectories generated by the CNMPs architectures developed before.

Testing The robot was initially used to record some trajectories. This is part of the initial teaching of the "learn by demonstration" process. The trajectories were recorded with the previous code repository developed.

In fig. 6.1, it is possible to see two moments of this process. On the left image, the capture of an instant from the first action demonstration on how to overcome obstacles. In the right image, another action is taught on how to pass under a tunnel.

The UR10 robot has 6 Degrees of Freedom (DoF), so the network input is expanded to 6 dimensions. The choice of joint space is motivated by not dealing later with the inverse kinematics to reproduce trajectories from cartesian space. The six dimensions, one for every joint, of the two trajectories recorded are visible in fig. 6.2.

In the figure, the first action of passing over the obstacle is represented by the blue line in the graphs, it indicates the movement of every joint in time. Similarly, the action of passing through a tunnel is visible as the orange line for the same joints in time.

After training the network with the two trajectories and obtaining a satisfactory result, the method proposed is applied. The output trajectory generated is visible in the grey line of every graph.

It is worth noting that it is not a simple linear transition, but the network captures the non-linearity dependencies across the trajectories and interpolates them accordingly. This results in a mixture of skills that is coherent and maintains

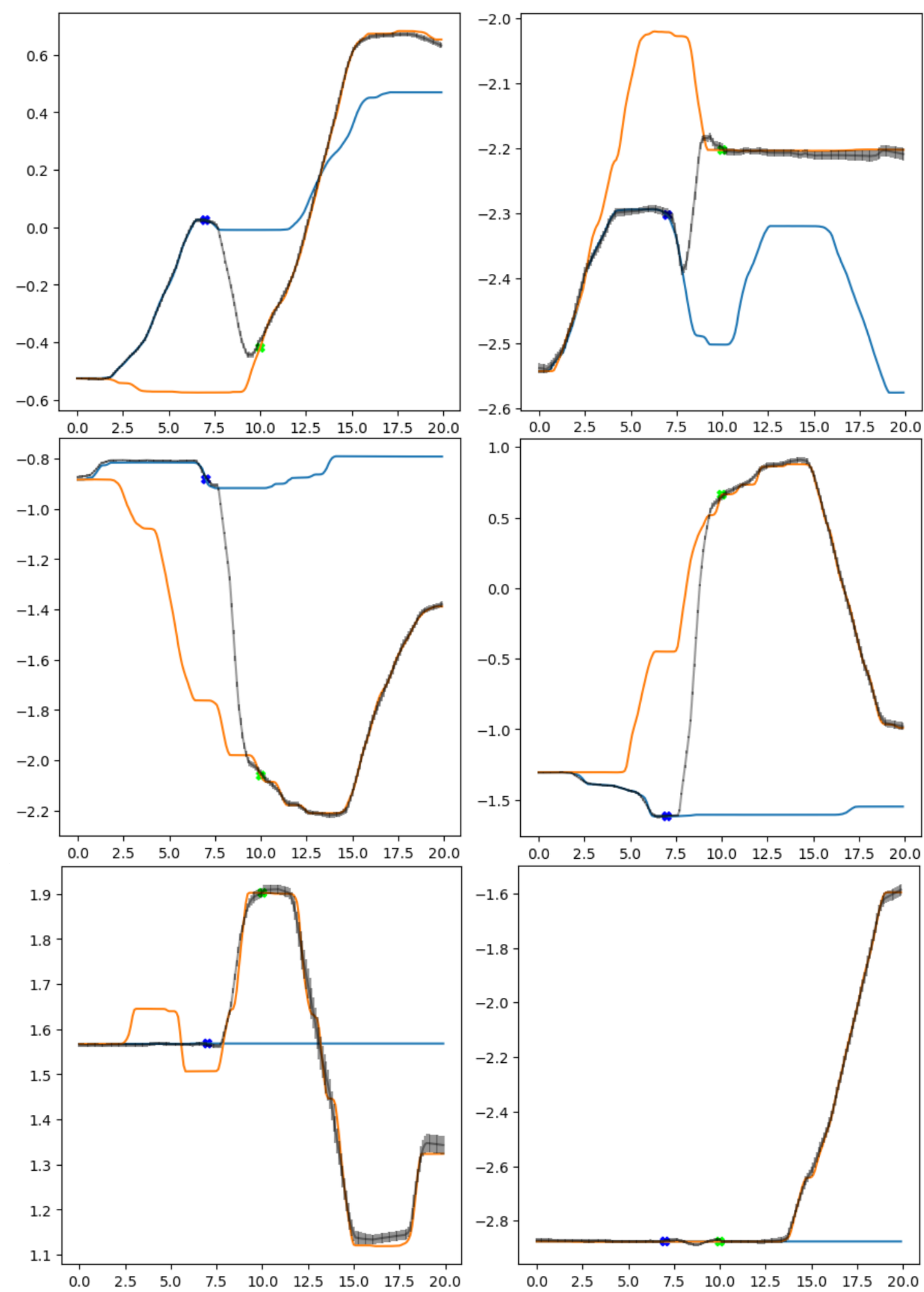

Figure 6.2: The 6 dimensions of a recorded trajectory on a real robot and the resulting mixed trajectory in grey generated by the network.

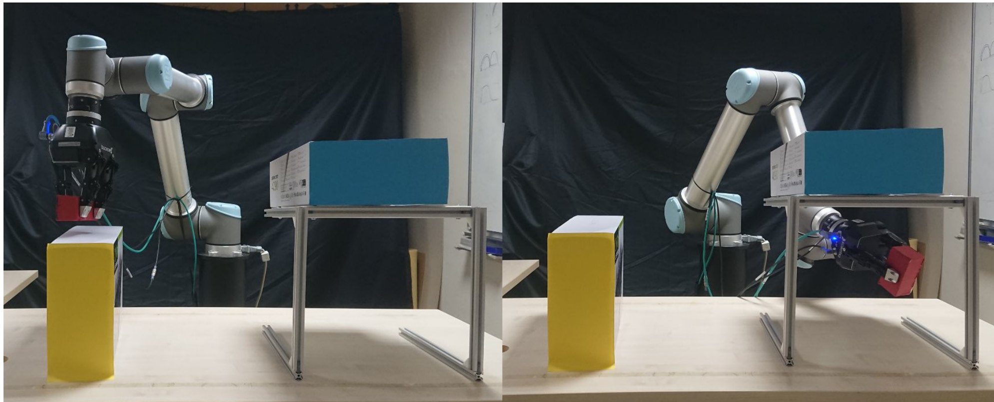

Figure 6.3: Some instants from the playback of the trajectory generated where the robot successfully completes the combination of the two actions.

the properties of the actions.

In the graph where the blue joint executes two curves, it is possible to notice this behavior. The trajectory generated does not pass directly from one conditioning point to the other. The final trajectory keeps the properties of the first blue one, descending for a while and then mixing with the orange one.

