2019

Fast Online Model Learning for Controlling Complex Real-World Robots

Loviken, Pontus

http://hdl.handle.net/10026.1/15078

http://dx.doi.org/10.24382/458

University of Plymouth

All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.
This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's prior consent.
Fast Online Model Learning for Controlling Complex Real-World Robots

By

PONTUS LOVIKEN

A thesis submitted to the University of Plymouth in partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

School of Computing, Electronics and Mathematics

SEPTEMBER 2019
AUTHOR'S DECLARATION

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award without prior agreement of the Doctoral College Quality Sub-Committee.

Work submitted for this research degree at the University of Plymouth has not formed part of any other degree either at the University of Plymouth or at another establishment.

This work has been carried out by Pontus Loviken under the supervision of Dr. Nikolas Hemion, Prof. Dr. Angelo Cangelosi, Dr. Alban Laflaquière and Dr. Michael Spranger. The work was funded by the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 674868 (APRIL).

Publications:

• P. Loviken and N. Hemion, *Online-learning and planning in high dimensions with finite element goal babbling*, in 2017 Joint IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob), pp. 247-254. DOI: https://doi.org/10.1109/DEVLRN.2017.8329814


Word count of main body of thesis: 54,849

SIGNED: ............................................................... DATE: ..........................................

i
FAST ONLINE MODEL LEARNING FOR CONTROLLING COMPLEX REAL-WORLD ROBOTS
Pontus Loviken

How can real robots with many degrees of freedom - without previous knowledge of themselves or their environment - act and use the resulting observations to efficiently develop the ability to generate a wide set of useful behaviours?

This thesis presents a novel framework that enables physical robots with many degrees of freedom to rapidly learn models for control from scratch. This can be done in previously inaccessible problem domains characterised by a lack of direct mappings from motor actions to outcomes, as well as state and action spaces too large for the full forward dynamics to be learned and used explicitly. The proposed framework is able to cope with these issues by the use of a set of local Goal Babbling models, that maps every outcome in a low dimensional task space to a specific action, together with a sparse higher level Reinforcement Learning model, that learns to navigate between the contexts from which each Goal Babbling model can be used. The two types of models can then be learned online and in parallel, using only the data a robot can collect by interacting with its environment.

To show the potential of the approach we present two possible implementations of the framework, over two separate robot platforms: a simulated planar arm with up to 1,000 degrees of freedom, and a real humanoid robot with 25 degrees of freedom. The results show that learning is rapid and essentially unaffected by the number of degrees of freedom of the robot, allowing for the generation of complex behaviours and skills after a relatively short training time. The planar arm is able to strategically plan series of motions in order to move its end-effector between any two parts of a crowded environment, within 10,000 iterations. The humanoid robot is able to freely transition between states such as lying on the back, belly, and sides, and occasionally also sitting up, within only 1,000 iterations. This corresponds to 30 – 60 minutes of real-world interactions.

The main contribution of this thesis is to provide a framework for solving a control learning problem, previously largely unexplored with no obvious solutions, but with strong analogies to, for example, early learning of body orientation control in infants. This thesis examined two quite different implementations of the proposed framework, and showed success in both cases for two different control learning problem.
# Table of Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acronyms</td>
<td>xi</td>
</tr>
</tbody>
</table>

## 1 Introduction
1.1 Outline .................................. 4

## 2 The task space control problem
2.1 Mathematical description ............. 8
   2.1.1 The robot black box ................ 8
   2.1.2 Forward dynamics .................... 9
2.2 Goal state based robot control ....... 9
   2.2.1 Postures and complete states ....... 9
   2.2.2 Posture control and goal states ..... 10
   2.2.3 Goal state control ................. 11
   2.2.4 Fully connected state sets .......... 13
2.3 The Control Learning Problem ......... 14
   2.3.1 The Control Problem ............... 15
   2.3.2 Challenges ....................... 15

## 3 Machine Learning Approaches ......... 17
3.1 Reinforcement Learning ................. 17
   3.1.1 Goal-conditioned RL ................ 18
   3.1.2 Model-free approaches .............. 18
   3.1.3 Model-based approaches ............ 21
3.2 Goal babbling .......................... 24
   3.2.1 Main features ...................... 24
   3.2.2 Challenges ....................... 25
   3.2.3 Nearest neighbour approach ......... 26
   3.2.4 Continuous manifold goal babbling .... 26
   3.2.5 Multi-Task Goal Babbling ........... 28
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The setup of the two robot problems investigated in this work</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>The assumed black box dynamics of the system</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Illustration of state spaces</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>The reach-by-accident approximation - when it works and when it fails</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Example and counterexample of fully connected state sets</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>A simple example of the proposed framework</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Conceptual comparison of two possible implementations of the proposed framework</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>Illustration of consistency</td>
<td>62</td>
</tr>
<tr>
<td>6.1</td>
<td>Set-up for robot arm control problem</td>
<td>70</td>
</tr>
<tr>
<td>6.2</td>
<td>Description of robot arm dynamics</td>
<td>71</td>
</tr>
<tr>
<td>6.3</td>
<td>Initialisation of robot arm</td>
<td>72</td>
</tr>
<tr>
<td>6.4</td>
<td>Overview of the home state approach, applied to robot arm problem</td>
<td>76</td>
</tr>
<tr>
<td>6.5</td>
<td>Reaching error for varying numbers of degrees of freedom (DoF)</td>
<td>77</td>
</tr>
<tr>
<td>6.6</td>
<td>Reaching error for different radii of reach-out regions</td>
<td>79</td>
</tr>
<tr>
<td>6.7</td>
<td>Number of home states vs. reach-out region radius</td>
<td>80</td>
</tr>
<tr>
<td>6.8</td>
<td>Reaching error distribution over room</td>
<td>80</td>
</tr>
<tr>
<td>6.9</td>
<td>Overview of the continuous manifold approach, applied to robot arm problem</td>
<td>81</td>
</tr>
<tr>
<td>6.10</td>
<td>Different room segmentations</td>
<td>85</td>
</tr>
<tr>
<td>6.11</td>
<td>Examples of learned reaching behaviours</td>
<td>86</td>
</tr>
<tr>
<td>6.12</td>
<td>Reaching error for different exploration strategies, using $6 \times 6$ segmentation</td>
<td>87</td>
</tr>
<tr>
<td>6.13</td>
<td>Reaching error for different exploration strategies, using $10 \times 10$ segmentation</td>
<td>88</td>
</tr>
<tr>
<td>6.14</td>
<td>Reaching error for different segmentations</td>
<td>88</td>
</tr>
<tr>
<td>6.15</td>
<td>Reaching error for different numbers of DoF</td>
<td>89</td>
</tr>
<tr>
<td>6.16</td>
<td>Reaching error for one longer run</td>
<td>89</td>
</tr>
<tr>
<td>6.17</td>
<td>Manifold smoothness for different numbers of DoF</td>
<td>90</td>
</tr>
<tr>
<td>6.18</td>
<td>Manifold smoothness for different segmentations</td>
<td>90</td>
</tr>
<tr>
<td>6.19</td>
<td>Comparison of results of the home state approach and the continuous manifold approach</td>
<td>92</td>
</tr>
</tbody>
</table>
7.1 Task and state spaces for the humanoid robot ................................................. 96
7.2 Set-up for body orientation control problem ..................................................... 99
7.3 Motor cog wear-down ......................................................................................... 103
7.4 Examples of discovered states by the humanoid robot ......................................... 104
7.5 The distribution of reached task space positions for every robot and radius .......... 105
7.6 Task space coverage of observed samples ........................................................... 106
7.7 The reaching error over time ............................................................................... 107
7.8 The reaching error distribution over the task space ............................................. 107
7.9 Example of a robot learning to roll around 360° after 30 minutes of training .... 108
7.10 Another robot able to make a 360° rotation ......................................................... 109
7.11 Modification of the robot in order to investigate the effect of changed morphology . 109
7.12 Task space distribution for altered vs. default morphology .................................. 109
7.13 Examples of states and skills discovered with altered morphology ...................... 110
7.14 Task space segmentation for continuous manifold approach for humanoid robot . 112
7.15 Examples of learned states for moving between the back and both sides ............ 115
7.16 Learning progress for moving between different task space regions .................. 115
7.17 Exploration and performance with and without ensured consistency ................. 116
AC  Artificial Curiosity. 30

CCW  counter-clockwise. 61, 91

CW  clockwise. 61, 91

DDPG  Deep Deterministic Policy Gradients. 20

DMP  Dynamic Motion Primitives. 29

DoF  degrees of freedom. ix, 2–5, 9, 12, 15, 20, 21, 23–25, 29, 31–33, 63, 69, 70, 72, 74–80, 84, 86, 88–92, 95, 96, 120–122

DQN  Deep Q-Networks. 20


GPR  Gaussian Process Regression. 63

HER  Hindsight Experience Replay. 20

IM  Intrinsic Motivation. 30

LLR  Local Linear Regression. 63, 65, 83, 92, 113

LR  Linear Regression. 83, 113, 116

MB  Motor Babbling. 114, 116

MDP  Markov Decision Process. 21, 39, 43, 44, 48–52, 54, 56–58, 66–68, 72, 73, 75, 76, 81, 82, 84, 86–88, 93, 113, 114, 116, 119, 120, 122, 123, 125, 127

RL  Reinforcement Learning. 17–21, 24, 30, 34, 35, 44, 48, 75, 119, 120, 124
Robots learning by themselves how to use their bodies in order to perform various skills is a long-standing aim of the field of “Developmental Robotics” [Asada et al., 2009; Cangelosi and Schlesinger, 2015; Nguyen-Tuong and Peters, 2011; Oudeyer et al., 2007; Schmidhuber, 2006]. Traditionally most robots rely on preprogrammed behaviours, designed by engineers with detailed models of the specific robot who are attempting to predict and account for any context the robot could find itself in. This is limiting in many ways. It restricts what types of robots can be used, since they need to be easily modelled [Ghosh et al., 2018; Renda et al., 2012]. It also means that a robot will only work reliably when operating within boundaries accounted for by the programmer. Move the robot to a new environment, or use a robot that is slightly damaged or worn down, and there is no guarantee that the preprogrammed behaviours will work anymore.

If robots could learn to use their bodies by themselves it would solve many of these problems while also opening up new possibilities. It would allow end-users to customise robot bodies to particular tasks and then let the robot figure out by itself how to best use the particular body to solve the given task Mohamed et al. [2008]; Rus [2015]. This principle could also be used in the development of the robot’s body to begin with, as it allows for a rapid evaluation of the capability of any particular morphology Cheney et al. [2018, 2014]; Sims [1994]. Furthermore, it would allow the introduction of a greater variety of robots on the commercial market. If robots can learn by themselves how to use their bodies in interaction with real, physical environments, it becomes less important whether a particular robot can be accurately modelled in a simulation or not. At the moment this is a major issue in the field of, for example, soft robotics [Duriez, 2013; Duriez and Bieze, 2017; Haddadin et al., 2018; Kim et al., 2013]. Self-learning robots finally hold the promise of leading to more robust functionality. Such robots could in principle be able to adapt to new contexts on the fly, including changes in the environment and to the robot itself.
Despite the potential advantages of self-learning robots it is, in practice, an unresolved question how to best do this on real complex robots, acting in a physical world Chatzilygeroudis et al. [2018]; Mouret [2016]. Most of today’s machine learning methods only focus on very narrow cases: only specific single tasks are learned, the robots are simulated with only a few degrees of freedom (DoF), the training time is more or less unlimited, the robot is able to return to any previous state at will, among other things. This is far from the case of real robots learning to act in a physical environment. Many real robots have a considerable number of DoF with vast state and action spaces. Actions are not necessarily reversible, meaning that a humanoid robot learning to walk but falls over needs to learn to stand up again before attempting another step. Learning time is usually extremely limited, as every action takes time and every action also poses the risk of damaging the robot in one way or another. This means that a robot that is training for too long might change its own dynamics so much that earlier data becomes useless. How can such a robot possibly learn relevant skills from scratch? This is the research question of this work. In short:

*How can real robots with many DoF independently, and without outsider interference, learn from scratch how to use their bodies in order to perform various skills, using only the limited data they are able to collect themselves through their lifetime?*

This question is closely related to the topic of “micro-data learning”, which can be seen as a counter reaction to otherwise dominating machine learning approach of “big data learning”, where vast amounts of data are collected in order to efficiently generalise for every situation an agent can find itself in Chatzilygeroudis et al. [2018]; Mouret [2016].

A first necessary step in investigating this question is to define what we mean by *skills*. From Wikipedia we find the following definition:

“A skill is the ability to carry out a task with determined results often within a given amount of time, energy, or both”.

A common way to translate this into a machine learning problem is to construct a measurement, often referred to as the *utility or loss function*, that measures to what extent a particular task is carried out Davidor [1991]; Sutton and Barto [2015]. This transforms the skill learning problem into an optimisation problem, where optimising the function translates into learning the skill. Walking can, for example, be expressed as the optimisation of the measurement of moving forward; grasping an object can be expressed as the minimisation of the distance between a robots hand and the object, etc.

A drawback of this approach is that it requires someone to define in advance what particular skill the robot should learn, and then handcraft a utility/loss function that can only be optimised by actually performing that skill. This is often challenging and opens up the risk of “reward hacking” Hadfield-Menell et al. [2017], where the robot finds unintended ways of optimising the function without actually learning the desired skill. A vacuum cleaner robot can for example
learn to eject collected dust in order to collect even more Russell and Norvig [2010]. This is closely related to Goodhart’s law which states: “When a measure becomes a target, it ceases to be a good measure” Strathern [1997]. Another drawback with the utility function approach is that it typically only allows the robot to learn skills to which utility/loss functions are explicitly given, prohibiting the discovery of other possible skills an agent might be well equipped to do Pong et al. [2018].

Going back to the definition of “skills” there is another interpretation that is potentially able to avoid these problems in many cases. Instead of viewing skills as the ability to carry out a predetermined task, skills themselves can be seen as something emerging from something more fundamental: the control over results Bechtle et al. [2016]; Mugan and Kuipers [2011]; Von Hofsten [2004]. Such an interpretation allows for a different approach. Instead of defining some measurement for how well a particular task is carried out, we define a task-space over which we want the robot to gain control Forestier and Oudeyer [2016]; Rolf et al. [2011]. Controlling such a space then naturally leads to the emergence of a whole set of skills that are instrumental in controlling that space. Control over one’s heads height over the ground necessitates the generation of skills such as standing up or lying down, the control over an object’s position in a room requires behaviours such as grasping, fetching, carrying, etc. Learning particular skills is thus not an end goal in itself, but rather the tools by which a robot is able to manipulate outcomes in a given task space. This is the approach that will be used in this work.

How can a real robot learn task space control from scratch? How can a robot with many DoF act in its relatively short “life time” so that the observed experiences can later be used to inform the robot how to best act to achieve any possible outcome in the task space, from any possible state the robot might find itself in? There are several challenges to consider here. There are an endless number of experiences a robot could potentially generate, but only a limited time in which to do so. What experiences a robot can discover also depends on what state the robot happens to be in, which depends on every previous action of the robot. On top of this is also the constant risk of robot damage, which would change the very dynamics the robot attempts to learn.

In this work we introduce a novel framework that tackles the task space learning problem in two ways. The first is to limit the set of experiences an agent attempts to collect, to focus on those that create novel effects in the task space. Since this space is generally low dimensional it would thus suffice with relatively few samples to cover it. Secondly we focus on online learning, so that every new observation can be immediately used in order to steer the robots behaviour towards actions that are more likely to generate useful data to the robot.

The capabilities of the proposed framework is exemplified in two separate robot problems, seen in Fig. 1.1. The first considers end-effector control of a redundant planar arm with up to 1,000 DoF acting in an environment with obstacles, and the second considers body orientation control of a physical humanoid robot. Learning to control redundant arms is a common problem in machine learning Baranes and Oudeyer [2013]; Hayashi [1991]; Rolf et al. [2011], while control
of body orientation has strong analogies to the early stages of child development Cangelosi and Schlesinger [2015]. Although body orientation control in robotics has been occasionally investigated previously Asada et al. [2009]; Kuniyoshi et al. [2004], examples of how to achieve such control autonomously are essentially absent in the literature.

1.1 Outline

Chapter 2 starts off by providing a mathematical description to the task space control problem we want to solve, and provides a mathematical description of the assumed dynamics of the robotic system. From these prerequisites emerges a set of particular challenges any particular solution will need to be able to handle, which will later be referred back to when analysing possible solutions to the problem.

Chapter 3 provides a literature review of previous common approaches for robot skill learning problems, and relates these to our particular set of challenges. We will see that although many of the previous methods are able to handle different aspects of the learning problem, neither is able to handle them all at once by itself.

In Chapter 4 we introduce the theoretical framework of this work, with its different parts and how they interact to create an overarching solution. This is followed up in Chapter 5 where we will see how each of the interacting parts of the framework can be practically learned online and in parallel, by a robot starting from, or close to, scratch.

Chapters 6 and 7 go on to experimentally evaluate two possible implementations of the proposed framework, over the two separate robot problems seen in Fig. 1.1. Each chapter is dedicated to one of these robot problems, where Chapter 6 considers end-effector control for a redundant planar arm with up to 1,000 DoF, and Chapter 7 considers body orientation control
for a physical humanoid robot with 25 DoF.

Finally, in Chapter 8 we conclude the outcomes, and speculate about the potential implications for future robotics, as well as outlining possible directions for future research on the topic.
The goal of this chapter is to give a general description of the robotic systems we consider, together with its implications to the task space control problem. Task space control here refers to the ability of a robot to freely control a given measurable parameter $x \in X$ into any value $x^\ast$. The space $X$ is referred to as the Task Space and $x^\ast$ is the task space goal which can be set freely. In some cases we will also discuss the control of multiple task spaces $(X^{(1)}, X^{(2)}, \ldots)$ in parallel, and in such a case it is also necessary to choose a goal task space $X^{(\ast)}$ for the particular task space goal. Task spaces will be manually defined throughout this work, although many previous studies have looked into how such spaces can be automatically detected and represented [Cartoni and Baldassarre, 2018; Jonschkowski and Brock, 2015; Laflaquière et al., 2015; Péré et al., 2018; Rolf and Asada, 2014].

To control a given task space $X$, the robot is allowed to select control signals $a \in A$ that are sent to some inner controller of the robot. This is translated into motor actions that lead to some effect $x$ in the task space. Learning to select such control signals in order to produce particular outcomes $x^\ast$ is, however, far from trivial. The outcome of a particular control signal $a$ does not only depend on the signal itself, but also the state $s \in S$ of the robot when sending the signal. Many task space outcomes are furthermore only achievable from a subset of initial states, meaning that the robot might first need to find a series of appropriate control signals $\{a_t, a_{t+1}, \ldots\}$ that would make the robot reach such a starting state to begin with.

The chapter ends by analysing what the described dynamics implies for any real robot, attempting to master task space control from scratch. This allows for the formulation of 5 core challenges that any particular solution would need to overcome.
Figure 2.1: The robot body with its environment seen as a black box. A robot in a state $s_t$ receives an action command $a_t$ at some time $t$. This action influences the robot’s behaviour for some time $\Delta t$ before the robot returns a new state $s_{t+\Delta t}$ and a task space outcome $x_{t+\Delta t}$.

### 2.1 Mathematical description

#### 2.1.1 The robot black box

To understand the task space control problem, it is essential to first understand the robotic system intended to achieve such control. For this it is helpful to view the robot and its environment as a black box. A robot body and its environment is here in some state $s_t \in S \subseteq \mathbb{R}^n$. At regular intervals $\Delta t$ a motor signal $a_t \in A \subseteq \mathbb{R}^m$ can be sent into this box. This is referred to as the action. The action induces some change in the system for another interval $\Delta t$ of time, before the black box returns the state $s_{t+\Delta t}$ which the robot ended up in, together with the resulting task space outcome $x_{t+\Delta t} \in X \subseteq \mathbb{R}^k$. Figure 2.1 gives an overview of this process. For simplicity $\Delta t$ will henceforth be set to 1, to be interpreted as “one iteration”.

Although we will later restrict what types of actions and task spaces are considered in this work, it is worth pointing out that the most general description allows for many different definitions of actions and task spaces. Actions $a$ are not necessarily restricted to what classically might be considered “actions”, such as motor torques or goal values, but could in principle be anything that influences some inner motor policy of a robot. This includes actions $a$ as weights $\theta$ of neural networks [Baranes and Oudeyer, 2013], or dynamic motion primitives [Forestier et al., 2017]. Task spaces are similarly not restricted to some feature of the state the robot, like the end-effector position of an arm, but could be something measured during the execution of $a$, for example how high the robot jumped or how much energy it used.