Finally, the trajectory generated is reproduced with the playback code developed. The robot successfully combines the actions and brings the object to the final destination without colliding with the environment. Some moments of the executions are visible in the fig. 6.3. The robot initially overcomes the obstacles with the first part of the action taught, then proceeds to successfully pass through the tunnel, changing its position and orientation.

6.2 End-To-End Skill Concatenation

For the End-To-End Skill Concatenation, the robot used for real-life testing is Baxter Robot, chapter 3. It is crucial to highlight that the research is not exclusive to this specific robot though. The platform has been used only to demonstrate the real applicability and efficacy of the work presented.

Moreover, part of the choice to use this platform instead of UR10 is indeed to prove that the models and methods developed are platform-independent and can be implemented on any robot.

Baxter robot has a complete set of interaction and Learning from Demonstration capacities. Moreover, the manipulation capabilities, although limited by the two-finger parallel grippers, are easier to integrate with the expert demonstrations and trajectory recordings. The robot has two buttons on the hands that are easy to press and link with the gripper position. The end effectors are light and simple to move.

This part of the research implies the concatenation of primitives, so many different demonstrations. Furthermore, the skills are related to the environment and manipulation of objects, so it's crucial to have a platform that is easy to interact with. For this reason, the Baxter robot has been selected for this research demonstration.

Baxter robot interface Another repository implemented along with the previous ones for the experiments in real life is the interface for the Baxter robot in Python 3. This work was driven by the need for Python 3 compatible functions since Baxter has an interface, but it is only Python 2, and it's not compatible with the majority of ML frameworks nowadays. This interface is available in a GitHub public repository [18].

Baxter exposes a ROS system as well, with ROS topics and services. For this reason, the interface uses nodes and ServiceProxies to send messages or call actions in the robot. To speed up the reading process, there are buffer variables for the data that the robot publishes. This has been especially useful in the recording stage since it increased the resolution of the recordings from 10Hz to 100Hz.

The robot exposes plenty of options, being designed for collaborative tasks. It has the possibility to set the joint position, to set the cartesian position of the end effector, and to retrieve these last ones. Moreover, it's possible to set and get the gripper position or the display image, or read the data from the infrared sensors and cameras in the hands.

All these options are available in the Python 3 interface developed. Moreover, some examples are present of how to move the robot or control from the keyboard, how to get the data from the sensors and cameras, and how to use the inverse kinematic services and the grippers.

Trajectories recorder and playback Lastly, in the interface developed previously, it is possible to find also the code to record the trajectories of the arms of the robot and play them back.

The recorder waits for the pressing of a button and starts to save current time and the position of the joints at the frequency desired. Moreover, the recorder also saves the end effector cartesian position and orientation for later comparison.

The gripper position and force read are also saved in the data for every timestep. This is useful for manipulation tasks that require handling objects.

Furthermore, a trajectory visualizer for cartesian and joint space has been developed as well. The code enables the user to see the movement primitive in 3D and check the correctness of the data recorded or about to be played. The 3D visualization of the trajectory can be optionally surrounded by five graphs for each individual dimension of the cartesian or joint space. The color of the points in 3D corresponds to the gripper aperture.

Lastly, the complementary code for the playback of trajectories enables the robot to move precisely from the data provided. The trajectories can be previously recorded or generated with a network. For this reason, this code was especially useful in testing this research.

Baxter detecting and reaching objects Another code repository was developed to complement the research done with a method that can bring the robot actuator to the initial position. In the discussion presented, the robot arm has to be in an initial state from which possible and meaningful actions are generated. Here, we briefly propose a method that can reach the initial state. This will make the demonstration more complete from the side of a spectator and independent from an expert who has to guide the robot in the initial state.

The code and the neural network model weights are available in a GitHub public repository [17].

The method developed consists of putting the robot arm in a pose from which the cameras in the hands are leveraged to acquire the RGB image of the table below. Subsequently, the objects present will be detected with the use of a neural network for object detection, namely YOLO. The robot will use the given positions of the objects to move the arm slightly toward the one desired. The inverse kinematics service of the robot takes care of computing the pose for the new cartesian position. The process repeats till the infrared (IR) sensor of the hand detects the distance of the object as "graspable". At this point, the gripper closes, and the object is reached for further desired manipulation.

This code has been used in the research as a viable way to get to the initial pose and the object. Many other more complicated approaches are possible. Reached the initial state, then demonstrated the method presented are demonstrated.

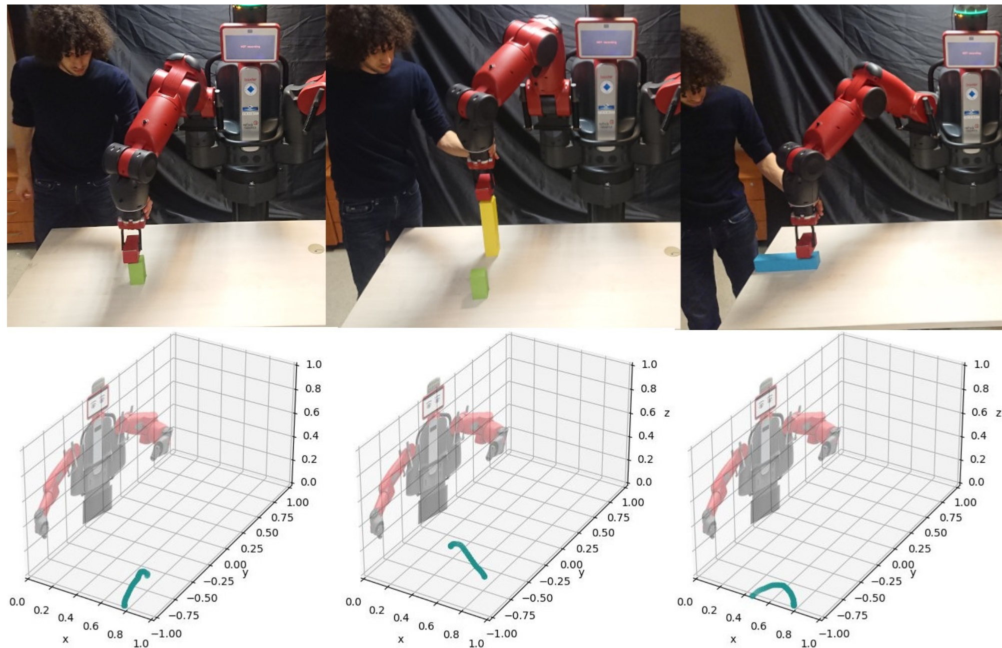

Figure 6.4: Teaching by demonstration three different actions. From left to right: move from the ground to the green block, move on to the yellow block, and overcome the blue obstacle.