When considering choice of state descriptions $s \in S$ the most important thing is that that state description is complete. By complete we mean that there are no hidden parameters that effect the outcome of a given action. The outcome of doing a specific action $a$ from a state description $s$ should be deterministically predictable. For physical systems this will generally not be completely true. Most systems cannot be perfectly captured by any state description while also including inherently stochastic elements. In practice however smaller such deviations will not be a problem. No real physical system can ever return exactly to any previous state, which means that any difference in outcomes from doing the same action from similar (but not identical) states, can be explained by the slight difference in starting states.
2.1.2 Forward dynamics

As resulting states $s_{t+1}$ and task space position $x_{t+1}$ are assumed entirely dependent on previous state $s_t$ and action $a_t$, the resulting state can be described by a function $f : S \times A \rightarrow S$, so that:

$$f(s_t, a_t) = s_{t+1}$$

while the resulting task space outcome can be expressed by a function $\xi : S \times A \rightarrow X$, so that:

$$\xi(s_t, a_t) = x_{t+1}$$

This allows for the description a combined forward dynamics model $F(s, a) \equiv \left[ \begin{array}{c} f(s, a) \\ \xi(s, a) \end{array} \right]$ so that:

$$F(s_t, a_t) = \left[ \begin{array}{c} s_{t+1} \\ x_{t+1} \end{array} \right]$$

or if considering multiple task spaces $\{X^{(1)}, X^{(2)}, \ldots\}$, with $\xi^{(i)} : S \times A \rightarrow X^{(i)}$:

$$F(s_t, a_t) \equiv \left[ \begin{array}{c} f \\ \xi^{(1)} \\ \xi^{(2)} \\ \vdots \end{array} \right] (s_t, a_t) = \left[ \begin{array}{c} s_{t+1} \\ x_{t+1}^{(1)} \\ x_{t+1}^{(2)} \\ \vdots \end{array} \right]$$

Note that the state space $S$ here can be seen as a special task space $X^{(0)}$, with $f \equiv \xi^{(0)}$.

2.2 Goal state based robot control

The dynamics descriptions presented so far are, as previously mentioned, compatible with many different types of state and action spaces. For this work we will mainly focus on a particular type of robot control which will be referred to as “goal state based control”. This control is in turn based on robot controllers characterised by posture control, i.e control of a robot’s joint angles.

2.2.1 Postures and complete states

A posture $q \in Q$ is described as a vector $q = [\alpha_1 \ \alpha_2 \ \ldots]^T$ where each value $\alpha_i$ describes the angle of some joint $i$. The number of such angles determines the number of degrees of freedom (DoF) of the robot. In many cases a posture provides a complete description of an agent, meaning that the postures can also be used as states, so that $S = Q$. Robotic arms provides one such example, see Fig. 2.2 (left). Knowing the joint angles here uniquely defines the state of the arm. In other cases, for example humanoid robots, the posture alone is not sufficient to represent the state of the robot. The same posture can here be used in many possible configurations. A posture for standing can for example also be used for lying down, as shown in Fig. 2.2. One way to construct complete states in this case (for a robot at rest) is to also include a direction of gravity $\omega \in \Omega$ to the state, so that $s = [q^T \ \omega^T]^T$, giving a state space $S = Q \cup \Omega$. 

9
2.2.2 Posture control and goal states

A robot controller characterized by posture control is a controller that accepts a goal-posture \( \hat{q} = [\hat{\alpha}_1 \hat{\alpha}_2 \ldots]^T \) as action, which it then attempts to move the robot into. We thus have an action space \( A = Q \). Using goal postures \( \hat{q} \) as actions we get the following forward state dynamics:

\[
 f(s, \hat{q}) = s' 
\]

This notation is however not very useful as it carries no information on whether the given goal posture was reached or not. Instead of considering goal postures \( \hat{q} \), a more powerful representation is achieved by considering “goal states” \( \hat{s} \) so that \( A = S \). In a case such as the planar arm, this is already the case since \( S = Q = A \). To a humanoid robot on the other hand it means that we also need to add the orientation space \( \Omega \) to our action space, so that \( A = Q \cup \Omega = S \). An action is thus not only to choose a goal posture \( \hat{q} \), but also a goal orientation \( \hat{\omega} \), giving a goal state \( \hat{s} = [\hat{q}^T \hat{\omega}^T]^T \).

This leads to a forward state dynamics expression:

\[
 f\left(\begin{bmatrix} q \\ \omega \end{bmatrix}, \begin{bmatrix} \hat{q} \\ \hat{\omega} \end{bmatrix}\right) = \begin{bmatrix} q' \\ \omega' \end{bmatrix} 
\]

In practice, the choice \( \hat{\omega} \) has no influence on the dynamics of this system, since the actual controller of the robot only accepts goal postures \( \hat{q} \) as input. This means that Eq. (2.6) would result in the same outcome regardless of the choice of \( \hat{\omega} \). More important however is that this new notation allows us to express states and actions in the same space, since \( S = A \). We can thus rewrite Eq. (2.5)-(2.6):

\[
 f(s, \hat{s}) = s' 
\]
Here $s$ is the starting state, $\hat{s}$ the state the robot attempted to move to, and $s'$ is the state the robot actually ended up in. Using this notation we can now clearly see whether a goal was reached or not, since that would imply that $\hat{s} = s'$. For this reason this formalism will be preferred throughout this work, whenever discussing goal state control.

### 2.2.3 Goal state control

Given dynamics (2.7) we are now able to express some fundamental assumptions about the robot control. To begin with we will assume that the robot controller is able to preserve any state it has ended up in, which means that:

\[
(2.8) \quad f(s,s) = s
\]

This might seem trivial but is a first example that the robot controller is able to reach some goal states, even when that state includes parameters such as $\omega$ that the controller has no direct influence over. Since it is assumed that a controller only returns new states that can be stably held by the controller, it means that $\omega$ will not change unless $q$ changes. By telling the controller to preserve $q$, we indirectly also preserve $\omega$.

#### The “reach-by-accident” approximation

Having already seen that some goal states are indeed reachable, even in cases where the state includes parameters such as $\omega$ that the controller cannot affect directly, we move on to investigate what other goal states have this property. We are in other words looking for goal states $\hat{s}$ to a state $s$, for which:

\[
(2.9) \quad f(s,\hat{s}) = \hat{s}
\]

To find such reachable states we consider a reach-by-accident approximation. The idea is that if a state $s'$ can be reached from a state $s$ by attempting to reach some other state $\hat{s}$, then $s'$ can also be reached from $s$ by attempting to reach $s'$ directly. In other words:

\[
(2.10) \quad f(s,\hat{s}) = s' \Rightarrow f(s,s') = s'
\]

In the case of a planar arm where $S = Q$ and $(s,\hat{s},s') = (q,q',q')$ this might seem obvious. If a robot’s controller is able to change a posture $q$ into $q'$ when trying to change $q$ into $\hat{q}$, it should also be able to change $q$ into $q'$ intentionally. Remember however that Eq. (2.10) is just an approximation. If the controller does not behave identically with $q'$ as a goal to how it acted with the original goal $\hat{q}$, the outcome might be different. The approximation is nonetheless still useful and we will later see how it can be used to find true reachable goal states.

For robots with a state space $S = Q \cup \Omega$ we also need to consider orientation $\omega$. We have:

\[
(2.11) \quad f \left( \begin{bmatrix} q \\ \omega \end{bmatrix}, \begin{bmatrix} \dot{q} \\ \dot{\omega} \end{bmatrix} \right) = \begin{bmatrix} q' \\ \omega' \end{bmatrix}
\]
Figure 2.3: An illustration of the reach-by-accident assumption for two different controllers of varying strength. For controller \textbf{a}) the reach-by-accident assumption holds, as the agent reached a posture \( q' \approx \hat{q} \) that would also make it fall over. For controller \textbf{b}) the assumption fails, as the controller is not strong enough to preserve the angle that made it fall over, and returned to a posture \( q' \approx q \). In this case the agent would not be able to move from \( s \) to \( s' \) as \( q' \approx q \) would not make it fall over to begin with.

As in the previous case this gives us a reasonable assumption that posture \( q' \) can be reached intentionally from the initial state. The next question is, what will the effect be of reaching \( q' \) with regard to the orientation, will it still be \( \omega' \)? To answer this it is necessary to discuss the nature of orientations \( \omega \) and their relationship to postures.

In Fig. 2.2 we saw that the same posture can be consistent with many different orientations, as a standing posture can be used both for standing and lying down. Generally however, there is not an infinite set of orientations in which a given posture is stable. A dice at rest will only ever observe 6 different directions of gravity relative to some inner frame of reference. A counterexample to this is the uniform sphere, but we will assume such cases are rare and can be ignored. We thus have a finite set \( \{ \omega_1, \omega_2, \ldots, \omega_k \} \) of stable orientations to any given posture \( q \). These are the solution branches of \( q \), meaning that reaching posture \( q \) will always result in one of the orientations \( \omega_i \) of that set.

Going back to Eq. (2.11), this means that if the controller is able to reach posture \( q' \) again (in any way), then the resulting orientation will always be one solution branch \( \omega_i \) of posture \( q' \). We know that the previous result \( \omega' \) is one such branch by the fact that it was observed together with \( q' \). We cannot be sure however that if we would repeat experiment Eq. (2.11) but replace \( \hat{q} \) with \( q' \), that the controller would behave identically in reaching \( q' \). The more similar the controller behaves, the higher the likelihood that the same solution branch \( \omega_i = \omega' \) will be reached.

Fig. 2.3 gives an example of what the body orientation dynamics might look like. In it, a simple robot of 1 DoF starts out standing and attempts to reach a goal that accidentally makes it fall over. In outcome \textbf{a}) it fell into a state \( s' \) it can also reach intentionally from \( s \). The controller is here able to keep the posture \( q' \approx \hat{q} \) it moved to, that also made it fall over. In \textbf{b}) the state \( s' \) is not a state that could be intentionally reached from \( s \). Here the controller made the robot fall over when trying to reach \( \hat{s} \), but was then forced to change its posture back to \( q' \approx q \), as the controller was not strong enough to resist the induced joint torque. Note that the behaviour of the controller
when reaching \( q' \) by trying to reach \( \hat{q} \) and when trying to reach \( q' \) directly, differs significantly. In the first case the controller moves to \( \hat{q} \) and then back to \( q' \approx q \), while when asked to move to \( q' \approx q \) directly, it just attempts to preserve its starting posture \( q' \approx q \). As the behaviour for reaching \( q' \) intentionally is much different from reaching it unintentionally, the risk of reaching another solution branch increases, and would in this case keep the robot standing.

**Iterative search for reachable states**

Although the reach-by-accident approximation is not completely reliable, it hints towards a strategy for choosing increasingly reachable goals. The initial goal state \( \hat{s} \) can then be seen as a first approximation of a reachable state from \( s \). When attempting to reach it, the environment returned a better approximation \( s' \) of a reachable state from \( s \). This state too might not be (intentionally) reachable, but is a better approximation than \( \hat{s} \).

Assume \( \hat{s}_0 \) is the first approximation of a state that can be intentionally reached from some state \( s \), with subsequent approximations \( \{\hat{s}_1, \hat{s}_2, \ldots\} \) defined so that:

\[
\hat{s}_{i+1} = f(s, \hat{s}_i)
\]

For many controllers this series can be assumed to converge, so that:

\[
\lim_{i \to \infty} ||\hat{s}_{i+1} - \hat{s}_i|| = 0
\]

Defining \( \hat{s} = \lim_{i \to \infty} \hat{s}_i \) we thus get:

\[
f(s, \hat{s}) = \hat{s}
\]

which means that \( \hat{s} \) is indeed a reachable state from \( s \). In the case of Fig. 2.3a) this is happening already for \( i = 1 \) as \( s' \) is a reachable state, which is identical to the original reach-by-accident approximation, Eq. (2.10). For Fig. 2.3b) convergence is reached for \( i = 2 \), as \( \hat{s}_1 = s' \) gives \( f(s, \hat{s}_1) = s \), making \( \hat{s}_2 = s \). This results in the trivial case \( f(s, s) = s \), which nonetheless confirms that the procedure resulted in a reachable goal state.

**2.2.4 Fully connected state sets**

So far we have seen how robots can be controlled by goal states, and how that allows some states to be reached intentionally. This is theoretically useful and gives us an intuition about the dynamics of the system. It is however not very practical in a case of continuous, high dimensional, state spaces, where the chance of randomly finding ourselves in the exact state twice is practically non-existent. In this section we introduce the concept of “fully connected state sets” and see how these can be used for more efficient state space control.

**Definition 1.** A subset \( \hat{S} \subset S \) is fully connected if for all elements \( s_i, s_j \in \hat{S} \):

\[
f(s_i, s_j) = s_j
\]
A fully connected subset is thus a set of states the agent can move freely and intentionally in-between, as exemplified in Fig. 2.4. For a planar arm this could be a set of postures the arm can move directly between without risk of collision, or for a humanoid robot it could for example be a set of postures it can move between without falling over, if starting out standing.

A more detailed discussion of how to effectively find fully connected state sets will be left to later chapters. For now consider a simple example, similar to the process described for finding a single goal state ˆs. Assume an initial discrete set of states ˆS₀ = {s₁, s₂, ..., sₙ}, generated randomly or by some other mechanism. We can then use this initial set to create new sets {ˆS₁, ˆS₂, ...} by the following procedure:

1. Attempt to reach a random element ˆs ∈ ˆSᵢ.
2. Observe new state s' = f(s, ˆs).
3. Replace Sᵢ by a new version Sᵢ₊₁ with element ˆs replaced by s'.

In the limit such a set can only converge to a set ˆS = limᵢ→∞ ˆSᵢ if every transition always succeeds, since for a successful transition s' = ˆs ⇒ ˆSᵢ₊₁ = ˆSᵢ. Note that this example is mainly an illustration that iterative strategies can be used for finding fully connected state sets to begin with. This particular method has many shortcomings such as bad scalability with the number of states in the set, as well as no guarantees that the final set will not be trivial and for example only include duplicates of the robots current state.

### 2.3 The Control Learning Problem

Having described the particular dynamics of goal state based robot control, we now return to the more general description Eq. (2.3) of the system to describe the task space control problem, were actions are not necessarily goal states so that A ≠ S.
2.3.1 The Control Problem

Controlling a task space $X$ entails the ability to, given a freely specifiable goal $x^* \in X$ and any starting state $s_t$, choose a sequence of actions $\{a_t, a_{t+1}, a_{t+2}, \ldots, a_{t+\tau-1}\}$, so that:

\[
F(s_t, a_t) = [s_{t+1} \ x_{t+1}]^T \\
F(s_{t+1}, a_{t+1}) = [s_{t+2} \ x_{t+2}]^T \\
F(s_{t+2}, a_{t+2}) = [s_{t+3} \ x_{t+3}]^T \\
\vdots \\
F(s_{t+\tau-1}, a_{t+\tau-1}) = [s_{t+\tau} \ x^*]^T
\]

(2.16)

Often there will also be some upper limit $T \geq \tau$ for how many iterations the robot is allowed in order to reach any particular goal $x^*$.

To select the sequence of actions we search a goal-conditioned policy $\pi: S \times X \rightarrow A$, with:

\[
a_t = \pi(s_t, x^*),
\]

so that Eq. (2.16) is fulfilled.

2.3.2 Challenges

Finding a policy $\pi(s, x^*)$ that is able to solve Eq. 2.16 is associated to a number of challenges, especially considering that such a policy needs to be found by a complex physical robots starting from as close to tabula rasa as possible, where the only data on which the policy can be based is the outcomes of the robot’s own actions.

Starting from the point of view of available data on which a policy can be based, there are two fundamental limitations to consider:

1. **Sparsity**: The number of samples that can be collected by the robot is strictly limited. Every action takes some time to do, while also posing a risk of resulting in robot damage, making training costly both in time and eventual need for repair.

2. **No reset**: Since the robot is assumed to act in the real world, it can only observe outcomes of actions attempted from the states the robot already happens to be in. This means that a robot that falls over when attempting to learn to walk cannot simply reset itself to that initial state and try again, but will first need to learn how to stand up before being able to attempt another step.

The next challenge is how to generalise the observed data in order to generate a policy $\pi(s, x^*)$. The main issue here lies in combinatorial explosions:

3. **Dimensionality**: The number of state-actions needed to roughly approximate the outcome of any combination $(s, a)$ grows exponentially with the number of dimensions of the combined state-action space $S \times A \subseteq \mathbb{R}^{m+n}$. For a humanoid robot with 20 DoF and goal state
control, this implies a state-action space of at least 40 dimensions. To evenly collect every state and action combination of such a robot, by only considering 2 different settings for each joint, would require $2^{20 \times 20} \approx 10^{12}$ samples to collect.

4. **Planning:** Planning a sequence of intermediate states $\{s_{t+1}, s_{t+2}, \ldots\}$ in order to reach a state $s_{t+\tau-1}$ from which $x^*$ can be reached, as seen in Eq. 2.16, opens up for a combinatorial explosion of possible action sequences $\{a_t, a_{t+1}, \ldots, a_{t+\tau-1}\}$. With an action space $A \subseteq \mathbb{R}^n$, this implies $k^{\tau n}$ possible choices for a planned sequence of $\tau$ steps, if every action parameter is allowed $k$ different values.

Finally it is also important to consider effects directly related to acting in a real world:

5. **Risk of damage:** Although already mentioned, the risk of robot wear down and damage is worth pointing out as its own challenge. Since the robot has to learn the outcomes of actions from as close to tabula rasa as possible, there is no way for the robot to know in advance whether an action might be damaging to it or not. The main problem here is, from a machine learning point of view, that a robot that gets damaged during a training session also changes its dynamics, making large parts of the previously collected data inaccurate.

In the following Chapter 3 we will consider some common approaches of machine learning, and discuss how these approaches relate to the challenges presented here.
Having robots or other types of artificial agents learn skills by themselves is a long standing endeavour in the field of machine learning. This chapter presents a literature review of some of the most relevant approaches with respect to the control problem outlined in the previous chapter, with the main focus of how each method handles the 5 core challenges: Sparsity, No reset, Dimensionality, Planning and Risk of damage. Table 3.1 finally gives a comprehensive overview of the most prominent approaches and scores how each of these are able to cope with the different challenges of our described system.

3.1 Reinforcement Learning

One of the most well-known frameworks for handling problems where an agent has to learn how to act from scratch is Reinforcement Learning (RL) [Sutton and Barto, 2015]. RL considers systems where an agent is in some state $s_t \in S$ and can do an action $a_t \in A$, which results in a new state $s_{t+1}$ and a reward $r_{t+1} \in \mathbb{R}$ that scores how desirable the outcome was with respect to some given task. The main goal of RL is to find an action-policy $\pi : S \rightarrow A$, that maximises the future accumulated reward $\sum_{i=1}^{\infty} \gamma^i r_{t+i}$ an agent can receive from any starting state $s_t$. The parameter $0 \leq \gamma < 1$ is called the future discount factor, and controls how much the agent should prioritise immediate rewards in relation to delayed future rewards. RL has previously been used for learning a wide range of different skills, such as playing computer games [Barratt and Pan, 2019; Mnih et al., 2015; Vinyals et al., 2017], generating gate [Heess et al., 2017; Lillicrap et al., 2015] and hand-eye-coordination [Levine et al., 2016, 2018]. Although many of these results are impressive they generally rely on huge sets of data, which prevents the methods from being directly applied to most real robotic systems. Some attempts to counter this can be found in the research directions of “micro-data RL” [Chatzilygeroudis et al., 2018; Mouret, 2016], “one-shot
learning” [Duan et al., 2017; Finn et al., 2017], and “transfer learning” [Barrett et al., 2010; Devin et al., 2017; Tobin et al., 2017].

3.1.1 Goal-conditioned RL

The RL problem is in many ways similar to the problem described in Sec. 2.3.1. The two main differences are that the RL problem receives a scalar reward $r_{t+1}$ and not a task space outcome $x_{t+1}$ after every iteration, and that the objective of RL is to optimise the future accumulated reward $\sum_{i=1}^{\infty} \gamma^i r_{t+i}$, whereas the task space control objective is to be able to reach any freely specifiable goal $x^*$.