Testing For testing the method, the trajectories are initially shown to the robot. The recorder developed provided an easy way to save the data.

For this test, the choice was to record the trajectories in cartesian space. There is no substantial difference since the network will learn multiple dimensions anyway, but to demonstrate this method is valid also for cartesian space, this time, the network got trained on those trajectories.

Furthermore, the three dimensions plus four dimensions for the orientation of the hand sum up to the seven dimensions, and Baxter has 7 Degrees of Freedom (DoF) as well.

For these reasons, the network input is expanded to 7 dimensions in both cases, but using the cartesian space, the inverse kinematic service of the robot has to be used.

The trajectories recorded are visible in fig. 6.4. On the left, it depicts an action to move the box from a position on the table to the top of the green box. In the center, there is a movement primitive that brings the red box from the green one to the top of the yellow one. Lastly, on the right, there is a movement that overcomes the blue box and reaches a new position.

It has to be highlighted that the actions are demonstrated separately, and they are independently shown to the network.

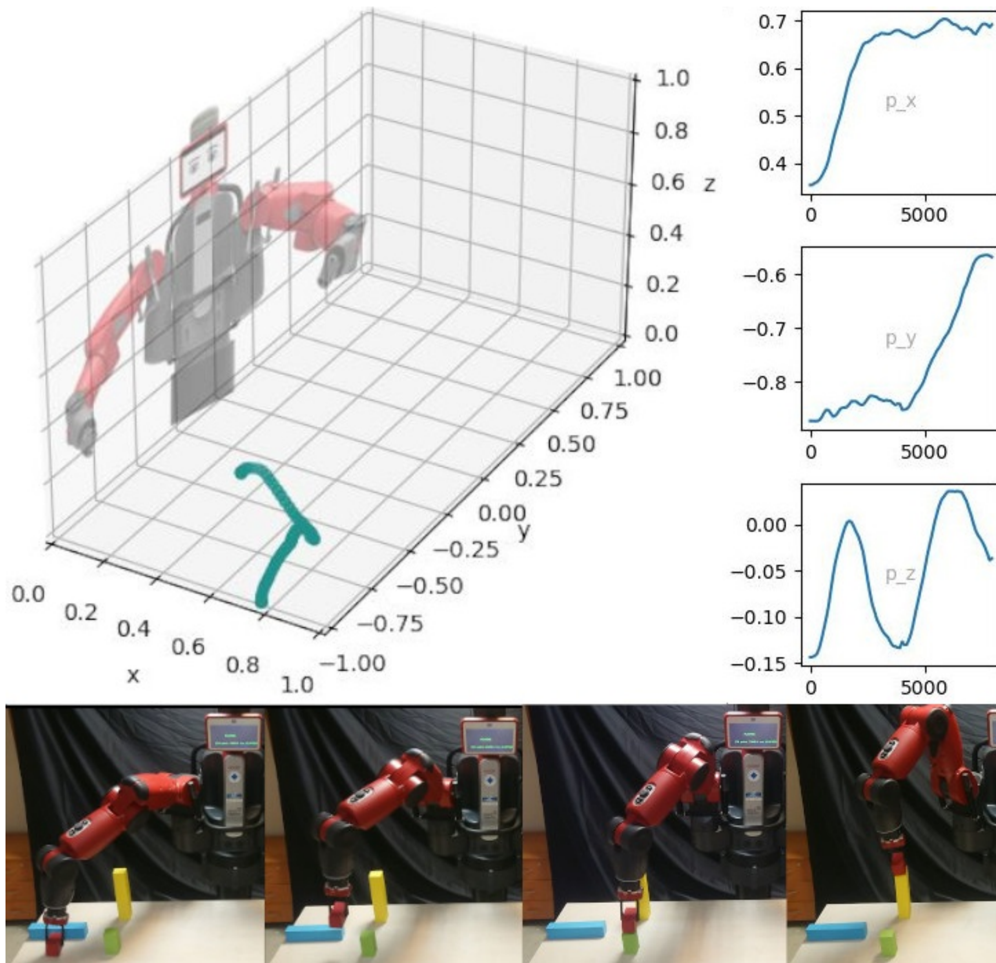

Figure 6.5: Results of the concatenation method given a starting point in front of the blue box and end point on the yellow box

The proposed method is used after training the network and obtaining satisfactory performance. Two examples are presented below to demonstrate that the method is not bonded to time or a specific order of actions.

In the first example, in fig. 6.5, the robot reaches the red box. This is the starting position and the condition of the network. Subsequently, all possible actions are generated, and their end position is used to concatenate the next level of possible actions. The process continues until one finds the final state requested, which is the top of the yellow box.

The results are shown in fig. 6.5, the cartesian trajectories on the right are combined together, and the fast jumps are not present. The relative 3D representation is shown in the 3D plot. When executed with the trajectory playback,

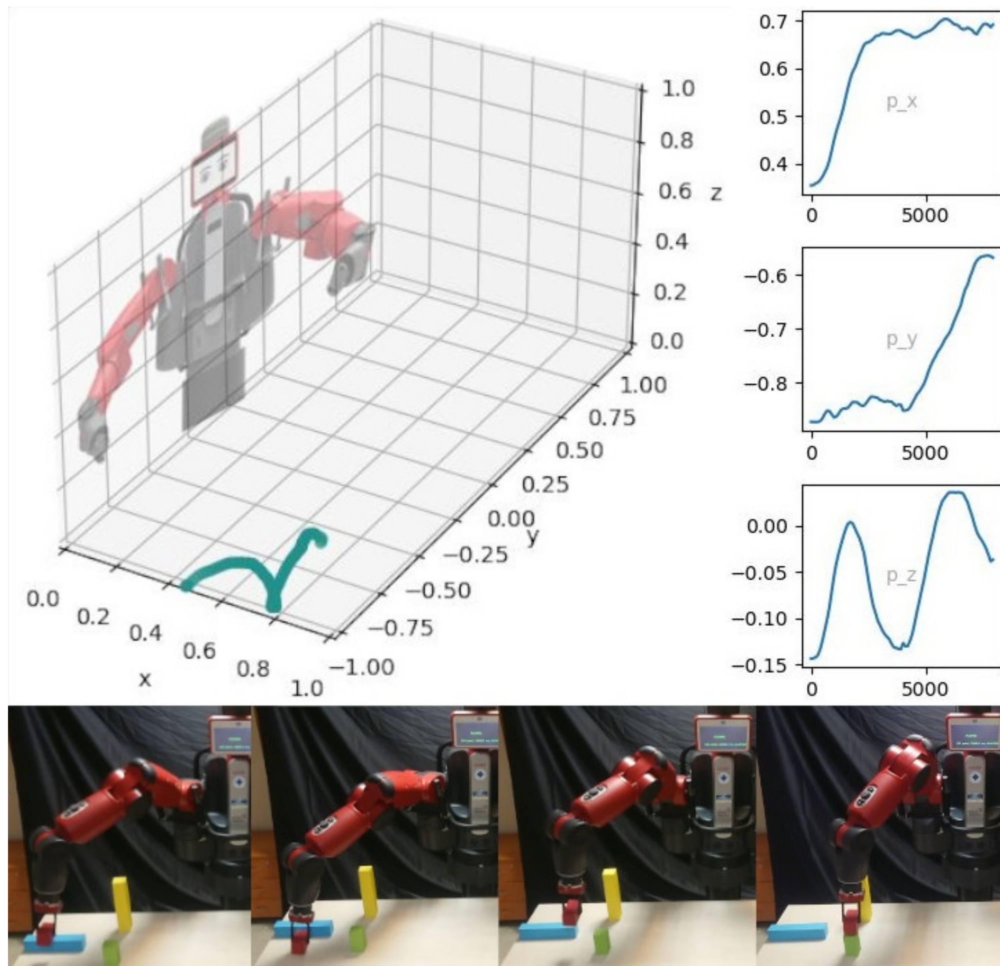

Figure 6.6: Results of the concatenation method given a starting point in the back of the blue box and end point on the green box

the robot performs the action correctly and reaches the goal, combining the ones learned.