There is a special sub-field of RL dealing with exactly these types of problem, sometimes referred to as goal-conditioned RL [Ghosh et al., 2018]. Goal-conditioned RL bridges the gap between traditional RL and task space control by defining the reward $r_t$ as a function of how close a task space outcome $x_t$ is to a given goal $x^*$. For example:

$$r_t(x_t, x^*) = \exp\left(-\frac{||x_t - x^*||}{\mu}\right)$$

where $\mu$ is some constant describing how fast the reward should decline with the distance. Optimising the expected future reward then turns into the problem of reaching as close to $x^*$ in as few steps as possible (to avoid future discount due to $\gamma$), and then be able to continuously sustain a $x_t \approx x^*$ output for as long as possible.

Considering goal-conditioned RL, the policy we are searching for thus takes the form

$$\pi : S \times X \rightarrow A \ , \ \pi(s_t, x^*) \rightarrow a_t$$

which is identical to the policy described in Chapter 2. In order to find such a policy there are two basic branches of RL to consider: model-free, and model based approaches.

3.1.2 Model-free approaches

The idea of model-free RL is to optimise a set of parameters $\theta$ with respect to accumulated future reward, where the parameters $\theta$ controls the behaviour of an action-policy $\pi$ either directly as weights in a neural network, or more indirectly by defining an action value function.

Direct policy search

In direct policy search [Deisenroth et al., 2013], the robot is controlled by an action-policy $\pi_\theta$ which is directly encoded by the parameters $\theta$, for example as weights in a neural network. The idea is to explore different choices of $\theta$ and then evaluate each choice by the reward an agent collects when using that setting. This is ideally done over multiple roll-outs from some given starting distribution. A “roll-out” is an isolated experiment in which a given policy $\pi_\theta$ is allowed to control the robot for some predetermined number of iterations or time, while collecting reward.
The received reward of each choice \( \theta \) can then be used to update \( \theta \), using for example gradient based approaches [Peters and Schaal, 2006; Williams, 1992], or gradient-free methods such as genetic algorithms [Clune et al., 2009; Salimans et al., 2017]. Although approaches like these have proven powerful in many contexts, they are inherently ill-suited for learning problems where many different starting conditions and objectives must be considered. Besides the need for the robot to be able to return to an unbiased distribution of starting conditions before any new roll-out (in order to compare different choices of \( \theta \) fairly), the sheer size of possible combinations of starting states \( s_0 \) and goals \( x^* \) makes the method unfeasible to any situation with limited training data.

**Value based RL**

Another model-free approach is the *value based* approach. Instead of finding parameters to a policy \( \pi \) directly, the idea is to use \( \theta \) to approximate optimal action-value functions \( Q^* : S \times A \to \mathbb{R} \). An optimal action-value function \( Q^*(s,a) \) is defined to return the average accumulated reward a robot will receive if doing a particular action \( a \) from a state \( s \), and then act optimally for all other actions. Mathematically this can be expressed using the *Bellman equation* [Sutton and Barto, 2015]:

\[
Q^*(s,a) = \mathbb{E} \left[ r' + \gamma \max_{a'} Q^*(s',a') \mid s,a \right]
\]

Since this function returns the highest possible expected future reward for any action, it also allows the formulation of an optimal policy:

\[
\pi^*(s) = \arg\max_{a \in A(s)} Q^*(s,a)
\]

which by definition returns the optimal action in any situation. Here, \( A(s) \) is the set of actions that can be made from state \( s \).

The idea of value-based RL is to approximate this optimal function using a parameterized function \( Q_{\theta} \). This can be done by using observations of state-action combinations \((s,a)\) and the new states and rewards \((s',r')\) that combination led to. By inserting these observations into Eq. (3.3), \( \theta \) can be updated to fulfil the equality. An important property of this approach compared to the direct policy search is that it here does not matter how these observations \((s,a,s',r')\) were collected. This means that it is no longer necessary to explicitly collect new data for every new update of \( \theta \), since the outcome of any given state-action \((s,a)\) does not depend on the mechanics by which the action \( a \) was selected. This allows the agent to sometimes act “off policy” and select actions that seems non-optimal for the current version of \( Q_{\theta} \), which is often useful as \( Q_{\theta} \) is only an approximation and such exploration allows it to improve. A popular example of such off-policy exploration is to use the so called \( \epsilon \)-greedy policy. Here the agent selects a random action with a probability \( 0 \leq \epsilon \leq 1 \), and an action using Eq. (3.4) otherwise, which is also referred to as the *greedy* action.
Value based RL has been used in a long list of applications [Andrychowicz et al., 2017; Lillicrap et al., 2015; Mnih et al., 2015; Schaul et al., 2015], of which the Deep Q-Networks (DQN) algorithm, first introduced in [Mnih et al., 2015], is among the most well known. In this original work it was shown that a $Q_\theta$ function trained using a deep neural network could learn to play Atari games from scratch, sometimes to a superhuman level. More relevant to the context of this thesis however is that value-based approaches like DQN can also be extended to handle goal-conditioned RL, in particular the special case when $X = S$ with state goals $s^*$ instead of task space goals $x^*$. Since the value functions $Q_\theta$ can be trained off-policy, and the given reward $r(s, s^*)$ from an observation $(s_t, a_t, s_{t+1})$ can be deduced in retrospect, it is possible to create training data for all possible goals $s^*$ using these samples, regardless of the goal the agent tried to reach when doing so. This approach was tested and shown to work in series of smaller computer games [Schaul et al., 2015].

An issue with DQN is that it only works for finite discrete action spaces, since that is generally necessary to solve $\arg\max_a Q_\theta(s, a)$ of Eq. (3.4). To deal with continuous action spaces, actor-critic methods like Deep Deterministic Policy Gradients (DDPG) [Lillicrap et al., 2015] is one approach. Actions are here selected using a parameterized policy $\pi_\phi$, called the “actor”, together with a value function $Q_\pi^\theta$, called the “critic”. The critic is however slightly different from previous value functions, in that it does not attempt to approximate the optimal function $Q^*(s, a)$, but rather attempts to approximate the value of a given action, if the agent would then continue by following the policy $\pi_\phi$ (compared to acting optimally). This means that $Q_\pi^\theta$ cannot learn off-policy and needs to be retrained for every update of $\pi_\phi$. Despite this the DDPG approach has proven able to learn many successful behaviours on simulated robots with continuous action spaces, for example using the MuJoCo engine [Lillicrap et al., 2015; Plappert et al., 2017; Todorov et al., 2012]. In the Hindsight Experience Replay (HER) framework [Andrychowicz et al., 2017] it was shown that DDPG can also be adapted to goal-conditioned RL, in a similar manner to DQN. In the paper [Andrychowicz et al., 2017] a simulated robot with 4 degrees of freedom (DoF) was able to learn 3 different tasks with variable goals after approximately 80 million simulated iterations. The developed policies could then be transferred to a real robot.

**Advantages and disadvantages**

Although there are many impressive results from the different approaches of model-free RL, a general problem with all model-free approaches is their sample inefficiency. A model-free approach can only learn from rewards, which means that experiences that receive no rewards offer minimal supervision for the parameter optimisation [Pong et al., 2018]. This generally restricts model-free approaches to simulations, as the required training time would be unfeasibly long for any physical robot of any complexity. Another problem is that the model needs to be trained for a large set of situations the agent might find itself in, so that parameters $\theta$ have been evaluated for each such settings. The typical way to get around this is to construct experiments so that the contexts a robot might find itself in are limited in one way or another. This becomes
even more important if an agent has many DoF. In the case of a running humanoid in the work of [Heess et al., 2017] for example, this is done by roll-outs where the agent is reset into a small set of starting conditions after every evaluation. As the learner progresses it learns better and better how to contain the humanoid to a narrow band of states that are part of the running motion, and should the robot fall outside this small set of states the roll-out would simply terminate and the agent is allowed to start over. This means that the agent has neither knowledge of how to get up again if falling, nor how to act in order to achieve any other objective than the running task it was trained for.

To finally evaluate model-free RL from the point of view of the challenges presented in Sec. 2.3.2, we see that it is in some cases able to handle both large state and action spaces as well as planning actions in multiple steps. It would however require massive amounts of training data to do so in a system without the ability to reset and with freely definable goals $x^*$, making it unfeasible for any robot solely acting in the real world.

### 3.1.3 Model-based approaches

Instead of trying to immediately find policies for how to optimise some long term reward, an alternative approach is to first attempt to create a model of the world and then use this world model to develop policies. This is the essence of model-based RL and it has many advantages. Since every observation $(s_t, a_t, s_{t+1})$ contains some true information about the structure of the world, no observation is completely wasted as it tells us something about the world, meaning that observed data can be used more efficiently than in model-free counterparts [Deisenroth et al., 2013; Nagabandi et al., 2018]. Secondly a model gives the possibility to optimise policies to any number of tasks that can be captured by that model, making it well suited to goal-conditioned RL. Optimising policies from a model however has the disadvantage that the model will never be able to capture the world perfectly, which reduces the quality of the policies as they are adapted to the model, and not to the real world [Pong et al., 2018].

#### Tabular model learning

Probably the simplest model-based approach is tabular RL, where the goal is to approximate a forward model of a discrete Markov Decision Process (MDP) [Frank et al., 2014; Sutton and Barto, 2015]. Consider a finite discrete state space $S = \{s_1, s_2, \ldots, s_n\}$ with a finite discrete action space $A(s_i) = \{a_{i1}, a_{i2}, \ldots, a_{imi}\}$ to every state $s_i$. For simplicity we will consider task spaces $X = S$. In this setting we attempt to learn a model $P(s'|s, a)$, that expresses the probability that the agent will end up in a state $s'$ when doing action $a$ in state $s$. This probability can be approximated in many different ways. The most straightforward approach is to simply record in a table $T(s'|s, a)$ the number of times each state-action $(s, a)$ resulted in subsequent state $s'$, and then compute:

$$
P(s'|s, a) = \frac{T(s'|s, a)}{\sum_{s''} T(s''|s, a)}
$$

(3.5)
Note however that this is undefined if state-action \((s, a)\) has never been attempted. Two common choices here is to either define in advance that \(0 = P_0\) for Eq. (3.5), where \(P_0\) is some initial bias, or to initialise \(T(s, a, s') > 0\) for some outcomes \(s'\) to every state-action \((s, a)\).

Another way to approximate \(P(s'|s, a)\) is to initialise it to some value \(P_0\) and then update it using an exponential moving average. Given a new observation \((s, a, s')\) we then get:

\[
P(s''|s, a) \leftarrow (1 - \alpha)P(s''|a, s) + \alpha I(s', s'')
\]

where \(I(s', s'')\) is an identity function where \(I(s', s'') = 1\) if \(s' = s''\) and \(I(s', s'') = 0\) otherwise, and \(\alpha \in [0, 1]\) is the decay rate of old estimates. This approach can be advantageous if the underlying probabilities may change over time as it keeps a constant adaptation rate, but at the cost of never converging completely.

Given approximated probabilities \(P(s'|s, a)\) we can also define a goal-conditioned reward function:

\[
R(s, a, s', s^*) = \begin{cases} 
1, & s' = s^* \\
0, & s' \neq s^*
\end{cases}
\]

which allows us to rephrase the Bellman equation (3.3) so that

\[
Q(s, a, s^*) = \sum_{s'} P(s'|s, a)[R(s, a, s', s^*) + \gamma V(s', s^*)]
\]

with \(V(s, s^*) = \max_a Q(s, a', s^*)\), referred to as the state-value function. Functions \(Q\) and \(V\) can then be approximated using for example value iteration, see Alg. 1.

**Algorithm 1** Value iteration (Adapted from [Frank et al., 2014])

1: Initiate \(V(s, s^*) \leftarrow 0\) for all \(s \in S\)
2: Initiate \(Q(s, a, s^*) \leftarrow 0\) for all \(s \in S, a \in A(s)\)
3: Set threshold \(\Delta_{\text{max}}\)
4: 
5: while Running do
6: \(\Delta \leftarrow 0\)
7: \(\text{for each } s \in S \text{ do}\)
8: \(\quad \text{for each } a \in A(s) \text{ do}\)
9: \(\quad \quad Q(s, a, s^*) \leftarrow \sum_{s'} P(s'|s, a)[R(s, a, s', s^*) + \gamma V(s', s^*)]\)
10: 
11: \(v_0 \leftarrow V(s, s^*)\)
12: \(V(s, s^*) \leftarrow \max_a Q(s, a, s^*)\)
13: \(\delta \leftarrow |V(s, s^*) - v_0|\)
14: \(\Delta \leftarrow \max(\Delta, \delta)\)
15: 
16: \(\text{if } \Delta < \Delta_{\text{max}} \text{ then}\)
17: \(\quad \text{break}\)
18: 
22
Continuous model learning

To approximate a continuous forward model, such as Eq. (2.3) in Sec 2.1.2, it is necessary to use some function approximator that can take observations \( \{(s_t, a_t, s_{t+1}, x_{t+1})\}_{t=0}^{T-1} \) from the environment and construct approximations \( \hat{f} : S \times A \to S \) and \( \hat{\xi} : S \times A \to X \) so that \( \hat{f}(s_t, a_t) \approx s_{t+1} \) and \( \hat{\xi}(s_t, a_t) \approx x_{t+1} \). There are multiple ways such approximations can be made, where deep neural networks [Kwiatkowski and Lipson, 2019; Nagabandi et al., 2018], radial basis functions [Sigaud et al., 2011], local linear regression [Cleveland and Devlin, 1988] and Gaussian process regression [Rasmussen, 2004] are among the more common. When considering what method to use it is important to consider how the function will generalise to unseen state-actions. Unless care is taken such generalisations could lead to predictions such as negative mass or positive friction, that policies in turn might attempt to exploit when being trained on the model. Since such effects do not exist in the real world these policies would not be able to perform well when implemented on the real system. In order to avoid such effects, many methods try to incorporate uncertainty levels for the parts of the model where less data has been observed [Deisenroth et al., 2013; Lütjens et al., 2018].

Despite the challenges of using continuous models there are some examples where they have been successfully utilised in the development of robot policies [Gu et al., 2016; Nagabandi et al., 2018]. In [Nagabandi et al., 2018] simulated agents with up to 8 DoF learned forward models in MuJoCo. In a first epoch the agent would pick actions randomly, commonly referred to as “motor babbling”, in order to create an initial dataset of observations \( \{(s_t, a_t, s_{t+1})\}_{t=0}^{T-1} \). These observations where then used in order to train a deep neural network \( \hat{f}_\theta(s, a, s') \), where \( \theta \) are the weights in the network. Given such a model, policies where developed in order to generate forward locomotion. The policy was then evaluated in MuJoCo while collecting new observations which could be used to improve the model, allowing the policy to improve, and so on. This feedback loop allowed the model to become increasingly accurate around state-actions that the developing policies would visit when attempting to generate walk. In the end the approach would typically require around \( 10^8 \) environmental steps before an effective behaviour was developed.

One way to allow for faster learning of forward models is by introducing some initial knowledge as bias into the model. In [Bongard et al., 2006] this forward model consisted of a physics environment where the agent only needed to find a few parameters that described the robots basic morphology. Given such bias even physical robots with many DoF can acquire a useful forward model within a feasible time span, although it also diverges from the idea of learning a forward model from tabula rasa. It also excludes any robot that does not behave as this prior implies.

Advantages and disadvantages

As previously stated, the main advantages of using a model is that it allows every observation to be used, making such methods potentially more sample efficient than model-free approaches.
Having a model also allows for the development of policies to many different tasks, as the physics
of the world are independent of what particular task an agent might try to solve at any given
time.

The drawbacks of model learning is that any policy developed using a model will be based
on the dynamics of the model, and not reality, leading to a worse asymptotic performance than
any policy developed in the real world. It also scales badly with the sizes of the state and action
spaces, which are both directly connected to the number of DoF of an agent. This is particular
bad for a tabular RL solution, as any segmentation of such a state-action space would quickly
explode in the number of elements with the DoF. One advantage with a tabular model however,
compared to a continuous model, is that if the state space and action spaces are indeed discrete
and small such a model can be learned with a relatively sparse set of observations as the set of
possible state-actions is finite and can thus be observed explicitly.

3.2 Goal babbling

One of the biggest drawbacks of existing RL methods, is their need for massive amounts of data
in order to handle high-dimensional continuous state and actions spaces. Goal Babbling (GB)
is a recently developed approach [Rolf et al., 2011] that is known for high efficiency in exactly
such cases, but only for the subset of problems where policy search can be expressed as an inverse
model search [D’Souza et al., 2001; Nguyen-Tuong and Peters, 2011]. This happens in cases
where the initial state \(s\) can be ignored when evaluating the task space outcome \(x\) of an action \(a\).
This can for example happen if the robot is reset to some starting state after every action (so that
\(s\) is always the same), or if the effects of actions are simply independent of the state in which
they are made, generally due to some inherent property of the system. Assuming that states
can be ignored, the forward task space function \(\xi : S \times A \rightarrow X\) can be reduced to \(\xi : A \rightarrow X\). This
removes the need to plan transitions to any particular state, and the goal of the action policy
simply becomes to map every goal \(x^*\) to a particular action, giving a policy \(\pi : X \rightarrow A\), so that
ideally:

\[
\xi(\pi(x^*)) = x^*
\]

The policy is thus an inverse function \(\pi(x) = \xi^{-1}(x)\), by the definition of inverse functions.

3.2.1 Main features

The GB approach is very general and versatile, and has previously been applied to a vast range
of problems such as posture control [Benureau, 2015; Rolf, 2013; Rolf et al., 2011], soft robotics
[Rolf and Steil, 2013], inverse statics [Rayyes et al., 2018], hand-eye coordination [Schmerling
et al., 2015], speech [Moulin-Frier et al., 2014], gate [Baranes and Oudeyer, 2013], and tool use
[Forestier et al., 2017; Forestier and Oudeyer, 2016].
The main feature of GB is that it is a learning-by-doing approach. Instead of first creating training data using “motor babbling”, where motor actions are randomly chosen, which is a common approach in machine learning, GB instead selects goals $x^*$ in the task space that it attempts to achieve, hence the name “goal babbling”. The agent then attempts to reach these goals using its current version of the policy $\pi(x)$, which is then updated using the observed result of the attempt. The effect of this is that GB explores the task space directly, compared to the more common approach of first exploring the action space and then observe the effect of each action with respect to the task space. This fundamentally uncouples the learning speed of GB from the dimensionality of the action space and instead ties it to the dimensionality of the task space, which is generally very low dimensional. GB thus allows agents, with in principle any number of DoF, to efficiently learn to control low dimensional task spaces [Benureau, 2015; Rolf et al., 2011].

### 3.2.2 Challenges

#### The unknown range problem

How to choose goals $x^* \in X$ in order to collect data to improve a given policy $\pi(x)$ is an integral part of the GB framework, as generally not all goals can be physically achieved. A planar arm can never reach beyond its length, a robot cannot stand in the air, etc. To avoid wasting training time on impossible goals it could thus be useful to somehow determine what the achievable goals are. There are many potential solutions to this problem. “Direction sampling” [Rolf, 2013] uses the idea that that boundaries of what is reachable can be extracted by attempting to follow straight trajectories in the task space and then see when the robot would diverge from these paths. This diversion is a necessity for any goal trajectory reaching out in unreachable territories, as following would mean reaching something unreachable. These deviations are then used by the direction sampling approach to estimate the reachable boundary. An alternative approach is presented in [Baranes and Oudeyer, 2013], where goals $x^*$ where selected based on the agents learning progress in different regions of the task space. This allows the agent to focus its exploration on the parts of the task space to which the likelihood is high that the inverse model can improve, avoiding spending time on goals that the robot already knows how to reach or on goals that are seemingly impossible. This is an intrinsic motivation approach, a concept which will be furthered explored in Sec. 3.3.1.

#### The consistency problem

The consistency problem emerges from the fact that the forward function $\xi(a)$ is surjective in general. This means that there can be two actions $a_i, a_j$ that produce the same outcome $x$, so that $\xi(a_i) = \xi(a_j) = x$, but where $\xi(\frac{ma_i + na_j}{m+n}) \neq x$, for some choices $m, n > 0$. If this is the case we say that $a_i$ and $a_j$ are inconsistent [Rolf et al.]. A simple example of such a case is the forward function $\xi(a) = \sin(a)$, for $a_i = 0, a_j = \pi$. The main issue with the consistency problem is that it affects how
an inverse model can estimate actions, given a set of observations \( D = \{(a_t, x_t)\}_{t=1}^{T} \). A spontaneous approach for reaching a goal \( x^* \) would be to simply generate an action by interpolating over the actions \( a_t \) in the dataset with outcomes \( x_t \) close to \( x^* \), using some kind of regression method. This was the general approach for approximating forward functions in Sec. 3.1.3. Because of the consistency problem however, such an approach risks averaging over inconsistent samples in a destructive way as averages over solutions are not necessarily solutions themselves in this case. Some method of generating actions from the observed data is however still needed in GB in order to create an inverse model \( \pi(x) \) and approximate actions for unseen outcomes. In the following Sec 3.2.3 and 3.2.4 we will discuss two ways of doing so.

3.2.3 Nearest neighbour approach

Perhaps the simplest implementation of GB is based on nearest neighbour search [Benureau, 2015]. The basic idea is, given a goal \( x^* \in X \), to simply use the single action previously observed with the most similar outcome to \( x^* \). Noise is then added to the action (often proportionally to the distance between the sample and \( x^* \)) to allow discovery of new samples, potentially closer to \( x^* \). Since only one sample is used at a time in the nearest neighbour approach the consistency problem is completely avoided as no averaging is taking place. The drawback here is that a single sample does not provide a gradient which makes generalisations to unseen task space positions limited to the distance to the nearest sample. Another drawback is that the agent will not be able to follow a smooth trajectory in the task space as the policy is not continuous and might change significantly between neighbouring samples.

The nearest neighbour approach can be implemented using a dataset \( D = \{(a_i, x_i)\}_{i=1}^{T} \) of observations, which are collected by attempting to reach randomly drawn goals \( x^* \in X \). If the dataset is empty a random action can be made to initialise it. Using these we define the policy \( \pi : X \rightarrow A \) so that:

\[
\pi(x) = a_i : i \leftarrow \arg\min_j \|x_j - x\|
\]

Before using this estimate to attempt to reach a goal \( x^* \) we also add a noise vector \( \eta(x_i, x^*) \) for exploration:

\[
a_t = \pi(x^*) + \eta(x_i, x^*)
\]

This noise vector is typically normally distributed, with an amplitude proportional to the distance \( \|x_i - x^*\| \) to the nearest neighbour \( x_i \). The final action \( a_t \) is then sent to the robot’s controller, leading to an outcome \( x_t \), which is added to the dataset. The full approach can be seen in Alg. 2.

3.2.4 Continuous manifold goal babbling

A continuous policy \( \pi(x) \) has many advantages. Firstly it allows for interpolation and extrapolation from observations towards undiscovered regions of the task space. Secondly, when considering
Algorithm 2 Nearest Neighbour Goal Babbling (Adapted from [Benureau, 2015])

1: \( D \leftarrow \{(a_0, x_0)\} \) \( \triangleright \) Initiate dataset with an initial observation \((a_0, x_0)\)

2: 

3: \textbf{for} \( t \in [1, \ldots, T] \) \textbf{do}

4: \( x^* \sim U(X) \) \( \triangleright \) Sample a goal uniformly randomly from the task space

5: \( i \leftarrow \text{argmax} \sum_j ||x_j - x^*|| \) \( \triangleright \) Find closest sample in \( D \)

6: 

7: \( a_t \leftarrow a_i + \eta(x_i, x^*) \) \( \triangleright \) Add exploratory noise to nearest action

8: \( x_t \leftarrow \xi(a_t) \) \( \triangleright \) Execute action \( a_t \) and observe task space outcome \( x_t \)

9: 

10: \( D \leftarrow D \cup (a_t, x_t) \) \( \triangleright \) Add the new observation \((a_t, x_t)\) to the dataset

goal state based control (see Sec. 2.2.3) where \( A = S \), a continuous policy creates a smooth manifold in the task space of states the agent can move along while following trajectories in the task space. Given a discontinuous policy such as the nearest neighbour approach this would not be possible as it would require significant posture changes with every change of nearest neighbour during the trajectory.

To create a continuous policy \( \pi(x) \) in a goal state setting it is necessary to solve the consistency problem. In GB, this problem was solved in [Rolf et al., 2011] in the case of goal state control, with \( A = S \). To do so two different mechanisms were used: state recoveries, and consistency checks.

State recoveries

The first mechanism will be referred to as “state recoveries”. It means that now and then the agent will return to a state that is part of the solution manifold we want the policy \( \pi(x) \) to converge towards. In [Rolf et al., 2011], this was done by defining a single state \( s_{\text{home}} \), called the “home state”\(^1\) which the robot returns to with varying frequency. This acts as an anchor and a first point in the manifold \( \pi(x) \), as it gives a predetermined value \((s_{\text{home}}, \xi(s_{\text{home}}))\) in the final manifold. New data \((s_t, x_t)\) is then dismissed or accepted depending on its consistency with respect to this home state.

Consistency checks

The second mechanism is the “consistency check”. It gives every observation \((a_i, x_i)\) a weight \( w_i \) that corresponds to to what degree it is consistent to the manifold \( \pi(x) \) that is being created. The main intuition is that because this manifold is based on previous observations, inconsistencies only appear when the results of an action diverges from what is intended. The reason for this is that action estimates are based on averages of previous observations, and if this average creates the desired effect it should be compatible with the dataset that created it. If the agent however

\[^1\]In literature it is more commonly referred to as the “home posture”, but generally only if it is implicit that the posture is also the state of the agent, see Fig. 2.2.
attempts an average and ends up somewhere else, there is no reason to believe that this sample would work well with the data that would be used to create an average in the place where it ended up instead. To practically enforce consistency, [Rolf et al., 2011] rated two failure modes:

1. When the trajectory of the end effector is moving in another direction than the trajectory it attempts to follow.

2. When the end effector is not moving significantly, despite a significant change in state.

To compute these values it is necessary to collect observations of how well a trajectory \( \{x_t^*\}_{t=1}^k \) of multiple goals was followed, rather than to just evaluate how well a single goal \( x^* \) was reached. The agent then attempts to follow this trajectory of goals using goal states \( \hat{s}_t = \pi(x_t^*) \). The observed outcomes \( x_t = \zeta(\hat{s}_t) \) are then given a directional weight \( w_{t}^{\text{dir}} \):

\[
\begin{align*}
\text{(3.12)} \\
\ w_{t}^{\text{dir}} = \frac{1}{2}(1 - \cos\angle(x_t^* - x_{t-1}^*, x_t - x_{t-1}))
\end{align*}
\]

where \( \angle(a, b) \) is the angle between two vectors \( a, b \). This means that observations where the agent starts to move in a direction different than intended are given less significance, with 0 significance if the end effector moves away from the goal.

The second weight deals with movement efficiency and is:

\[
\begin{align*}
\text{(3.13)} \\
\ w_{t}^{\text{eff}} = \frac{||x_t - x_{t-1}||}{||\hat{s}_t - \hat{s}_{t-1}||}
\end{align*}
\]

The main property of this weight is to benefit the solutions that requires the least amount of state change to move in the task space. The two weights are then combined into a final weight:

\[
\begin{align*}
\text{(3.14)} \\
\ w_t = w_{t}^{\text{dir}} \cdot w_{t}^{\text{eff}}
\end{align*}
\]

Together these weights make sure that any inconsistent sample is automatically weighted too low to have a significant impact on the policy \( \pi(x^*) \), which is created using weighted regression on the dataset \( D = \{(s_t, x_t, w_t)\}_{t=1}^T \) of all seen observations.

### 3.2.5 Multi-Task Goal Babbling

One direction of GB worth mentioning is Multi-Task Goal Babbling, often called “model babbling” [Forestier et al., 2017; Forestier and Oudeyer, 2016]. Here multiple task spaces \( \{X^{(1)}, X^{(2)}, \ldots\} \) are considered in parallel, for which a set of policies \( \zeta^{(i)} : A \rightarrow X^{(i)} \) are searched. This adds another dimension to choosing goals, as it is now also necessary to choose in which task space \( X^{(i)} \) to put the goal in pursuit. Note however that any action, regardless of intention, will lead to observations in all task spaces. This can be used to train all inverse models using the nearest neighbour formulation of GB. For the continuous case formulation it would be more challenging as goal trajectories would only exist in the goal space. It could also create problems as each task space would create its own manifold, and the interactions between these manifolds is not trivial.
Learning multiple models in parallel can have many advantages. In [Oudeyer et al., 2007] it was shown that robots that can choose between many different task spaces can speed up learning in some spaces that are otherwise difficult to master by themselves. Exploration of simpler task spaces would naturally lead the robot to attempt actions more likely to also influence the more challenging task space. In [Forestier and Oudeyer, 2016] this was exemplified in the case of a simulated planar arm with single actions representing extended motions, expressed as a Dynamic Motion Primitives (DMP). Each such DMP motion was performed from the same starting home state. In the experiment the arm could reach for different objects, where the position of each object represented its own task space. Some objects resembling sticks could also be gripped during a motion, which would allow the agent to use them as tools to reach and move other objects that would otherwise be out of reach. The chance of randomly generating a motion that would influence these out-of-reach objects is negligible, but by first focusing on how to move the sticks around, occasionally the robot would also hit one of the out-of-reach ones and create an effect in that task space. That motion could then be used as a seed for how to influence these objects, from which more precise ways of moving them around using the sticks could be found. The work was later repeated on a real robot [Forestier et al., 2017] that was able to learn to move a joystick to control another robot that would push a ball around.

### 3.2.6 Advantages and disadvantages

When considering GB for the control problem described in Chapter 2 it is clear that the method has many attractive features, but also some limiting ones. GB is a rapid method that is often able to learn to control robots of essentially any number of DoF within a reasonable amount of time, making it applicable to real physical robots [Forestier et al., 2017; Rolf and Steil, 2013]. The main issue is that it requires problems where the state an action is made from can be ignored. There are two basic cases where this happens: Either the robot is reset to a starting home state after every action, so that every action is made from the same state, or the system has properties that makes initial states irrelevant, for example if the state space is fully connected (see Sec. 2.2.4). Whether a state space is fully connected or not is outside the control of any particular experiment setup, while the home state approach could potentially be used for any robot system. The approach however has two main drawbacks. It would require the robot to always be able to reset itself into this starting state, which cannot be generally assumed. Secondly such a starting state would only allow the robot to reach task space outcomes that are directly reachable from that state.
3.3 Other notable approaches

3.3.1 Intrinsic motivation

In many machine learning problems such as the task space control learning problem there are not just one but a potentially infinite number of things a robot could learn. Very often these learning processes would also influence each other, so that knowledge in some problem area would make it easier to acquire knowledge in another. Learning to walk is for example easier to someone who has first learned to stand up, compared to someone who is stuck lying down. This becomes relevant when an agent can choose between many different tasks or goals to attempt to learn or reach. It would be a waste of time to attempt to learn to walk before knowing how to stand, although this might not be initially obvious to an agent starting from tabula rasa.

One method for prioritising learning tasks is to use Intrinsic Motivation (IM) [Schmidhuber, 2010]. The general idea of IM is to frame the prioritisation between different learning problems as a learning problem in itself. Many different approaches for how to best do this have been proposed, from the Free-Energy formulation of [Friston, 2010] where the purpose is to minimise future surprise, to Empowerment [Karl et al., 2017; Salge et al., 2014] where an agent should optimise its ability to affect the world in the future, and finally perhaps the most widely used: Artificial Curiosity (AC) [Oudeyer et al., 2007], where the goal is to optimise the learning progress by identifying the tasks on which the agent improves the fastest. This idea is closely related to a well known phenomenon from child psychology called the “Zone of proximal development” [Vygotsky, 1978], which describes how the most efficient way to teach a child is at a level that is slightly out of his or her reach; neither trivial, nor completely incomprehensible.

AC and similar approaches have been applied to many different robotic tasks and used in combination with both GB and RL. For GB this has been the selection of both task space goals [Baranes and Oudeyer, 2013], as well as in the selection of task spaces to place such goals in to begin with [Forestier et al., 2017; Forestier and Oudeyer, 2016]. In RL, AC has been used in work such as [Frank et al., 2014; Houthooft et al., 2016].

To give an example of how AC can be implemented in a RL setting we will focus on the formulation of [Frank et al., 2014]. The task here is to learn a transition model \( P(s'|s,a) \), that expresses the probability that action \( a \) in state \( s \) will lead to state \( s' \). Both state spaces and action spaces are discrete and finite here, identical to the model presented in Sec. 3.1.3 for Tabular model learning. The goal here is to learn this model \( P(s'|s,a) \) as efficiently as possible, and the set of tasks the agent can choose between is the different state-actions \( (s,a) \) it can examine in order to approximate the model \( P(s'|s,a) \). To do so the artificial curiosity approach puts an intrinsic reward on every state-action, which should reflect how much the model \( P(s'|s,a) \) would improve by collecting additional information on that particular state-action.

To estimate such an intrinsic reward an approximation \( P^{(i)}(s'|s,a) \) is made based on how many times each state-action \( (s,a) \) have led to the outcome \( s' \) up until the iteration \( i \). This
approximation is then updated every time the same state action \((s, a)\) is attempted, giving a new probability \(P^{(i+1)}(s'|s, a)\). The intrinsic reward for “task” \((s, a)\) is then computed by how much the approximation changed by the addition of this new data. This is calculated using the Kullback-Leibler divergence [Lindley et al., 1956] so that:

\[
R^{(i+1)}(s, a) = \sum_{s' \in S} \ln \left( \frac{P^{(i)}(s'|s, a)}{P^{(i+1)}(s'|s, a)} \right) P^{(i)}(s'|s, a)
\]

What is interesting with this reward is that when the outcome of a state-action is predictable, e.g. whenever the previous model is already quite accurate, then \(P^{(i)}(s'|s, a) \approx P^{(i+1)}(s'|s, a)\) which results in a low reward, while unexpected outcomes leads to a high reward. Depending on how the probabilities are initiated, attempting new state-actions are often expected to lead to high rewards as new data have bigger impact on estimates based on few previous data-points. Using value iteration, Algorithm 1, the expected future reward can also be included. Actions are then not only taken with respect to the immediate future reward of attempting a specific action in the current state, but attention is also taken to what probable future states will be and what can be learned from those states. By selecting actions \(a = \arg\max_{a' \in A(s)} Q(s, a')\) the agent thus uses the current model \(P^{(i)}\) to move to the states from which it can learn the most, while at the same time choosing the path that optimises the quality of the model itself.

3.3.2 Embodiment

Although developing control behaviours to physical robots with many DoF acting in high dimensional state spaces may seem like a daunting if not impossible task, this is the standard case for most biological creatures. Many biological creatures are able to perform complicated and robust behaviours, apparently without advanced brains or cognitive computational power. One idea of how this can be done is that creatures outsource computation to their body, to the way their bodies respond to an environment. This idea is often referred to as morphological computation [Müller and Hoffmann, 2017]. It has been shown that many important and challenging behaviours can be easily solved if the embodiment is right. One example is the “passive walker” [McGeer, 1990], which is a brainless, motorless structure that is able to do downhill bipedal locomotion just by mechanical design. Such effects however rely on a good bodily design to begin with. To implement it for robotics a possible approach is to evolve the body of an agent rather than brain, in order to achieve some task. One such example is [Cheney et al., 2014] where simulated creatures were developed using a set of distinct voxels (3-dimensional pixels), each with a different property. Some acted as bones or fat, others as muscles following predetermined control policies of when to contract. Each creature was then given a score based on how far they would naturally move given some time. By evolving the creatures to optimise this score led to bodies to which the pre-given control policies created an efficient gate.

In most cases there is no feasible way in which a robot body can be altered for more efficient training. Instead the focus needs to be how to make the best out of a given morphology. Sometimes
a given body can be simplified through the use of synergies, where many behaviours and skills can be projected on low dimensional manifolds without much loss [Brown and Asada, 2007; Ciocarlie and Allen, 2009; Höfer et al., 2010; Narioka et al., 2015]. A related idea for simplified dynamics is to start with some DoF “frozen” which are then activated once basic control has been achieved by the accessible DoF [Stulp and Oudeyer, 2018; Yamaguchi et al., 2013]. In [Der and Martius, 2012, 2015], behavioural manifolds were naturally found by observing how the environment interacted with a given body, and enhance the modes of movement this interaction led to. A humanoid agent landing on its feet would feel a wave of motions starting in the feet, bending the knees, moving to the hips, the back, the arms, etc. This gives the agent important information on how different body parts relate to each other, and could this wave of motions be re-generated by the agent but in reverse it would lead to an almost perfect jumping motion. By observing such effects and recreating them in a simple sensory-motor loop, it was shown that high dimensional agents would naturally start moving in naturally resonating modes of motions. Such motions automatically lead to behaviours such as turning a crank or crawling, and all of this happened without any specific goals but simply as the effect of how motions in a body resonated with the response from the environment.

3.3.3 Associative networks/reservoir computing

Another interesting approach to robot control problems is associative networks, such as reservoir computing [Reinhart, 2011; Reinhart and Steil, 2009]. It typically deals with situations where actions are goal states, so that \( A = S \), and outcomes are a direct function of the action (or equivalently state) \( \xi : A \rightarrow X \). This is the same situation as for GB, and a typical example of this is a planar arm controlling its posture, where the end effector position defines the task space.

Associative networks are unique in that they do neither learn a forward nor an inverse model, but in a way both at the same time. An associative network can learn and generalise over combinations \((a_i, x_i)\), which are seen as single data points in the space \( A \cup X \). Given a combination \((a, x)\) the network will generate the sample \((a', x')\) that is closest (by some metric) to the set of training data of the network. This ideally leads to a function \( N : A \cup X \rightarrow (A \cup X)' \), where \((A \cup X)'\) represents the set of actions and task space outcomes that can physically be observed together.

Once trained, the network can be used both as a forward, and an inverse model. In the forward model case we want to approximate the outcome \( x^* \) given an action \( a^* \). We then pass a configuration \((a^*, x_0)\) into the network, where \( x_0 \) is an initial guess for \( x^* \). The network returns:

\[
N(a^*, x_0) = (a_1, x_1)
\]

which is the closest configuration in \((A \cup X)'\) to \((a^*, x_0)\), meaning that \( \xi(a_1) = x_1 \). This is not exactly the answer we are looking for, but \( x_1 \) is a better estimate to \( x^* \) than \( x_0 \). We then simply repeat Eq. (3.16) but with \((a^*, x_1)\). By always passing combinations \((a^*, x_i)\) to the network, where \( a^* \) is held fixed and \( x_i \) is updated, the network will finally converge to a combination \((a^*, x^*)\) for which \( N(a^*, x^*) = (a^*, x^*) \). Our forward estimate for the effect of action \( a^* \) is thus \( x^* \).
This mechanism can be used in the exact same way to find an inverse estimate to what action $a^*$ to use, in order to generate a task space outcome $x^*$. The only difference is that we in this case instead lock the task space part $x^*$ and let the action part $a_i$ update. To initialise this action part it is common to start with the agents current state so that $a_0 = s$ (provided that actions are goal states). This means that if there are multiple goal states that would result in the outcome $x^*$, the network would converge on attempting to reach the state that is most similar to the agent’s current state [Reinhart and Rolf, 2013].

What is important to note here is that associative networks does not encounter the consistency problem (Sec. 3.2.2) when dealing with cases where multiple actions $\{a_1, a_2, \ldots\}$ leads to the same outcome but interpolations of those actions do not. Since the network is trained on actions and outcomes together, there is no risk of mixing actions with similar outcomes as they are separated in the action space.

Associative networks provide no clear mechanism by themselves for how to collect data, and it should be noted that attempting to completely cover every $(a, x)$ combination is impossible, given the restrictions of Sec. 2.3.2. One way around this is to first observe that we are not necessarily interested in approximating the outcome of every action, but mainly want to know what actions to do to get a given task space outcome. In [Reinhart and Rolf, 2013] this was achieved for a 10 DoF planar arm, by first learning two separate inverse models $\pi_1(x), \pi_2(x)$ using continuous GB (as described in Sec 3.2.4). An associative network was then trained using data $(\pi_i(x_j), x_j)$ generated by these two policies, for different goals $x_j$ and using both policies, allowing $(A \cup X)'$ to contain both manifolds $(\pi_1(x), x)$ and $(\pi_2(x), x)$. Of these manifolds, one was developed using a home state with an “elbow down” posture, while the other was developed using an “elbow up” posture. This allowed both manifolds to reach most of the task space $X$, while each could also reach some parts the other could not. Once trained the associative network could be used as an inverse model which would follow one of the manifolds $(\pi_i(x), x)$ to any goal $x^*$ that could be reached by that solution branch. If asked to reach a goal that was only reachable by the other solution branch it would switch to that one, and then keep it until asked to reach a goal that was only reachable by the other.