In the second example, in fig. 6.6, another starting point and another goal is given. The method performs the search, and the resulting combination of cartesian trajectories is visible on the right part of the figure. The 3D trajectory is visible in the plot.

In this case, the end and beginning of different actions also match without abrupt jumps. Finally, the trajectories generated are reproduced. The robot executes the combinations of actions to bring the box from the starting point to the goal state.

Chapter 7

Conclusions

This thesis provides a possible approach to novel high-level skill generation by combining movement primitives learned by CNMP models.

The main key findings of the study are two methods developed to synthesize new actions from demonstrated ones.

In the first approach, parts of trajectories are blended thanks to the combination of the task interpolation ability of the neural network analyzed and the mathematical system developed to pass the proper parameters to it.

Furthermore, two different architectural changes have been proposed to the classic CNMP model. The two different architectures achieve similar results compared to the original model, but they enable its use with partial information.

The second approach presented achieved action synthesis thanks to the concatenation of primitives combined with the spatial interpolation of the network and the ability to encode multidimensional data.

The approaches are not free from limitations, the main ones discovered are the need for an initial condition and the awareness of the environment.

Finally, the practical applications have been investigated, and both methods proposed performed well on the tests on real-life robots.

7.1 Future work

Possible future works are multiple since the research touched extensively many topics. This research leaves many challenges open to tackle, from planning to a better world representation, perception, and grasping.

Another area in which the network output can be used is navigation. The trajectories analyzed in this study focused on robotics manipulation but can also be extended to robotic navigation.

A possible research in the CNMP with the task parameters only in query would

be to condition it with one state without parameters and let the network, queried with different tasks, pass through that state.

Further research is needed on the same network developed for better training to improve the results in the interpolation dimensions. For example, conditioning it with task parameters between the original ones during training. This will force mixed task parameters trajectories to pass through the designated point.

Moreover, it's possible to extend the research to use the task parameter changing capabilities to shift to another meaningful plan if one fails. The same abilities could be used to change among continuous actions.

In the second part, some further optimization of the graph building can be achieved. More intelligent methods of research and pruning could be developed.

Furthermore, actions of different time lengths in the concatenation procedure can be integrated.

Finally, an excellent addition would be extending the simple low-dimensional world representation given to the network in the second method with an environment image.

Bibliography

[1] Alper Ahmetoglu, M. Yunus Seker, Justus Piater, Erhan Oztop, and Emre Ugur. Deepsym: Deep symbol generation and rule learning for planning from unsupervised robot interaction. *Journal of Artificial Intelligence Research*, 75:709–745, November 2022.

[2] Anaconda. Anaconda. `https://www.anaconda.com`, 2023. [Online; accessed 20-November-2023].

[3] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

[4] Luigi Biagiotti and Claudio Melchiorri. *Trajectory planning for automatic machines and robots*. Springer Science & Business Media, 2008.

[5] Sylvain Calinon. A tutorial on task-parameterized movement learning and retrieval. *Intelligent service robotics*, 9:1–29, 2016.

[6] Vivian Chu, Ian McMahon, Lorenzo Riano, Craig G McDonald, Qin He, Jorge Martinez Perez-Tejada, Michael Arrigo, Naomi Fitter, John C Nappo, Trevor Darrell, et al. Using robotic exploratory procedures to learn the meaning of haptic adjectives. In *2013 IEEE International Conference on Robotics and Automation*, pages 3048–3055. IEEE, 2013.

[7] Andreas Damianou and Neil D Lawrence. Deep gaussian processes. In *Artificial intelligence and statistics*, pages 207–215. PMLR, 2013.

[8] Gedeon O Deak. The development of cognitive flexibility and language abilities. *Advances in child development and behavior*, 31:273–328, 2003.

[9] Chad DeChant and Daniel Bauer. Toward robots that learn to summarize their actions in natural language: a set of tasks. In *5th Annual Conference on Robot Learning, Blue Sky Submission Track*, 2021.

[10] Massimo Egidi. Decomposition patterns in problem solving. *Contributions to Economic Analysis*, 280:15–46, 2006.

[11] Marta Garnelo, Dan Rosenbaum, Chris J. Maddison, Tiago Ramalho, David Saxton, Murray Shanahan, Yee Whye Teh, Danilo J. Rezende, and S. M. Ali Eslami. Conditional neural processes. *CoRR*, abs/1807.01613, 2018.