### 3.4 Summary of approaches

This chapter has presented a series of approaches which are all capable of solving problems similar but not identical to the problem laid out in Chapter 2. In short we look for a method that is ideally able to: generalise from the sparse data a physical robot can collect in a relatively short time; handle high dimensional continuous action-spaces; handle high dimensional continuous state-spaces; plan sequences of actions in order to reach a goal; learn even if the agent cannot reset itself to some initial state; adapt to variable goals $x^* \in X$; and potentially also follow trajectories in the task space; and learn many task spaces $\{X^{(1)}, X^{(2)}, \ldots\}$ in parallel. These last two are optional to solving the task space control problem. Table 3.1 shows the abilities to
meet these demands for the most suitable approaches mentioned in this section.

It is clear from Table 3.1 that no method is able to handle all challenges by itself. Often the ability to solve one problem is at the expense of another. Tabular RL is for example able to learn a model using limited data, but only if the model to learn is also limited with few discrete states and actions. GB is on the other hand able to handle large continuous action spaces, but at the expense of planning and accounting for the effect of the state an action is made in. In the next chapter we will see that even though neither the tabular RL nor the GB approach are able to solve the problem by themselves, they can be combined in such a way so that tabular RL can be used to plan transitions between a limited set of independent GB modules, in which the state can be ignored with respect to the outcomes of actions. This allows each GB module to be trained using regular GB techniques to reach different parts of the task space, while the planning between the modules can be learned using regular tabular RL approaches. This effectively combines the strengths of the tabular RL and the GB approaches, where the ability to follow smooth trajectories in a task space, or to learn multiple task spaces in parallel, depends on what of the 2 branches of GB modules is being used.

<table>
<thead>
<tr>
<th>Sparse data</th>
<th>Action space</th>
<th>State space</th>
<th>Planning</th>
<th>No reset</th>
<th>Variable goal $x^*$</th>
<th>Trajectories ${x^<em>_i, x^</em>_{i+1}, \ldots}$</th>
<th>Multiple task spaces $X^{(i)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct policy search</td>
<td>Value based</td>
<td>Tabular</td>
<td>Continuous model</td>
<td>Home state</td>
<td>Continuous manifold</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reinforcement Learning</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goal babbling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: The typical ability of different approaches to handle some different challenges, associated to the problem described in chapter 2. The “×” indicates ability.
So far we have seen that Goal Babbling (GB) approaches are able to learn to reach task space goals within a relatively short time, even for action spaces of high dimensionality. The main drawback here is that it is only applicable if the state can be ignored when considering task space outcomes of actions. This might either be the case due to some intrinsic properties of the system, or because every action is made from the same home state that the agent can return to after every action. For many robot problems however these assumptions are not applicable. Generally the outcome of an action does depend on the state it was made in, and it is not always trivial to find a single home state a robot is always able to return to by itself, from which most of the task space can be reached.

In this chapter we introduce a new approach where we show that although GB is not applicable by itself to problems where the state of the agent has to be taken into account, it is possible to generalise the GB problem in such a way so that it can be used as local building blocks in a bigger framework. Each building block is able to reach a particular part of the task space, and tabular Reinforcement Learning (RL) can then be used to plan transitions between the building blocks to reach particular parts of the task space. This makes the solution hierarchical in the sense that a RL agent plans transitions between a set of GB agents, which are each responsible for controlling outcomes in their part of the task space. The solution is thus structurally similar to other hierarchical solutions such as MOSAIC [Haruno et al., 2001], sequential composition [Burridge et al., 1999; Le Ny and Pappas, 2012], and hierarchical RL approaches [Hengst, 2012] such as Feudal networks [Dayan and Hinton, 1993; Vezhnevets et al., 2017]. The method proposed here however diverges from these in some important ways. MOSAIC also requires forward dynamic models to be approximated, which runs into the problem of dimensionality. Sequential composition has so far almost exclusively relied on handcrafted behaviours. Feudal networks differ in the sense that, if our method is a hierarchy in motor space (since the GB
modules simplifies the action space to the planner), then feudal networks are hierarchies in time, where lower level modules seek policies to carry out over many iterations using only a small set of discrete actions.

To introduce this new approach, this chapter is divided into two different parts corresponding to different levels of abstraction. In the first part Sec. 4.1 we present the general framework of the solution, its different parts and how they interact to produce a solution to the general task space control problem, presented in Chapter 2. In the second part, Sec. 4.2, we present possible implementations to each of these different parts. In the next Chapter 5 we will see how these particular implementations can then be learned online and in parallel using only the sparse data a real robot is be able to collect “within a lifetime”.

4.1 Theoretical solution for the general case

In this section we will describe the structure of the approach that will be used to solve the general task space control problem (where actions are not necessarily goal states). This will be done using a set of GB domains \( \{ D_n \}_{n=1}^N \) from which the agent can reach a corresponding task space \( X_n \subseteq X \), together with 4 different control policies: Recovery, Transition, Planning and Reach-out. Given a starting state \( s \) and a goal \( x^* \), these domains and policies can be used as follows:

1. The agent will use the recovery policy to reach some domain \( D_n \) from initial state \( s \).
2. It then selects a goal domain \( D_{\hat{n}} \) using the planning policy.
3. It moves to the goal domain \( D_{\hat{n}} \) using the transition policy.
4. By repeating step 2-3 the agent travels from domain to domain, until it reaches a domain \( D_{n^*} \) for which \( x^* \in X_{n^*} \).
5. Finally in domain \( D_{n^*} \) it uses the reach-out policy to reach \( x^* \).

The idea presented above is also later illustrated in Fig. 4.1c.

The purpose of this initial section is to present a theoretical structure of this framework together with its interacting parts, and show how such a framework solves the task space control problem. We will thus not consider in this section how each of these domains can be practically implemented which is left for the following Sec. 4.2.

Before presenting the theoretical framework for how to solve the task space control problem it would be valuable to remind ourselves what that problem is (originally described in Chapter 2).

4.1.1 The task space control problem

Consider an agent in some state \( s_t \in S \) at time \( t \) with some task space goal \( x^* \in X \) that the robot should attempt to reach/achieve. Given an action \( a_t \in A \), the robot’s dynamics are characterised
by two different forward functions. One describing the change in state:

\[ f(s_t, a_t) = s_{t+1}, \]

and one describing the outcome in task space:

\[ \zeta(s_t, a_t) = x_{t+1}. \]

Sometimes these will be written together in a forward dynamics function \( F(s, a) \equiv \begin{bmatrix} f(s, a) \\ \zeta(s, a) \end{bmatrix} \), so that:

\[ F(s_t, a_t) = \begin{bmatrix} s_{t+1} \\ x_{t+1} \end{bmatrix} \]

or if multiple task spaces \( \{X^{(1)}, X^{(2)}, \ldots\} \) are considered with corresponding forward dynamics \( \zeta^{(i)} : S \times A \rightarrow X^{(i)} \):

\[ F(s_t, a_t) = \begin{bmatrix} f \\ \zeta^{(1)} \\ \zeta^{(2)} \\ \vdots \end{bmatrix} (s_t, a_t) = \begin{bmatrix} s_{t+1} \\ x^{(1)}_{t+1} \\ x^{(2)}_{t+1} \\ \vdots \end{bmatrix} \]

Assuming a single goal space \( X \), the goal is to find a policy \( \pi : S \times X \rightarrow A \) so that:

\[ F(s_t, \pi(s_t, x^*)) = [s_{t+1} \ x_{t+1}]^{\top} \]
\[ F(s_{t+1}, \pi(s_{t+1}, x^*)) = [s_{t+2} \ x_{t+2}]^{\top} \]
\[ F(s_{t+2}, \pi(s_{t+2}, x^*)) = [s_{t+3} \ x_{t+3}]^{\top} \]
\[ \vdots \]
\[ F(s_{t+r-1}, \pi(s_{t+r-1}, x^*)) = [s_{t+r} \ x^*]^{\top} \]

where \( \tau \) is generally limited by some upper bound time we are willing to wait in order to reach any goal \( x^* \in X \).

### 4.1.2 Local Goal Babbling domains

A key element of the proposed method is the idea that although GB cannot be used to solve the full problem (since outcomes generally depend on both the state and the action) there are \textit{local domains} within which the exact state can be ignored with respect to the outcome of an action. We call such domains “local GB domains” \( \mathcal{D}_n \) and define them as a tuple of subsets \( \mathcal{D}_n = (S_n, A_n, X_n) \), for which:

\[ \zeta(s_i, a) = \zeta(s_j, a) \in X_n \]

for all \( s_i, s_j \in S_n \) and \( a \in A_n \), where \( X_n \) is the set of achievable outcomes using such states and actions. In other words, a GB domain is defined so that the exact state (within \( S_n \)) can be ignored with respect to the outcome of any particular action (within \( A_n \)).
Figure 4.1: Example of a simple agent moving between 4 different rooms, separated by walls. In this case \( S = A = X \), where \( s = x \) is the agent’s position, and \( a \) is the position it attempts to reach.

a) The dynamics of the system, where the agent will move towards the goal \( a \) in a straight line until stopped by a wall, or reaching the goal.

b) An arbitrary example of a local GB domain \( \mathcal{D}_n = (S_n, A_n, X_n) \) in the environment. The cyan region corresponds to both \( A_n \) and \( X_n \), and are positions that can be reached in a straight line from any position in \( S_n \). The variable \( \varphi_n(s) \) highlights the inverse problem of trying to move to \( S_n \) from any given part of the state space (in a straight line). Cyan means that this is always possible, light grey means that this is sometimes possible (depending on where in \( S_n \) the goal \( a \) is selected), and dark grey means it is never possible.

c) The agent is able to move from a state \( s \) to a goal \( x^\ast \) by first doing a recovery to domain \( \mathcal{D}_4 = (S_4, A_4, X_4) \) (blue arrow), after which it transitions from domain to domain (green arrows) until \( \mathcal{D}_1 = (S_1, A_1, X_1) \) is reached. Since \( x^\ast \in X_1 \) a reach-out can be made (red arrow) that allows the agent to reach \( x^\ast \).

### 4.1.3 Control policies

Given the definition of local GB domains, assume that a finite set of domains \( \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_N\} \) with \( \mathcal{D}_n = (S_n, A_n, X_n) \) can be found so that:

1. Even if the agent is in a state \( s \not\in \bigcup_{n=1}^{N} S_n \), there is always some action \( a \in A \) that will move the agent to such a state space \( S_n \), so that \( f(s, a) \in S_n \), for some \( n \).

2. For any task space goal \( x^\ast \in X \) there is at least some domain \( \mathcal{D}_{n^\ast} \) for which \( x^\ast \in X_{n^\ast} \), meaning that \( \bigcup_{n=1}^{N} X_n = X \).

3. There is always a sequence \( \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k\} \) between any two domains \( \mathcal{D}_1, \mathcal{D}_k \), so that there is some action \( a_i \) able to bring the agent directly from \( \mathcal{D}_i \) to \( \mathcal{D}_{i+1} \) in the sequence.

In principle such a set would allow an agent to move from any state \( s \) to reach any goal \( x^\ast \). This can be done by first reaching some domain \( \mathcal{D}_n \) (1), and then transition from domain to domain until \( \mathcal{D}_{n^\ast} \) is reached (3) from which \( x^\ast \) can be reached (2). An illustration of this is shown in Fig. 4.1. To implement this scheme we propose 4 different kinds of sub-policies that will be referred to as the “recovery policy”, “transition policy”, “planning policy” and the “reach-out policy”.

38
Recovery policy

The goal of the recovery policy is to bring the agent from a state outside of any local GB domain, into such a domain. Consider an agent in a state $s$ for which:

\[(4.7) \quad s \notin \bigcup_{n=1}^{N} S_n\]

We then search a policy $\mathcal{R} : S \rightarrow A$ so that:

\[(4.8) \quad f(s, a \sim \mathcal{R}(s)) \in S_n\]

for some $n$. The recovery policy in other words selects an action that makes the agent reach a new state which is part of some local GB domain.

Transition policy

The transition policy $g : S \times \mathbb{N} \rightarrow A$ selects actions in an attempt to move the agent to a specific domain $\mathcal{D}_n$ which implies reaching its state space $S_n$. A successful transition thus means that:

\[(4.9) \quad f(s, a \sim g(s, n)) \in S_n\]

This is very similar to the recovery policy, with the main differences that a specific domain is targeted, and that the policy is not assumed to always be successful. Since a domain can generally only be reached from a subset of states, the policy will necessarily fail if starting outside of this subset. Fig. 4.1b shows this for a transition policy that randomly selects a goal position $a \in S_n$ that the agent attempts to reach in a straight line. The colour of $\varrho_n(s)$ shows the probability that such a policy will be successful from a particular starting condition.

Planning policy

Since each domain $\mathcal{D}_n$ has a state space $S_n$ an agent must reach in order to be considered to be in that domain, we can express a probability to move between two domains $(\mathcal{D}_n, \mathcal{D}_{\hat{n}})$ using the transition policy $g(s, \hat{n})$ with success probability $\varrho_{\hat{n}}(s)$ from state $s$:

\[(4.10) \quad P(n, \hat{n}) = \int_{S_n} \delta(s) \varrho_{\hat{n}}(s) \, ds\]

In other words, $P(n, \hat{n})$ expresses the probability that an attempted transition to domain $\mathcal{D}_{\hat{n}}$ will be successful if starting from a random state $s \in S_n$ (by some given distribution $\delta(s)$) of domain $\mathcal{D}_n$. This allows for an abstraction where we view each domain $\mathcal{D}_n$ as a discrete state $n$ in an Markov Decision Process (MDP). To avoid confusing these discrete states $n \in \mathbb{N}$ from the continuous states $s \in S$, we will refer to these discrete states $n$ as "nodes".

Given this set of nodes $\{n\}_{n=1}^{N}$, associated probabilities $P(n, \hat{n})$, and a goal $x^* \in X_{n^*}$, we search a planning policy $\Pi : \mathbb{N} \times X \rightarrow \mathbb{N}$ that decides what node:

\[(4.11) \quad \hat{n} \leftarrow \Pi(n, x^*)\]
to try to reach next given current node $n$ and a goal $x^*$. The ultimate goal of this planning is to eventually reach a node $n^*$ for which $x^* \in X_{n^*}$. The green arrows of Fig. 4.1c provide one such example.

Reach-out policy

To finally reach a goal $x^*$ once in a domain $\mathcal{D}_n$ for which $x^* \in X_n$, we observe that any local GB domain $\mathcal{D}_n = (S_n, A_n, X_n)$ fulfils the prerequisite for using GB by definition. Since:

$$\xi(s_i, a) = \xi(s_j, a) \in X_n$$

for all $s_i, s_j \in S_n$, and $a \in A_n$, and $X_n$ includes the all reachable outcomes, it is possible to find a reach-out policy $\pi_n : X_n \rightarrow A_n$, so that:

$$\xi(s, \pi_n(x^*)) = x^*$$

for any $s \in S_n$ and goals $x^* \in X_n$. The policy $\pi_n$ thus acts as a local inverse model in domain $\mathcal{D}_n$.

4.1.4 Global task space control

Given a set of domains $\{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_N\}$ with corresponding control policies, we are now ready to express a solution to the main control problem presented in Chapter 2 and restated here in Sec. 4.1.1. This solution is a global goal-conditioned policy:

$$\pi(s, x^*) = \begin{cases} 
\mathcal{R}(s), & s \notin S_n, \quad \forall n \quad \text{(Recovery)} \\
g(s, \hat{n}), & s \in S_n, \quad x^* \notin X_n \quad \text{(Transition)} \\
\sim \hat{n} \sim \Pi(n, x^*) & \quad \text{(Planning)} \\
\pi_n(x^*), & s \in S_n, \quad x^* \in X_n \quad \text{(Reach-out)}
\end{cases}$$

This policy is able to move from any state $s$ to any goal $x^*$ by first doing a recovery to some domain $\mathcal{D}_n$, after which it transitions from domain to domain using the transition policy, in an order decided by the planning policy. It finally reaches a domain $\mathcal{D}_n^*$ for which $x^* \in X_{n^*}$, allowing the reach-out policy $\pi_n^*(x^*)$ to be used to reach the task goal $x^*$. Algorithm 3 gives a general overview of what this process might look like.

4.2 Practical solution for particular case

In this section, we will discuss different ways to implement the domains and policies of the proposed framework. To do so we will move away from the most general description presented in previous sections, towards more particular choices and robot setups on which the suggested implementations are based.
**Algorithm 3** Pseudo algorithm

1: **while** Running **do**
2: Read current state $s$
3: Read current goal $x^*$
4: Find $n$ so that $s \in S_n$
5: **if** $n$ is undefined **then**
6: Execute recovery action $a \sim \mathcal{R}(s)$
7: **else if** $x^* \in X_n$ **then**
8: Execute reach-out action $a \sim \pi_n(x^*)$
9: **else**
10: Get goal node $\hat{n} \leftarrow \Pi(n, x^*)$
11: Execute transition action $a \leftarrow g(s, \hat{n})$

### 4.2.1 Scope

**The robot**

In this work we consider robots controlled by goal state based control, as described in Sec. 2.2. This means that the inner controller of the robot accepts goal postures $\hat{q}$, corresponding to a set of joint angles of the robot which the controller will attempt to reach. Each joint is independently controlled, and there is no upper bound on the number of such joints, although it is assumed to be finite.

**State space**

We will consider state spaces $S$ that are completely, or partially, defined by the posture space $Q$. This means that $S = Q$ or $S = Q \cup \Omega$, where $\Omega$ is some other space, for example the direction of gravity in the case of a humanoid robot (see Fig. 2.2). This means that $s = q$ or $s = [q^\top \omega^\top]^\top$ depending on the state space, where $q \in Q$ and $\omega \in \Omega$.

**Action space**

As discussed in Sec. 2.2, for this work, we will consider robotic systems characterised by goal state control. This means that our action is to choose what state $\hat{s}$ we want the robot to try to move to next, giving us $A = S$. We thus have forward functions:

$$F(s, \hat{s}) = \begin{bmatrix} f(s, \hat{s}) \\ \zeta(s, \hat{s}) \end{bmatrix} = \begin{bmatrix} s' \\ x' \end{bmatrix}$$

It is important to keep in mind that although actions are expressed as states, they are in another sense only the signal that is sent to a robot’s controller in order for it to generate motions. It would be perfectly reasonable to select goals states $\hat{s}$ that might not even make physical sense, as long as such goals influenced the robot controller to act in a desirable way with respect to the resulting new state $s'$ and task space outcome $x'$. 

41
Task space

We will consider task spaces $X$ that captures some low dimensional feature of the robot’s state, for example the 2D end effector position in the case of a planar arm. This means that we have a task space position $x$ that is a function of the current state $s$ of the robot, related by some function $\psi: S \rightarrow X$ so that:

\begin{equation}
    x = \psi(s)
\end{equation}

We will furthermore only consider one task space $X$ at a time, although generalisations to learning multiple task spaces $\{X^{(1)}, X^{(2)}, \ldots\}$ in parallel will be discussed occasionally.

4.2.2 Implementing local Goal Babbling domains

Given the robotic system described in previous section, the goal is to find local GB domains $\mathcal{D}_n = (S_n, A_n, X_n)$, with $s_i, s_j \in S_n$ and $\hat{s} \in A_n$, so that:

\begin{equation}
    \xi(s_i, \hat{s}) = \xi(s_j, \hat{s}) \in X_n
\end{equation}

Perhaps the easiest way to achieve this is to define $S_n$ as a single state $s_n$, so that $S_n = \{s_n\}$. This corresponds to the home state approach in the GB literature (Sec. 3.2.6), where states can be ignored since every action is made from a single home state. In this case it is easy to see that $A_n = S$ is a valid set of actions for a problem domain $\mathcal{D}_n$, since $s_i = s_j = s_n$ in Eq. (4.17).

Another possible domain $\mathcal{D}_n$ for GB problems is if $S_n = A_n$ corresponds to a fully connected state set. A fully connected state set $S_n \subseteq S$ is, as defined in Sec. 2.2.4, a set of states the agent can move freely between, so that:

\begin{equation}
    f(s, \hat{s}) = \hat{s}
\end{equation}

for all $s, \hat{s} \in S_n$. This means that the state space outcome $s' = \hat{s}$ is independent on exactly where in $S_n$ the agent started. Since:

\begin{equation}
    x = \psi(s)
\end{equation}

this also means that:

\begin{equation}
    \xi(s, \hat{s}) = \psi(\hat{s})
\end{equation}

for all $s, \hat{s} \in S_n$, so that the initial state in $S_n$ is also irrelevant with respect to the task space outcome, fulfilling Eq. (4.17). This is the situation for the continuous manifold approach of GB (Sec. 3.2.4) and it allows the agent to move directly between any two positions $x_i$ and $x_j$ in the task space, by finding states $s_i, s_j \in S_n$ so that:

\begin{equation}
    \psi(s_i) = x_i \\
    \psi(s_j) = x_j
\end{equation}

Since $s_i$ and $s_j$ are part of a fully connected state set the agent is then able to transition directly between the two, which leads to a transition from $x_i$ to $x_j$ in the task space.
4.2.3 Implementing policies

Transition policy

Since actions are goal states \( \hat{s} \), a simple way to attempt a transition to a particular domain \( \mathcal{D}_n = (S_n, A_n, X_n) \) is to uniformly randomly select a goal \( \hat{s} \in S_n \), so that:

\[
g(s, n) \sim U(S_n)
\]

This means that the selection of goal states \( \hat{s} \) is independent of the current state \( s \), although this state will influence the probability of reaching \( \mathcal{D}_n \) successfully, as we saw in Fig. 4.1b.