[12] Marta Garnelo, Jonathan Schwarz, Dan Rosenbaum, Fabio Viola, Danilo J Rezende, SM Eslami, and Yee Whye Teh. Neural processes. *arXiv preprint arXiv:1807.01622*, 2018.

[13] Alessandro Gasparetto and V Zanotto. A new method for smooth trajectory planning of robot manipulators. *Mechanism and machine theory*, 42(4):455–471, 2007.

[14] Abhishek Gupta, Vikash Kumar, Corey Lynch, Sergey Levine, and Karol Hausman. Relay policy learning: Solving long-horizon tasks via imitation and reinforcement learning, 2019.

[15] Thomas E Horton, Arpan Chakraborty, and Robert St Amant. Affordances for robots: a brief survey. *AVANT. Pismo Awangardy Filozoficzno-Naukowej*, 2:70–84, 2012.

[16] Hengyuan Hu, Denis Yarats, Qucheng Gong, Yuandong Tian, and Mike Lewis. Hierarchical decision making by generating and following natural language instructions. *Advances in neural information processing systems*, 32, 2019.

[17] Igor Lirussi. Baxter object detection and grasping. `https://github.com/igor-lirussi/Baxter-Robot-ObjDet`, 2023. [Online; accessed 03-December-2023].

[18] Igor Lirussi. Baxter python interface. `https://github.com/igor-lirussi/baxter-python3`, 2023. [Online; accessed 30-November-2023].

[19] Igor Lirussi. Cnmp-robotic-skill-synthesis. `https://github.com/igor-lirussi/CNMP-Robotic-Skill-Synthesis`, 2023. [Online; accessed 04-December-2023].

[20] Igor Lirussi. Three finger gripper python interface. `https://github.com/igor-lirussi/Gripper3F_interface`, 2023. [Online; accessed 30-November-2023].

[21] Igor Lirussi. Ur10 python interface. `https://github.com/igor-lirussi/UR10_robot_interface`, 2023. [Online; accessed 30-November-2023].

[22] Auke Ijspeert, Jun Nakanishi, and Stefan Schaal. Learning attractor landscapes for learning motor primitives. *Advances in neural information processing systems*, 15, 2002.

[23] O. Deussen J. Görtler, R. Kehlbeck. A visual exploration of gaussian processes. In *Proceedings of the Workshop on Visualization for AI Explainability (VISxAI)*, 2018.

[24] Lorenzo Jamone, Emre Ugur, Angelo Cangelosi, Luciano Fadiga, Alexandre Bernardino, Justus Piater, and José Santos-Victor. Affordances in psychology, neuroscience, and robotics: A survey. *IEEE Transactions on Cognitive and Developmental Systems*, 10(1):4–25, 2018.

[25] Jupyter Notebook. Jupyter notebook. `https://jupyter.org`, 2023. [Online; accessed 20-November-2023].

[26] Ashish Kapoor, Kristen Grauman, Raquel Urtasun, and Trevor Darrell. Gaussian processes for object categorization. *International journal of computer vision*, 88:169–188, 2010.

[27] By A Karmiloff-Smith. Beyond modularity: A developmental perspective on cognitive science. *European journal of disorders of communication*, 29(1):95–105, 1994.

[28] Alexander Khazatsky, Ashvin Nair, Daniel Jing, and Sergey Levine. What can i do here? learning new skills by imagining visual affordances. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 14291–14297. IEEE, 2021.

[29] Hyun-Chul Kim and Jaewook Lee. Clustering based on gaussian processes. *Neural computation*, 19(11):3088–3107, 2007.

[30] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[31] Jens Kober and Jan Peters. Policy search for motor primitives in robotics. *Advances in neural information processing systems*, 21, 2008.

[32] George Konidaris. On the necessity of abstraction. *Current opinion in behavioral sciences*, 29:1–7, 2019.

[33] Oliver Kroemer, Scott Niekum, and George Konidaris. A review of robot learning for manipulation: Challenges, representations, and algorithms. *The Journal of Machine Learning Research*, 22(1):1395–1476, 2021.

[34] Dongheui Lee and Christian Ott. Incremental kinesthetic teaching of motion primitives using the motion refinement tube. *Autonomous Robots*, 31:115–131, 2011.

[35] Monica Maranesi, Luca Bonini, and Leonardo Fogassi. Cortical processing of object affordances for self and others' action. *Frontiers in psychology*, 5:538, 2014.