Recovery policy

In order to recover to a local GB domain \( \mathcal{D}_n \) when in a state:

\[
s \notin S_n, \forall n
\]

we can use the same approach as for Eq. (4.24) so that also:

\[
\mathcal{R}(s) \sim U(S_n)
\]

Here we select different \( n \) in some predetermined order, until one is reached successfully. Exactly how to decide on such an order is left to the specific implementation, but it is generally a good idea to start by attempting to reach for domains \( \mathcal{D}_n \) that are more likely to be reachable. This includes last visited domains, as well as domains with state spaces \( S_n \) or task spaces \( X_n \) close to an agent’s current position in the state or task space. If the initial attempts to recover are unsuccessful it might be necessary to start attempting to reach other \( \mathcal{D}_n \) as well. Note that it is assumed in Sec. 4.1.3 that there will always be some domain an agent can reach.

In cases where the recovery policy fails to return the agent to any domain \( \mathcal{D}_n \) within some predefined number of attempts, this is not seen as a failure of the recovery policy but of the set of domains \( \{\mathcal{D}_n\}_{n=1}^{N} \). How to deal with such a failure and update the domain set to better allow for recoveries, will be discussed in Sec. 5.1.

Planning policy

Transition between different domains can be seen as an MDP with transitional probabilities \( P(n, \hat{n}) \), if we also define a reward function. Since the goal is to reach a node \( n^* \) from which a given goal \( x^* \) can be reached, a natural choice for such a reward function is:

\[
R_{x^*}(n) = \begin{cases} 
1, & x^* \in X_n \\
0, & x^* \notin X_n 
\end{cases}
\]

This means that the agent only receives reward if in a node from which \( x^* \) can be reached.
This MDP is almost identical to the one described for Tabular RL in Sec. 3.1.3 with the main difference that the probability \( P(n, \hat{n}) \) does not capture the probability to reach any other node than the goal node \( \hat{n} \). This means that we have no estimate of where the agent might end up if the attempt to reach \( \hat{n} \) fails, which would be necessary in order to estimate future rewards after such a failure. This is however not necessarily a problem to the tabular RL approach, if we view every failed transition as the end of a particular “roll-out” (see Sec. 3.1.2). This means that any reward received after a failed transition can be ignored as it would belong to the next roll-out.

By not accounting for rewards after failed transitions it is possible to derive an optimal action-value function \( Q^*(n, \hat{n}) \) using the simplified Bellman equation (3.3):

\[
Q^*(n, \hat{n}) = P(n, \hat{n})[R^*(\hat{n}) + \gamma V^*(\hat{n})]
\]

where \( V^*(n) = \max_{\hat{n}} Q^*(n, \hat{n}) \) is the state-value function, and \( \gamma \in [0, 1] \) is the discount factor for future rewards. This leads to a slight alteration of how value iteration is done, seen in Alg. 4.

Algorithm 4 Value iteration - updated from Alg. 1

1. Get task space goal \( x^* \)
2. Initiate \( V^*(n) \leftarrow 0 \) for all \( n \in \{1, 2, \ldots, N\} \)
3. Initiate \( Q^*(n, \hat{n}) \leftarrow 0 \) for all \( n, \hat{n} \in \{1, 2, \ldots, N\} \)
4. Set threshold \( \Delta_{\text{max}} \)
5. \( \Delta \leftarrow 0 \)
6. \textbf{while} Running \textbf{do}
7. \hspace{1em} \( \Delta \leftarrow 0 \)
8. \hspace{2em} \textbf{for} each \( n \in \{1, 2, \ldots, N\} \) \textbf{do}
9. \hspace{3em} \textbf{for} each \( \hat{n} \in \{1, 2, \ldots, N\} \) \textbf{do}
10. \hspace{4em} \( R^*(\hat{n}) \leftarrow \begin{cases} 1, & x^* \in X_{\hat{n}} \\ 0, & x^* \notin X_{\hat{n}} \end{cases} \)
11. \hspace{4em} \( Q^*(n, \hat{n}) \leftarrow P(n, \hat{n})[R^*(\hat{n}) + \gamma V^*(\hat{n})] \)
12. \hspace{1em} \( v_0 \leftarrow V^*(n) \)
13. \hspace{1em} \( V^*(n) \leftarrow \max_{\hat{n}} Q^*(n, \hat{n}) \)
14. \hspace{1em} \( \delta \leftarrow |V^*(n) - v_0| \)
15. \hspace{1em} \( \Delta \leftarrow \max(\Delta, \delta) \)
16. \hspace{1em} \textbf{if} \( \Delta < \Delta_{\text{max}} \) \textbf{then}
17. \hspace{2em} \textbf{break}

18. \textbf{end while}
Reach-out policy

Once a domain $\mathcal{D}_n = (S_n, A_n, X_n)$ is reached for which $x^* \in X_n$, the goal is to find a reach-out policy $\pi_n : X_n \rightarrow A_n$ so that:

$$\zeta(s, \pi_n(x^*)) = x^*$$

Since the result of any goal state $\hat{s} \in A_n$ is independent of the exact state $s \in S_n$ from which it is attempted (by the definition of local GB domains), this problem is identical to the one of GB, described in Sec. 3.2. This means that $\pi_n(x)$ can be defined the same way as for GB, for example a nearest neighbour approach (as described in Sec. 3.2.3), or a continuous manifold approach (as in Sec. 3.2.4). Exactly how to learn these models online will be discussed in the next chapter, Sec. 5.3-5.4.
We have now seen how a set of domains $\{D_n\}_{n=1}^N$ together with policies for recovery, transitions, planning, and reach-out, can be used in order to solve the task space control problem, and we have also seen how particular implementations of these domains and policies can be made. Even though it is now clear that the presented structure can solve the task space control problem in theory, the next important question is how an agent could learn such structures and policies from scratch, considering the challenges presented in Sec. 2.3.2 of: Spare data, Large state space, Need for planning, No reset, and Risk of damage. Among these, the “risk of damage” challenge will be somewhat ignored at this point. For now we simply assume that minimising training time also leads to minimising damage, but we will later give more detail on this problem in Chapter 7.

To implement the proposed framework it is necessary for the agent to somehow learn or construct the following entities:

- A set of Goal Babbling (GB) domains $\{D_n\}_{n=1}^N$ with $D_n = (S_n, A_n, X_n)$. These domains will in turn define the transition and recovery policies, as described in Sec. 4.2.3.

- A reach-out policy $\pi_n : X_n \rightarrow A_n$ to each domain.

- The probability $P(n, \hat{n})$ to transition successfully between any two domains $(D_n, D_{\hat{n}})$.

It is important to observe here that there is a strong entanglement between the reach-out policies $\pi_n : X_n \rightarrow A_n$, and their corresponding domains $D_n = (S_n, A_n, X_n)$. A GB domain $D_n$ defines sub-spaces for which a local inverse model can be found, and $\pi_n$ is such a solution. This poses the question whether to attempt to learn the domain $D_n$ first, and then search a solution $\pi_n$ to that domain, or if it is better to learn a policy $\pi_n$ and then evaluate for what domain $D_n$ it works. Because of this it often makes sense to look at the domain and the domain solution as a single GB
Each such model corresponds to a single GB problem in literature, which is advantageous as we know that such models can be learned online for high dimensional agents with sparse training data (see Sec. 3.2), which makes them compatible with the core challenges of Sec. 2.3.2.

In addition to the GB models, we also have a model in the form of probabilities $P(n, \hat{n})$. These capture the mechanics of the Markov Decision Process (MDP) that is generated from domains $\{\mathcal{D}_n\}_{n=1}^N$ and transition policies $\{g_n\}_{n=1}^N$ (which are defined by the domains, see Eq. (4.24)). For this reason $P(n, \hat{n})$ will be referred to as the MDP model. Since the number of nodes $\{n\}_{n=1}^N$ is finite such a model can then be learned online without the need for massive amount of data, using the tabular Reinforcement Learning (RL) problem (see Sec. 3.1.3).

Finally, given the definition of a GB model $\mathcal{M}_n$ and the MDP model $P(n, \hat{n})$, there is a final structure that must be autonomously learned for the framework. This is the set of models $\{\mathcal{M}_n\}_{n=1}^N$ itself. It defines the GB models directly, and the MDP model indirectly as the MDP is generated by the domains $\{\mathcal{D}_n\}_{n=1}^N$. How can such a model set be learned online, to enable an agent to reach any task space goal $x^*$ from any starting state $s$? How do we ensure that the agent is always able to make a state recovery to some domain $\mathcal{D}_n$, that every node $n$ in the MDP can be reached, and that any goal $x^*$ can be reached by at least one policy $\pi_n$? We thus conclude that the creation of this set of models is a learning problem in itself which will also need to comply with the challenges of Sec. 2.3.2.

In the following Sec. 5.1 - 5.4 we will see how the GB models, MDP model and the set of models, can be learned online and parallel.

## 5.1 Online learning of set of models

The set of models $\{\mathcal{M}_n\}_{n=1}^N$ defines, as previously mentioned, both the GB models and the MDP model. GB models are defined by the initialisation of each new model $\mathcal{M}_n$, and the MDP model by the dynamics that follows from a given set of domains $\{\mathcal{D}_n\}_{n=1}^N$, provided the transition policies of Eq. (4.24). How can this set be learned and updated online, together with the GB and MDP models it generates, in order for the complete framework to work as intended? In this work this is done by formulating objectives that the model set is expected to fulfil, where failures to meet these expectations can be used as the learning signal by which the set can be updated.

### 5.1.1 Objectives

In order to evaluate to what degree a set $\{\mathcal{M}_n\}_{n=1}^N$ is adequate, we summarise three properties the final framework should ideally have:

1. **Recovery objective**: The recovery policy $\mathcal{R}$ should always be able to move the agent into
some GB domain $\mathcal{D}_n$ so that:

\begin{equation}
\exists n : f(s,a \sim \mathcal{R}) \in S_n, \forall s \in S
\end{equation}

2. Transition objective: Given some predefined lowest acceptable probability $0 < \rho < 1$, there should always be some sequence $(n_1,n_2,\ldots,n_\tau)$ between any two nodes $(n_1,n_\tau)$ so that:

\begin{equation}
P(n_1,n_2)P(n_2,n_3)\ldots P(n_{\tau-1},n_\tau) > \rho
\end{equation}

3. Reach-out objective: Any goal $x^* \in X$ should be reachable by some local inverse model $\pi_n : X_n \rightarrow A_n$, which is true if:

\begin{equation}
X_1 \cup X_2 \cup \cdots \cup X_N = X
\end{equation}

These can in short be summarised as: the agent is always able to return to some domain $\mathcal{D}_n$ (1), it has a sufficiently high probability to successfully move between any two domains $(\mathcal{D}_n,\mathcal{D}_{n_\tau})$ through some sequence of transitions (2), and any goal $x^* \in X$ can be reached from at least one domain (3).

5.1.2 Identifying and adjusting for insufficiencies

Note that this is our ideal framework. There is no guarantee this can actually be realised, but should rather be thought of as a tool in the development of the models. Whenever either assumption (1)-(3) fails, this is a signal to us to update the set of models somehow in order to make the assumptions hold again. Examples of such failures are:

1. If the agent is unable to recover to any domain, we interpret this as a violation of the recovery objective. Two ways to deal with this is to either create a new GB model $\mathcal{M}_n$ with a domain that includes the agent's current state so that $s \in S_n$. Alternatively an existing GB model can be redefined to the same effect. Since the agent is then once again in the domain of a GB model, the recovery was successful.

2. Should the MDP-model $P(n,\hat{n})$ predict that the probability to transition to some node $n_\tau$ falls below $\rho$, this can be seen as a violation of the transition objective. Two ways to deal with this are to either remove domains that cannot be reached with satisfactory probability (to search for better alternative domains), or to attempt to find new intermediary domains that can be used as stepping stones for reaching the hard-to-reach domains.

3. If a goal $x^*$ is outside the union of all regions $\{X_n\}_{n=1}^N$, this is a violation of the reach-out objective. One way to deal with this is to add new GB models whenever the agent reaches a task space position $x$ that does not belong to any previous region $X_n$. These new models are then initiated around a state that is already reaching some new parts of the task space. Ideally this will step-by-step lead the model set to cover the full task space.
At the heart of this is the interaction between the abilities of individual GB models and the combined effect of these as a set. Because of this connection, properties of the full model set can be manipulated by the models at an individual level. This is the approach which is taken in this work, where the goal will be to develop each individual model with the purpose of creating beneficial effects for the model set as a whole. Exactly how to do this will therefore be left to the description of how to develop the individual GB models, which is covered in Sec. 5.3-5.4.

5.2 Online learning of MDP

Before going into how GB models can be learned online by exploration, we start with the MDP model $P(n, \hat{n})$ that attempts to capture the probabilities to use the transition policy successfully to move between different GB models. Keep in mind that these GB models will also evolve over time, meaning that the “true” transition probability we will try to approximate will also evolve over time.

5.2.1 Initialisation

Our goal with finding a model $P(n, \hat{n})$ is not necessarily to approximate the true probabilities as well as possible, but rather to find a model that produces functioning transition sequences between nodes. One effect of this is that we do not necessarily care to find the probability to move between any two nodes $(n, \hat{n})$ in the set, as long as all nodes are connected by some passable sequence of nodes. This allows us to limit the number of transitions for which we need to approximate $P(n, \hat{n})$. Without such a limitation, learning the probability to move directly between any two nodes would require in the order of $N^2$ different transitions to approximate, given $N$ nodes. If we instead limit the number of possible transitions from each node to some number $k$ (typically the closest neighbours by some measure), this changes to the order of $kN$ different transitions. The number of transitions to learn thus scales linearly with the number of nodes.

Using this idea we can initiate the model so that:

$$
\begin{align*}
P(n, \hat{n}) & \left\{ 
P_0 \quad n \rightarrow \hat{n} \text{ allowed} \\
0 \quad n \rightarrow \hat{n} \text{ forbidden}
\end{align*}
\right.
$$

$P_0 > 0$ is here some initial assumed success probability constant. Note that this initialisation is made every time a new node is created, which is based on the evolution of the set of models $\{\mathcal{M}_n\}_{n=1}^N$ and can happen at any time during the learning process. When new nodes are introduced the new node will connect to the $k$ closest neighbours by Eq. 5.4, which means that previous nodes would get slightly more than $k$ edges. In the same manner might nodes get less than $k$ connections if a GB model is removed. We will accept these cases.

If the agent picks actions $\hat{n}$ greedily this initialisation will be sufficient, since any forbidden transition is assumed to fail (and end the roll-out, as discussed in Sec. 4.2.3). If goal nodes $\hat{n}$
are chosen some other way however, for example randomly, it would be necessary to exclude transitions initialised to 0 from being selected.

### 5.2.2 Learning

To update the model \( P(n, \hat{n}) \) we observe attempted transitions between two nodes \( n \) to \( \hat{n} \), which leads to new states \( s' \) when executed by the transition policy. This final state \( s' \) may or may not belong to some domain \( \mathcal{D}_n \). The feedback from this transition is then used to update the model so that:

\[
\begin{align*}
P(n, \hat{n}) &\leftarrow (1 - \alpha)P(n, \hat{n}) + \alpha I(s' \in \hat{S}_n) \\
&= P(n, \hat{n}) - \alpha I(s' \in \hat{S}_n) + \alpha I(s' \in S_{\hat{n}})
\end{align*}
\]

where \( I(s' \in \hat{S}_n) = 1 \) if \( s' \in \hat{S}_n \), and 0 otherwise, and \( 0 < \alpha < 1 \) is the decay factor which decides the rate with which the approximation is updated. Note that if \( s' \) would be in some other domain \( \mathcal{D}_{n'} \) we do not update \( P(n, n') \). The simple reason for this is that \( P(n, n') \) expresses the probability of reaching \( n' \) from \( n \) when trying to do so, and not in the pursuit of some other goal node \( \hat{n} \). The choice of \( \alpha \) here balances how rapidly we want the model to change. Smaller choices of \( \alpha \) are better at approximating stable underlying probabilities, but at the cost of requiring more samples to approach this value. Larger \( \alpha \) adapts faster to any change in the true probability, but at the cost of accuracy.

### 5.2.3 Exploration

We have now seen how transitional probabilities can be approximated by observing the outcomes of attempted transitions to various goal nodes \( \hat{n} \). The next question is how to choose these goal nodes \( \hat{n} \) given a current node \( n \) in order to collect data with which the MDP model \( P(n, \hat{n}) \) can be improved, using Eq. (5.5). This is the exploration problem. What is important to remember here is that transition probabilities do not simply change over time, but change as a consequence of the exploration itself. Whenever we attempt to reach a node \( \hat{n} \), we effectively train the corresponding GB model \( \mathcal{M}_{\hat{n}} \) since each such attempt also produces data which can be used to improve the GB model. This generally means that attempting a particular transition \( n \rightarrow \hat{n} \) naturally also improves the “true” probability to do that transition. To choose \( \hat{n} \) therefore not only includes the choice of what transition probability we want to approximate better, but also what GB models we want to develop.

In this work we will consider three different ways of selecting goal nodes \( \hat{n} \) in order to approximate the MDP model: Random actions, Learning by doing, and Intrinsic motivation.

**Random actions**

The simplest way from an algorithmic point of view is to simply select goal nodes \( \hat{n} \) randomly among the set of transitions we allow from a current node \( n \). This is a common approach in the exploration of MDPs, and ensures that every possible transition will eventually be attempted.
from every reachable node, given enough time. The problem is that this time might be very long
indeed, and that sampling will generally not be uniform. Assume, for example, \( N \) nodes where
every failed transition would return the agent to some initial node \( n_0 \). Transitions related to this
node, or to nodes that are easy to reach in a few transitions from that node, will be much more
thoroughly evaluated than transitions related to nodes that require a series of specific transitions
from \( n_0 \) to be reached to begin with. As an example, imagine some nodes corresponding to a robot
lying down, while other forms a sequence for standing up. Every time it fails it falls down to the
lying down node. It is unlikely that such a robot would randomly select the sequence of nodes
that allowed it to stand up within any reasonable time, since any deviation from that sequence
would make it fall down again. The robot would thus collect most of its data lying down.

**Learn by doing**

A different approach that is also easy to implement, is the “Learn by doing” approach. Here the
MDP model is trained by attempting to reach randomly generated goals \( x^* \in X \) for a predeter-
mined number of iterations. This is not different from the behaviour we want the agent to be able
to perform in the end, which means that we can simply select goal nodes \( \hat{n} \) using policy \( \Pi(n, x^*) \)
as defined in Eq. (4.27).

This is a greedy policy, and thus deterministic with respect to any given iteration of MDP model
\( P(n, \hat{n}) \). This means that there might be transitions we will never attempt with this approach,
as long as some other sequence is able to accomplish the same thing. As an example, assume 2
possible transitions between nodes, approximated to have success rates of \( P_1 \) and \( P_2 \). The planner
attempts the transition with highest probability, for example transition 1 if \( P_1 > P_2 \). This leads
to an attempt that provides new data, that is used to update \( P_1 \) somewhat closer to its “true”
value. As long as this updated value \( P_1 \) value does not fall below \( P_2 \) the agent will never even
attempt transition 2, although it might in fact have a higher success probability. Should the
first transition constantly fail however, then at some point \( P_1 < P_2 \) and the agent would start
attempting the other transition. This means that the agent will always eventually find some path
between any two nodes (as long as such a path exists), but it might not be the most efficient or
reliable one.

**Intrinsic motivation**

Finally we can consider an approach with the purpose of learning the probabilities of the MDP as
efficiently as possible. *Intrinsic motivation* was discussed in Sec. 3.3.1, and the general idea is
to tie rewards to qualities of the model itself, in order to guide the agents behaviours towards
the activities that improves the model the most. This means that an *intrinsic* reward is given
for every transition \( (n, \hat{n}) \), depending on how much the resulting data improved the complete
model \( P(n, \hat{n}) \) by some measure. A common such measure is the convergence of the approximated
probabilities towards the true underlying probabilities. This means that transitions with well
approximated probabilities will be avoided in favour of transitions where new data would improve probability estimations more.