[36] Katharina Mülling, Jens Kober, Oliver Kroemer, and Jan Peters. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research*, 32(3):263–279, 2013.

[37] Duy Nguyen-Tuong, Matthias Seeger, and Jan Peters. Model learning with local gaussian process regression. *Advanced Robotics*, 23(15):2015–2034, 2009.

[38] François Osiurak, Yves Rossetti, and Arnaud Badets. What is an affordance? 40 years later. *Neuroscience & Biobehavioral Reviews*, 77:403–417, 2017.

[39] Alexandros Paraschos, Christian Daniel, Jan R Peters, and Gerhard Neumann. Probabilistic movement primitives. *Advances in neural information processing systems*, 26, 2013.

[40] Peter Pastor, Heiko Hoffmann, Tamim Asfour, and Stefan Schaal. Learning and generalization of motor skills by learning from demonstration. In *2009 IEEE International Conference on Robotics and Automation*, pages 763–768. IEEE, 2009.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[42] Pytorch. Tensors and dynamic neural networks in python with strong gpu acceleration. `https://pytorch.org`, 2023. [Online; accessed 20-November-2023].

[43] Harish Ravichandar, Athanasios S Polydoros, Sonia Chernova, and Aude Billard. Recent advances in robot learning from demonstration. *Annual review of control, robotics, and autonomous systems*, 3:297–330, 2020.

[44] Robotiq. 3-finger adaptive robot gripper. `https://robotiq.com/products/3-finger-adaptive-robot-gripper`, 2023. [Online; accessed 20-November-2023].

[45] Robotiq. Ft 300-s force torque sensor. `https://robotiq.com/products/ft-300-force-torque-sensor`, 2023. [Online; accessed 20-November-2023].

[46] ROS. Robot operating system. `https://www.ros.org`, 2023. [Online; accessed 20-November-2023].

[47] Eric Rosen, Ben M Abbatematteo, Skye Thompson, Tuluhan Akbulut, and George Konidaris. On the role of structure in manipulation skill learning. In *CoRL 2022 Workshop on Learning, Perception, and Abstraction for Long-Horizon Planning*, 2022.

[48] Hugh Salimbeni and Marc Deisenroth. Doubly stochastic variational inference for deep gaussian processes. *Advances in neural information processing systems*, 30, 2017.

[49] Matteo Saveriano, Fares J Abu-Dakka, Aljaž Kramberger, and Luka Peternel. Dynamic movement primitives in robotics: A tutorial survey. *The International Journal of Robotics Research*, page 02783649231201196, 2021.

[50] Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

[51] Stefan Schaal. Dynamic movement primitives-a framework for motor control in humans and humanoid robotics. In *Adaptive motion of animals and machines*, pages 261–280. Springer, 2006.

[52] Matthias Seeger. Gaussian processes for machine learning. *International journal of neural systems*, 14(02):69–106, 2004.

[53] Muhammet Yunus Seker, Mert Imre, Justus Piater, and Emre Ugur. Conditional neural movement primitives. In *Proceedings of Robotics: Science and Systems*, FreiburgimBreisgau, Germany, 6 2019.

[54] Anthony Simeonov, Yilun Du, Beomjoon Kim, Francois Hogan, Joshua Tenenbaum, Pulkit Agrawal, and Alberto Rodriguez. A long horizon planning framework for manipulating rigid pointcloud objects. In *Conference on Robot Learning*, pages 1582–1601. PMLR, 2021.

[55] Edward Snelson and Zoubin Ghahramani. Sparse gaussian processes using pseudo-inputs. *Advances in Neural Information Processing Systems*, 18:1259–1266, 2006.

[56] Emre Ugur and Hakan Girgin. Compliant parametric dynamic movement primitives. *Robotica*, 38(3):457–474, 2020.

[57] Universal Robots. Ur10 (robot) — medium duty industrial collaborative robot. https://www.universal-robots.com/products/ur10-robot/, 2023. [Online; accessed 20-November-2023].

[58] Wikipedia contributors. Baxter (robot) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Baxter_(robot)&oldid=1183931177, 2023. [Online; accessed 19-November-2023].

[59] Andrew Gordon Wilson, Zhiting Hu, Ruslan Salakhutdinov, and Eric P Xing. Deep kernel learning. In *Artificial intelligence and statistics*, pages 370–378. PMLR, 2016.