In Sec 3.3.1 we discussed the Kullback-Leibler divergence as a potential reward for how much a particular attempt $n \rightarrow \hat{n}$ improved a model $P(n, \hat{n})$. This is however most useful if the underlying probabilities are static, as in [Frank et al., 2014]. This is not the case here, since the probability to reach a node $\hat{n}$ is based on the underlying model $M_{\hat{n}}$, which will itself be trained by data collected when trying to reach $\hat{n}$. In [Loviken, 2015] it was found in a similar case that simply rewarding changes in the probability approximations could then lead to an oscillating behaviour where the planner found a way to destroy and rebuild an underlying probability in order to receive constant reward, effectively stopping the model from improving further.

In this work we suggest another approach. We initialise an intrinsic reward for every allowed transition $n \rightarrow \hat{n}$:

$$ R(n, \hat{n}) \leftarrow R_0 $$

Then, every time a transition is successfully attempted, we decay the reward of that transition by some factor $0 < \beta < 1$. Given an attempted transition $n \rightarrow \hat{n}$ that ends up in a state $s'$, we thus have that:

$$ R(n, \hat{n}) \leftarrow \begin{cases} R(n, \hat{n}) & , s' \notin S_{\hat{n}} \\ \beta R(n, \hat{n}) & , s' \in S_{\hat{n}} \end{cases} $$

In short, if the transition fails the reward is unchanged, if it succeeds, it is decayed by $\beta$.

Consider the combined effect of updating the probability $P(n, \hat{n})$, Eq. (5.5), and the reward $R(n, \hat{n})$, Eq. (5.7), with regards to the estimated return:

$$ r(n, \hat{n}) = P(n, \hat{n})R(n, \hat{n}) $$

If the transition $n \rightarrow \hat{n}$ is attempted and fails, then $P(n, \hat{n})$ is decayed but $R(n, \hat{n})$ stays the same. This means that the overall estimated return $r(n, \hat{n})$ decreases, making the agent less interested in trying this transition again. If the transition is successful however, then $P(n, \hat{n})$ approaches 1 while the reward is decayed by $\beta$. In this case the estimated return $r(n, \hat{n})$ might either increase or decrease. If the estimated probability was already close to 1 the decay of the reward would dominate and the overall estimated return would decrease. If the estimated probability was low enough however the increase in the probability of receiving the reward would counteract the reduction of reward, leading to an overall increase in estimated return $r(n, \hat{n})$.

This reward allows us to develop an optimal action-function $Q(n, \hat{n})$ using value iteration Alg. 4, but with reward $R_{\pi^*}(\hat{n})$ replaced by the intrinsic reward $R(n, \hat{n})$. By selecting goal nodes:

$$ \hat{n} \leftarrow \arg \max_{\tilde{n}} Q(n, \tilde{n}) $$

the agent will naturally plan transitions that allow it to reach nodes from which it can attempt new transitions, while avoiding transitions that are unlikely to succeed. It is worth noting that
although transitions that always succeed do not give reward by themselves (since $R(n, \hat{n}) \to 0$), they are useful with regards to future rewards as they allow the agent to reach new nodes from which the expected return is still high.

### 5.3 Online learning of GB models (Approach 1)

We will now move away from the abstract dynamics of moving between nodes $n$ in the MDP, and focus on how the GB models $\mathcal{M}_n$ can be found, both in order to solve the GB problem of each such model individually, but also to ensure that the set of such models satisfies objectives described in Sec. 5.1.1.

The goal of individual GB models $\mathcal{M}_n = \{D_n, \pi_n\}$ is to find a domain $D_n = \{S_n, A_n, X_n\}$ and a policy $\pi_n : X_n \to A_n$ so that:

$$\xi(s, \pi_n(x^*)) = x^*$$  \hspace{1cm} (5.10)

for all $s \in S_n$ and $x^* \in X_n$. In other words to find what goal state $s \in A_n$ to attempt to reach, given a current state $s \in S_n$, in order to reach a task space goal $x^* \in X_n$. This is the same problem as for regular GB, except that we only consider a domain $D_n = (S_n, A_n, X_n)$ and not the entire $(S, A, X)$. For this reason we can look towards the GB literature for solutions. As was seen in Sec. 3.2, there are two main branches of GB:

- One where every action is made from a given starting state, the so called “home state” branch of GB.
- One where the agent moves along a continuous manifold in the state space which corresponds to different outcomes in the task space. This is referred to as the “continuous manifold” branch in GB.

Each of these branches have their own advantages, where the home state branch allows for the learning of many task spaces $(X^{(1)}, X^{(2)}, \ldots)$ in parallel, while the continuous manifold approach allows the agent to smoothly follow trajectories $(x^*_t, x^*_{t+1}, \ldots, x^*_{t+\tau})$ in the task space, as can be seen in Tab. 3.1 in Sec. 3.4.

In this work we will investigate the application of both these branches when generalized to the framework presented in this work, where multiple GB models are considered in parallel. This leads to two different versions, one where the domain of each GB model is a single home state, and one where the domains are spread out to jointly cover the full task space. These two approaches are illustrated in Fig. 5.1.

In this section 5.3 we consider the home state approach, where every domain $D_n$ is defined by a given home state $s_n$, from which every action is performed. This approach is relatively easy to implement as domains $D_n$ are easy to define and implement, while policies $\pi_n$ are pretty much
Figure 5.1: In this work we present two different implementations of the framework for global control using local GB modules. Approach 1 is based on the home state-branch of GB, so that every domain $S_n$ is reduced do a single point from which every reach-out is made. Approach 2 is based on the continuous manifold-branch of GB, so that $S_n$ is expanded to cover some region of the task space, until the combined set of models covers the entire task space. This allows the agent to reach any goal $x^*$, while still being in a region $S_n$.

untangled from the domain and can be learned independently. The main drawback with the home state approach is that the agent needs to recover to a home state after every reach-out, which means that the agent is not able to follow task space trajectories smoothly.

In the following Sec. 5.4 we will see how GB models can be found online, based on a continuous manifold approach.

5.3.1 The GB model

With the “home state” approach we simply define each domain $\mathcal{D}_n = (S_n, A_n, X_n)$ by a single home state $s_n$, so that:

$$S_n = \{s_n\}$$

As was defined in 4.1.2, any GB domain must fulfil the following relationship:

$$\xi(s_i, \hat{s}) = \xi(s_j, \hat{s}) \in X_n$$

for all $s_i, s_j \in S_n$ and $\hat{s} \in A_n$. Since $s_i = s_j$ in this case, this relationship clearly holds for any goal state $\hat{s}$, giving $A_n = A$. Finally $X_n$ represents every outcome $x$ that is achievable from $S_n$ using actions in $A_n$. Estimating this range $X_n$ is a common and challenging problem in the GB literature, see “unknown range problem” in Sec. 3.2.2. In this work we will not focus on this. Instead we will assume that $X_n$ is a disk with radius $r$ around a home state’s position $x_n = \psi(s_n)$ in the task space, so that:

$$x \in X_n \iff ||x - x_n|| < r$$

Given $X_n$ we are then looking for a reach-out policy $\pi_n : X_n \rightarrow A$ so that:

$$\xi(s_n, \pi_n(x)) = x$$

55
To find this policy any GB approach can in principle be used. Here we will use the nearest neighbour approach, described in Sec. 3.2.3. The idea is to simply save every observation:

\[ \xi(s_n, \hat{s}_i) = x_i \]

into a data set \( D_n = \{ (\hat{s}_i, x_i) \}_{i=1}^b \) for each model \( \mathcal{M}_n \). We can then estimate a reach-out policy:

\[ \pi_n(x) \leftarrow \hat{s}_{\text{min}} + \mathcal{N}(||x - \hat{s}_{\text{min}}||) \]

where \( \hat{s}_{\text{min}} \) is the sample in \( D_n \) that minimises \( ||x - \hat{s}_i|| \), and \( \mathcal{N}(\sigma) \) is a noise distribution that approaches 0 when \( \sigma \to 0 \). When asked to move to \( x \) from \( s \) the agent will thus use the previous action from \( s \) that ended up the closest to \( x \), and add some additional noise in order to allow the agent to discover potentially better estimates. In [Benureau, 2015] it is showed that this type of nearest neighbour exploration can indeed be powerful in exploring a task space.

Observe that \( X_n \) is an approximation of reachable goals from \( s_n \), meaning that some outcomes \( x \in X_n \) might actually be unreachable from the home state. The effect of this will be an asymptotic error corresponding to how close to that outcome the agent can reach. We do not have the same kind of drawback for goals outside of \( X_n \). Nothing stops us for attempting reach-outs for such goals, which can be useful when \( x^* \) is outside of the task spaces of all models \( \{ \mathcal{M}_n \}_{n=1}^N \). This allows the agent to reach towards new regions of the task space, for which no previous GB models exist.

### 5.3.2 Finding sets of GB models

The main purpose here is to find a set of GB models \( \{ \mathcal{M}_n \}_{n=1}^N \) that fulfills all objectives of Sec. 5.1. To reiterate this means that it should always be possible to recover to some state space \( S_n \), it should be possible to reach any goal \( x^* \in X \) using some reach-out policy \( \pi_n \), and it should be possible to transition between any two nodes in the MDP model that is generated by the domains \( \{ \mathcal{D}_n \}_{n=1}^N \). How can such a set be found iteratively?

Assume the robot starts in a state \( s \) with a task space position \( x = \psi(s) \). We can then define a first GB model \( \mathcal{M}_1 \) with home state \( s_1 = s \) and \( S_1 = \{ s \} \), with \( X_1 \) centred around \( x \). We can furthermore also initiate \( \pi_1 \) by initiating the dataset to \( D_1 = \{ (s, x) \} \), since \( f(s, s) = s \) and \( \psi(s) = x \). We now have a set of one model \( \{ \mathcal{M}_1 \} \). In some cases a set with one model will be enough, as long as the single model is able to fulfill all objectives of Sec. 5.1 by itself.

Sometimes it is not enough however, and this will be discovered through the failures described in Sec. 5.1.2, while attempting to reach different goals \( x^* \) using the current model set \( \{ \mathcal{M}_n \}_{n=1}^N \). The first way this can happen is if the agent reaches a state \( s' \) from which it is unable to do a recovery to any home state \( \{ s_n \}_{n=1}^N \). The second is if \( x' \notin \bigcup_{n=1}^N X_n \), i.e. if we reached a position \( x' \) not covered by any previous model. In both cases an option is to use the current state \( s \) to create a new model with home state \( s' \), so that \( S_{N+1} = \{ s' \} \) and \( X_{N+1} \) centred around \( x' \) in a new model \( \mathcal{M}_{N+1} \). This was precisely what was done when creating the first model \( \mathcal{M}_1 \). A possible problem if new models are created whenever the agent fails to recover is that it might lead to an explosion
of GB models, which is undesirable since each model will take some time to train and it could
also make the MDP model planning challenging. In this case it could make sense to remove a
previous model when adding a new one if they cover roughly the same part of the task space.
This way the number of models can be kept fixed.

An important question is how to deal with a situation where the goal \( x^* \) is outside of the task
spaces of all models, so that \( x^* \notin \bigcup_{n=1}^{N} X_n \). Note that as we previously described, there is nothing
to really stop us from using an inverse model \( \pi_n \) outside of the domain \( X_n \), as there will still be
a nearest neighbour \( x_{min} \) to that goal. The question is only from what domain we should try to
reach this goal \( x^* \). The general approach for this is to try to get as close as possible to the goal \( x^* \),
i.e. go to the domain with a task space that is closest to \( x^* \), and attempt to reach it from there.

### 5.3.3 Algorithm

With the previous sections in mind we can create a set of GB models \( \{ \mathcal{M}_n \}_{n=1}^{N} \) online, using Alg. 5.
Note that we do not discuss here how the planner makes its decisions. It is assumed to extract
necessary information from the algorithm, which includes the success of attempted transitions
as well as keeping track of addition or subtraction of GB models which represent addition or
subtraction of nodes in the MDP model.
Algorithm 5 Online learning - Home state approach

1: Decide a radius $r$ for regions $X_n$ (see Eq. (5.13))
2: while running do
3:     Observe goal $x^*$
4:     Observe current positions $(s,x)$
5:     if $x$ is outside $X_1 \cup X_2 \cup \cdots \cup X_N$ then
6:         Set current node $n$ to $N + 1$
7:         Create new GB model $\mathcal{M}_n$ with $s$ as home state $s_n$, and $x$ as center of $X_n$
8:         Initiate dataset $D_n = \{(s,x)\}$
9:     else if $s$ is not equal to any home state $s_n$ then
10:         if not too many recovery attempts have been made then
11:             Try to recover to some home state $s_{\hat{n}}$, based on the recovery policy
12:             Return to step 3:
13:         else
14:             Create a new model $\mathcal{M}_n$ based on current position $(s,x)$ as before.
15:             If necessary, remove the GB model the agent failed to reach
16:             Observe that you are in node $n$
17:     else
18:         Observe that you are in node $n$ (since $s = s_n$)
19:     Get goal node $\hat{n}$ from MDP planner
20:     if you are not in $\hat{n}$, i.e. $n \neq \hat{n}$ then
21:         Select goal state $\hat{s} = s_{\hat{n}}$ (Transition)
22:     else if $\hat{n} = n$ then
23:         Find sample $(s_i,x_i)$ in dataset $D_n$ that minimizes $||x_i - x^*||$
24:         Create goal state $\hat{s}$ based on $s_i$ and a random deviation for exploration (Reach-out)
25:     Send $\hat{s}$ to robot controller
26:     Observe new positions $(s',x')$
27:     Add $(\hat{s},x')$ to dataset $D_n$, since goal state $\hat{s}$ led to outcome $x'$ from home state $s_n$
5.4 Online learning of GB models (Approach 2)

We now turn to the continuous manifold branch of GB, where the goal is to find a state space $S_n$ the agent can move freely within, while reaching any goal in $X_n$. By spreading the task spaces $\{X_n\}_{n=1}^N$ to cover the entire task space, each task space position can then be reached using some GB model. The idea is to define each $S_n$ as a continuous manifold so that every position $x \in X_n$ in the task space maps to a particular position $s \in S_n$, with $\psi(s) = x$. If $S_n$ is a fully connected state set it then means that the agent can move between any two positions $x_i, x_j \in X_n$ with corresponding states $s_i, s_j \in S_n$, since:

$$f(s_i, s_j) = s_j$$

by the definition of fully connected state sets, see Sec. 2.2.4. Since the agent reached $x_j = \psi(s_j)$ without leaving $S_n$ no state recovery is necessary, allowing a new goal to be immediately reached. Compare this to the home state approach where the agent needed to first return to the home state $s_n$ after every reach-out in order to return to $S_n = \{s_n\}$.

Perhaps the most well known solution for how to find a continuous manifold $S_n$ is the one introduced in [Rolf et al., 2011], which was described more closely in Sec. 3.2.4. That approach however requires a predefined home state which is used as a starting point for the manifold $S_n$. In this work we do not want to rely on such bias which would also be very difficult to handcraft in advance, given that we are considering multiple GB models in parallel. For this reason a different approach was developed for this work.

5.4.1 The GB model

Our goal with each GB model $\mathcal{M}_n = (\mathcal{D}_n, \pi_n)$ is to find a domain $\mathcal{D}_n = (S_n, A_n, X_n)$ so that $S_n = A_n$ is a fully connected state space:

$$f(s_i, s_j) = s_j$$

for all $s_i, s_j \in S_n$, and where $X_n$ is the projection of $S_n$ in task space:

$$\psi(S_n) = X_n$$

meaning that there is some state $s \in S_n$ for every outcome $x \in X_n$:

$$x \in X_n \Rightarrow \exists s \in S_n : \psi(s) = x$$

This allows us to define our reach-out policy $\pi_n : X_n \rightarrow S_n$ so that:

$$\psi(\pi_n(x)) = x, \forall x \in X_n$$

The policy $\pi_n(x)$ thus maps every goal $x \in X_n$ to a state $s \in S_n$ that creates an outcome $x$ in the task space, if reached. Since $S_n$ is a fully connected set this happens every time the agent starts
from $S_n$. This in turn allows the agent to move freely within the region $X_n$ by selecting goal states $\pi_n$. If $\pi_n$ is continuous in addition that allows the agent to smoothly follow any trajectory \(\{x_1^*, x_2^*, \ldots\} \in X_n\). Consider the joint forward dynamics function:

\[
F(s, \hat{s}) = \begin{bmatrix} f(s, \hat{s}) \\ \xi(s, \hat{s}) \end{bmatrix} = \begin{bmatrix} s' \\ x' \end{bmatrix}
\]

We then get:

\[
F(s, \pi_n(x_1^*)) = [\pi_n(x_1^*), x_1^*]^	op
\]

\[
F(\pi_n(x_1^*), \pi_n(x_2^*)) = [\pi_n(x_2^*), x_2^*]^	op
\]

\[
F(\pi_n(x_2^*), \pi_n(x_3^*)) = [\pi_n(x_3^*), x_3^*]^	op
\]

\[
\vdots
\]

Each goal in the trajectory is here successively reached without intermediary steps, since the agent moves along $S_n$. By also enforcing $\pi_n$ to be continuous, each motion between any two subsequent steps in the goal sequence is contained, since a small motion in the task space then implies a small motion in the state space.

The solution manifold

To learn a continuous manifold GB model $\mathcal{M}_n$ online, we will start by first defining a space $X_n$ for which we search a continuous policy $\pi_n : X_n \to S_n$. This policy will in turn define our space $S_n$ so that for every $s \in S_n$ there exists a task space position $x \in X_n$ for which:

\[
\pi_n(x) = s
\]

This means that the space $S_n$ we search is a manifold, where every state $s$ is mapped to exactly one task space outcome $x$, and where a small motion in the task space leads to a small motion in the state space. For this reason the policy $\pi_n$ will often be referred to as the manifold, and since the manifold defines the state space $S_n$, “the state space” and “the manifold” will be used synonymously. The main challenge here will be to find a policy $\pi_n$ that is both continuous, and fully connected.

In this work the region $X_n$ will be handcrafted. Automatic ways to find a set $\{X_n\}_{n=1}^N$ will thus be left for future work. Even when handcrafted it is not obvious how to best define regions $X_n$, and what works well will depend on the problem at hand. Making $X_n$ small makes it easier to find manifolds that are continuous and fully connected, but at the price of requiring a larger number of such GB models to cover the full task space. If $X_n$ is large on the other hand, it might instead be impossible to find a manifold that is both continuous and fully connected. Imagine an obstacle in the middle of a region $X_n$. It would be impossible to move directly between different sides of the obstacle if such motions lead to collisions with it, which means that no fully connected state set can exist in this case, see Eq. 5.17.
Consistency

Like for the home state approach, in order to learn a manifold $\pi_n$ we will have to make use of observations $(s, x)$, since each such observation tells us that $\psi(s) = x$, i.e. reaching state $s$ also means reaching $x$. For this approach however, compared to the home state approach, we seek a manifold $\pi_n$ that is continuous. One way to achieve this is to use some sort of regression over a set of observed observations $D = \{(s_i, x_i)\}_{i=1}^k$, which in practice means that any specific estimate $\pi_n(x) = \hat{s}$ will not correspond to a single sample in the observation set $D$, but to some mixture of interpolation and extrapolation of the samples in that set. When doing so we run into the problem of “consistency”, also discussed in [Rolf et al., 2011]. Because of the low dimensionality of the task space, and the high dimensionality of the state space, there are generally an infinite number of states $\{s_i\}_{i=1}^\infty$ to every achievable outcome $x$, so that $\psi(s_i) = x$ for all $i$. The problem is that although these states produces the same effect in the task space, averages over the states might not, meaning that for some values $a, b > 0$:

$$\begin{align*}
\psi(s_i) = x \\
\psi(s_j) = x
\end{align*} \implies \psi\left(\frac{as_i + bs_j}{a + b}\right) = x$$

From a physical point of view these can be seen as different solution branches to $x$. An simple example is if $\psi(s) = \sin(s)$ and $s_i = 0$ and $s_j = \pi$. For robotic systems we could for example consider the orientation $x = \omega$ of a humanoid robot torso. A value $x$ corresponding to the torso pointing straight up can be achieved both by sitting up $s_i$, and standing up $s_j$. If we average over sitting and standing states, towards a posture with all the joint angles in-between, we are likely to get a posture that will make the robot fall down, leading to a completely different torso orientation. Another example is a planar arm reaching behind its own base. It can do so in a clockwise (CW) or a counter-clockwise (CCW) manner, where all joint angles are either positive or negative. Mixing these up however leads to a posture with joint angles around 0, which would point the arm straight forward, and definitely not allow it to reach around itself.

Finding consistent datasets

These are all examples of inconsistent samples that correspond to different solution branches. In order to use regression over a data set of observations $D_n = \{(s_i, x_i)\}_{i=1}^k$ we need to make sure that all the samples are consistent, so that we can meaningfully interpolate over them and find a continuous manifold $\pi_n$. In this work we loosen the definition of consistency slightly, so that two samples $s_i$ and $s_j$ are considered consistent if and only if:

$$\begin{align*}
\psi(s_i) \in X_n \\
\psi(s_j) \in X_n
\end{align*} \implies \psi\left(\frac{as_i + bs_j}{a + b}\right) \in X_n$$

for some $n$, and $a, b > 0$. This is necessary since in practice we will never find two observations with the exact same outcome $x$, while the reason we collect observations is to find a mapping for the entire region $X_n$. Note that such an approach might mean that we try to average over two
Figure 5.2: Examples of consistency by Eq. (5.25) for a planar arm with end effector positions as task space. The black arm illustrates a new observation of a posture $q'$ with an end effector position $x' \in X_n$. There is already a dataset $D_n$ of earlier postures seen in $X_n$ (shown in green). These are assumed to be consistent with each other, and the challenge is to determine if the new sample can also be added to this dataset while maintaining consistency. In this case we can see that the arm to the left is consistent with the dataset as a smooth transition to these postures can be done without leaving $X_n$. The arm to the right is, on the other hand, inconsistent as interpolations of the new posture and those in the dataset would lead to task space outcomes outside of $X_n$. Note that the arm could not have reached this posture by reaching an interpolation of earlier postures in $D_n$, but must have ended up in $X_n$ as a failure of reaching somewhere else.

samples in one part of the region, but the resulting state ends up in another part of the region. This will be taken care of by a “memory factor” which will soon be described in more detail. In practice this memory factor will successively prune the dataset, to allow the initial observations to be replaced by increasingly more consistent samples as learning proceeds.

The idea for the approach is to only add new samples $(s', x')$ to a dataset $D_n$ if the new sample is consistent with previous samples in $D_n$. It will be practically impossible to compare each new observation to every previous sample in the dataset, so instead we will rely on the following heuristics:

1. If an observation $(s', x')$ is the first observation for which $x' \in X_n$, then it is consistent.

2. If we know that an observation is consistent to at least one other sample in $D_n$, then we assume it is consistent with all of them.

3. If an agent tries to reach a region $X_n$ using a posture $\hat{q}$ and reaches both $X_n$ and (sufficiently close to) $\hat{q}$, then the sample is considered consistent with previous samples of region $X_n$.

The motivation for point 1 is simply that if a sample $(s', x')$ is the first one observed in $X_n$, it only needs to be consistent with itself. The first sample in each region can thus be used as a seed for that region. Point 2 comes from the fact that states that are consistent with each-other are generally similar to one another, and since all samples in a given set are supposed to be consistent at any time, it is likely that if a sample is similar to one sample it must also be similar
to the rest. The 3rd point comes from the fact that the goal state \( \hat{s} \) (including goal posture \( \hat{q} \)) is generated using some kind of regression, based on previous samples in \( D_n \). The goal state should thus be similar to the samples already in \( D_n \). If this similarity also made it end up in \( X_n \) it is very likely it did so in the same way as previous samples which means that it can be added. Fig. 5.2 gives an example of what this could look like.

Note that except for point 1, neither of these assumptions is guaranteed to be true. The heuristics should be seen more like the “reach by accident” approximation of Sec. 2.2.3 where it gives a first approximation of the dataset \( D_n \). We will later see how this dataset can be successively improved, but first we will see how to generate new samples to begin with.

**Generating samples**

When attempting to reach a goal \( \hat{x} \in X_n \), a goal state:

\[
\pi_n(\hat{x}) = \text{RegressionMethod}(\hat{x}, D_n)
\]

is approximated. \( \text{RegressionMethod}(x, D_n) \) here is some regression method, for example Local Linear Regression (LLR) or Gaussian Process Regression (GPR) [Melo, 2012; Weisberg, 2005], that uses the data set \( D_n \) of the region \( X_n \) the agent attempts to reach. If \( D_n \) is empty (if \( X_n \) has never been visited) the dataset of the agents current region can be used instead. The approximation \( \pi_n(\hat{x}) \) is then augmented with a noise term \( \eta \), to generate the final goal state:

\[
\hat{s} \leftarrow \pi_n(\hat{x}) + \eta
\]

This noise is added for exploration. Typically the noise would be Gaussian distributed, with an amplitude equal to the product of a random scalar \( U(0,1) \), and a second factor that scales the noise level to the number of previous samples seen in the region \( X_n \), so that less noise is used when there are more samples, and vice versa. The reason for the scalar is that when the number of degrees of freedom (DoF) of an agent increases, the norm of the noise vector approaches a mean deviation. This phenomenon is referred to as the “regression to the mean” [Everitt, 2006] and would in practice mean that the agent would never in practice attempt a goal state \( \hat{s} \approx \pi_n(\hat{x}) \), if such a scalar term was not added.

Once this noise \( \eta \) is added to the regression estimate \( \pi_n(\hat{x}) \) the goal state \( \hat{s} \) is sent to the robot controller, which moves the agent to a new state \( s' \) and a new task space outcome \( x' \). We are also able to potentially access the full trajectory \( \{(s_t, x_t)\}_{t=0}^T \) in which the agent reached the final position \( (s', x') = (s_T, x_T) \).

**Collecting samples**

We can now use the heuristics 1-3 in order to augment our datasets \( (D_n)_{n=1}^N \). If \( x' \in X_n \) and \( D_n \) is empty, we can conclude that the final sample \( (s', x') \) is consistent with \( D_n \), by heuristic 1. Similarly heuristic 3 tells us that if \( x' \in X_n \) and \( X_n \) is also the region we attempted to reach, while the
reached posture $q'$ was sufficiently close to the posture $\hat{q}$ the agent attempted to reach (by some suitable metric to the implementation), then the final sample $(s', x')$ is also considered consistent with previous samples in $D_n$.

Heuristic 2 can now be useful if we also received a trajectory $\{(s_t, x_t)\}_{t=0}^{T}$. Since we know that the last sample in the trajectory was consistent with $X_n$, then every sample $(s_t, x_t)$ in the trajectory is also consistent, if $x_t \in X_n$ and the subsequent sample $(s_{t+1}, x_{t+1})$ is also consistent. We can draw this conclusion since we could reach the final position $(s', x')$ directly from $(s_t, x_t)$ without leaving $X_n$. This means that an average of $(s', x')$ and $(s_t, x_t)$ is also in $X_n$ (since the intermediary samples in the trajectory are such averages). These trajectory samples are therefore consistent with each other by Eq. (5.26), which allows us to collect much more samples with every iteration whenever a trajectory is provided. If such trajectories are accessible or not depends on the robot problem at hand.

The memory factor

The purpose of collecting samples to a dataset $D_n = \{(s_i, x_i)\}_{i=1}^{k_n}$ is to be able to use some regression method to generate a continuous manifold $\pi_n : X_n \rightarrow S_n$ that fulfils two separate objectives:

1. The manifold provides state estimates for how to reach positions in $x \in X_n$, so that:
   \begin{equation}
   \psi(\pi_n(x)) = x
   \end{equation}

2. The manifold creates a fully connected state space, so that for all $x_i, x_j \in X_n$:
   \begin{equation}
   f(\pi_n(x_i), \pi_n(x_j)) = \pi_n(x_j)
   \end{equation}

We already have the heuristics to make the datasets consistent which increases the chance that $\psi(\pi_n(x)) \in X_n$ for $x \in X_n$. This is not as good as knowing that the goal $x$ was precisely reached, but it is a start for objective 1. For objective 2 however we still have no principle of expecting the resulting manifold to be fully connected.

In Sec. 2.2.3 a possible approach for finding a discrete set of fully connected states was discussed, where the general idea was to freely attempt to move between the samples, and replace the samples we tried to reach with the samples we actually reached. Such a set would only converge when a fully connected set of samples was found. We will use this idea here as well, where the general idea is to let old samples $(s_i, x_i)$ “fade away” in order to give room for newer and better samples. To practically do this we introduce a “memory factor” $m$ to every sample $(s, x)$, so that instead of considering samples $(s, x)$ we consider $(s, x, m)$. Every sample starts with $m = 1$ which represents a perfectly remembered sample. Every time the sample is then used in regression to create an approximation $\hat{s}$, the memory of the sample is a little bit decayed, so that

\begin{equation}
\begin{aligned}
m &\leftarrow \lambda m
\end{aligned}
\end{equation}
This happens every time the agent attempts to reach the region $X_n$ of the dataset, for most types of regression. An important exception is LLR where a subset of the dataset will be selected for regression, with the samples closest to the particular goal $x \in X_n$. Every time an estimate $\hat{s}$ works, the robot ends up in the region $X_n$, which means that new samples are added that are similar to the ones decaying, meaning that working samples are constantly renewed, while samples that lead to approximations that do not allow the agent to reach $X_n$, will decay, without providing new samples to the dataset. By viewing the memory factor as an extra coordinate in the task space, we can do regression to a goal $x$ also based on a goal $m = 1$. This means that the distance $d$ between a sample in the dataset and a goal $x$ becomes:

$$d_i = \left\| \frac{\hat{x}}{1} - \frac{x_i}{m_i} \right\|_2$$

for sample $i$ in $D_n$, and this is the metric that will be used in the regression. This will in turn give less weight to a sample with a memory factor close to 0 when doing regression, even if it is close to a target $x$, since it is far from the position $m = 1$ we want to reach in the memory space.

The overall effect of this is that samples will fade away the more times they have been used, since they get increasingly further from the goal $m = 1$. For some parts of $X_n$ our regression method will create states that can be reliably reached from many different starting positions, and since this region is often reached this leads to many new samples with similar characteristics. Slowly a reliable manifold will start to appear as every part of the manifold that does not allow it to be reached will fade away without renewing itself with new samples. In the end this leads to a manifold that tends to have states that can be reached from as many starting positions as possible (which tends to make it connected), where reaching the states leads to the desired outcomes, fulfilling both Eq. (5.29) and (5.30).

### 5.4.2 Finding sets of GB models

We now turn to the task of finding a complete set $\{M_n\}_{n=1}^N$ of continuous manifold models, in order for the agent to reach any part of the task space. To do so, the set of models needs to fulfill the three objectives of Sec. 5.1: the agent should always be able to recover to one of the models, to move between any two models (directly or through a sequence of intermediary models), and to reach every possible goal $x^* \in X$ using some model $M_n$. Since the models are learned online, these properties must be acquired successively through interaction with the environment.

In the implementation presented here we rely on regions $\{X_n\}_{n=1}^N$ that are handcrafted in advance. When doing so we will demand that the regions cover all of task space:

$$X_1 \cup X_2 \cup \cdots \cup X_N = X$$

and that they are disjoint:

$$X_i \cap X_j = \emptyset$$
The fact that the task spaces \(X_n\) do not overlap means that we get a single mapping from the task space to the state space, defined so that \(s = \pi_n(x)\), where \(n\) is given by the position \(x \in X_n\.

Exactly how to best handcraft a set of regions \(\{X_n\}_{n=1}^N\) is an open question, and we will see later in Sec. 6.3 the effects of different such choices. In particular it is worth pointing out that the choice to have only one region \(X_n = X\) is analogous to the normal GB problem. Note however that finding fully connected sets generally becomes harder the larger the task space is, and might not be possible at all in that case (which is why traditional GB is insufficient).

**Successive build up of models**

As the training of the agent begins we search a manifold, \(\pi_n : X_n \rightarrow S_n\) to each model \(\mathcal{M}_n\). These manifolds also define the state space \(S_n\) of each model, as discussed in earlier sections. To find these manifolds, note that the agent always starts in some state \(s\) and some task space position \(x\). Since \(x \in X_n\) for some \(n\), and the dataset \(D_n\) is previously empty, this sample can immediately be added to \(D_n\), allowing us to initiate the dataset:

\[
(5.35) \quad D_n = \{(s,x)\}
\]

Since we have a dataset for current region \(X_n\), we are also able to generate a manifold:

\[
(5.36) \quad \pi_n(x) = \text{RegressionMethod}(x,D_n)
\]

For this reason there will always exist a manifold \(\pi_n\) for the region \(X_n\) where the agent currently resides. To move to another region \(X_{\hat{n}}\) (given by a goal node \(\hat{n}\) from the MDP planner) we must now also choose a goal:

\[
(5.37) \quad \hat{x} \in X_{\hat{n}}
\]

within that region. Typically \(\hat{x}\) will be selected uniformly randomly in the region, so that:

\[
(5.38) \quad \hat{x} \sim U(X_{\hat{n}})
\]

If \(D_{\hat{n}} = \emptyset\), we can simply use the dataset \(D_n\) of our current region and extrapolate to \(\hat{x}\). If we are successful in reaching our goal region \(X_{\hat{n}}\) then we are also in \(S_{\hat{n}}\) since \(D_{\hat{n}}\) was previously empty. If we fail to reach \(X_{\hat{n}}\) however, a couple of things can happen. We might not leave our initial region, or we might end up in another region \(X_{n'}\). If this other region has no previous samples we can add the new position \((s,x)\) to it, since our position is always consistent with an empty set. If we however ended up in a region where a manifold is already defined, then the agent would need to recover somehow. Exactly how to do this is, as stated before, up to the particular implementation, but typically the agent would attempt to return to the previous region, or alternatively attempt to reach the manifold of the region it ended up in, by mistake. The agent is assumed to have recovered as soon as it is in a state that is consistent with the manifold of any model, by the criteria of consistency given in previous section.
Handling model set insufficiencies

During this training our set of models $\{\mathcal{M}_n\}_{n=1}^N$ can fail in two ways.

1. The agent might be unable to recover to any model $\mathcal{M}_n$.

2. A model might become unreachable. This can happen if the surrounding models change in a way so that an unchanged model cannot be reached from them anymore. It can also happen if by accident a sample is accepted that is inconsistent with previous samples.

The simplest way to deal with such problems (but probably far from optimal) is to simply replace a model with a new, empty one. In the case where a recovery cannot be made, this means that the model of the agents current position is replaced, so that the agent is once again in a consistent position. In the case where a region cannot be reached, removing that model allows extrapolation from the surrounding regions in order to find a new manifold. Whenever doing such a replacement it is important to update the MDP model, including all probabilities $P(n, \hat{n})$ and potential intrinsic rewards $R(n, \hat{n})$. Since this means that transitional probabilities are once again completely unknown for the region $X_n$, they have to be rediscovered.

If the model develops as intended, after a while every region will have a manifold that is at least reachable from some of the neighbouring regions. If trajectory samples $\{(s_t, x_t)\}_{t=0}^T$ can be collected during each iteration this means that regions that can be transitioned between will receive very similar samples at their borders, as the agent transitions over it. This will in turn allow the agent to smoothly follow trajectories $\{x^*_1, x^*_2, \ldots\}$ not only within individual regions $X_n$, but also between those regions that are connected in such a way.

5.4.3 Algorithm

Given a method to learn each GB model $\mathcal{M}_n$ online, as well as how to make the whole set of models $\{\mathcal{M}_n\}_{n=1}^N$ adhere to the objectives in the process, we can now summarise the complete approach in Alg. 6. As in previous section we do not discuss here how the planner will make its decisions, we simply assume that the planner collect the information it needs from the general process described here.
Algorithm 6 Online learning - Continuous manifold approach

1: Segment the full task space $X$ into a disjoint set of regions $\{X_n\}_{n=1}^{N}$
2: 
3: Observe current positions $(s,x)$
4: Identify current node $n$, for which $x \in X_n$
5: Add $(s,x,1)$ to dataset $D_n$, where 1 is the initialisation of the memory factor.
6: 
7: while running do
8: 
9: Observe goal $x^{*}$ (Could be None, depending on how the MDP model is trained)
10: Observe goal node $n^{*}$ so that $x^{*} \in X_{n^{*}}$ (Could subsequently also be None)
11: Get goal node $\hat{n}$ from MDP planner.
12: 
13: if $x^{*} \in X_{\hat{n}}$, i.e. if $\hat{n} = n^{*}$ then
14: Set $\hat{x} = x^{*}$ (Alternatively only do this if $n = \hat{n} = n^{*}$)
15: else
16: Draw a random position $\hat{x}$ in $X_{\hat{n}}$.
17: 
18: if $D_{\hat{n}}$ is empty then
19: Select goal state $\hat{s} \leftarrow \pi_{\hat{n}}(\hat{x})$ using the dataset $D_{\hat{n}}$ of current region $X_{\hat{n}}$.
20: else
21: Select goal state $\hat{s} \leftarrow \pi_{\hat{n}}(\hat{x})$ using $D_{\hat{n}}$ and some regression method.
22: Decay the memory factor of each sample used in $D_{\hat{n}}$, by a factor $\lambda$.
23: Add some exploratory noise to $\hat{s}$ based on number of samples in $D_{\hat{n}}$.
24: Send $\hat{s}$ to robot controller.
25: Observe resulting trajectory $\{(s_t,x_t)\}_{t=1}^{T}$ ($\tau = 1$ means only final positions are observed)
26: 
27: Add $\{(s_t,x_t,1)\}_{t=t_0}^{T}$ to $D_n$, where $x_t \in X_n$ for all $t < t_0$ (1 is the memory factor)
28: while $(s_t,x_t)$ is not consistent with any model $M_{n'}$ do
29: Choose a node $\hat{n}$ to try to recover to
30: Choose a random goal $\hat{x} \in X_{\hat{n}}$
31: Generate $\hat{s} \leftarrow \pi_{\hat{n}}(\hat{x})$ and add some random exploratory noise (as before)
32: Send $\hat{s}$ to robot controller.
33: Observe new resulting trajectory $\{(s_t,x_t)\}_{t=1}^{T}$
34: 
35: if still inconsistent after a predetermined number of attempts then
36: Observe current region $X_{n'}$ so that $x_t \in X_{n'}$
37: Reinitialise GB model $M_{n'}$ so that $D_{n'} = \emptyset$.
38: Note that this makes position $(s_t,x_t)$ consistent with $M_{n'}$
39: 
40: Add $\{(s_t,x_t,1)\}_{t=t_1}^{T}$ to $D_{n'}$, where $x_t \in X_{n'}$ for all $t \geq t_1$.
41: 
42: Observe current positions $(s,x) \leftarrow (s_t,x_t)$
43: Observe current node $n \leftarrow n'$
To evaluate the capabilities of the method laid out in previous chapter, we implement it here in a simulated robotic environment. The system we consider is a planar arm with many degrees of freedom (DoF), acting in an environment with walls that it cannot traverse. It is similarly not allowed to pass through itself, or move faster than some speed limit, see Fig 6.1 and 6.2. The goal is to learn to move the end-effector of the arm to any given position $x^*$ in its environment, which means that the arm does not only need to learn a high dimensional posture configuration that reaches that goal, but must also learn how to transition to that posture without breaking any of the rules of the environment. This provides a good starting point for evaluating the proposed method, since we are dealing with a system with high dimensional continuous state and action spaces, where the outcome of an action depends on the state it is made in. Similar problems have been investigated previously in [Hayashi, 1991; Hayashi and Kuipers, 1992] with handcrafted policies, and later with online learning using Goal Babbling (GB) [Baranes and Oudeyer, 2013; Benureau, 2015; Rolf et al., 2011] but in environments without obstacles so that no planning was necessary. We will attempt to solve the problem in two ways. The first is based on the home state approach of Sec. 5.3, where a set of home states is found iteratively that the robot learns to move between. The second is based on the continuous manifold approach of Sec. 5.4, where we search continuous manifolds covering the full task space that the arm can move smoothly along.
Figure 6.1: The state $s$ of the arm is defined by the angles $\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ so that $s = [\alpha_1 \ \alpha_2 \ \ldots \ \alpha_k]^T$ were $k$ is the number of DoF of the arm, which is freely specifiable. Every angle is allowed any value $\alpha_i \in [-\pi, \pi]$, and the segments between the joints are all of equal length, so that the total length of the arm is 1 unit. The width of the arm is $0.01$ units. For task space position we consider the Cartesian position $x = [e_1 \ e_2]^T$ of the tip of the arm relative to some reference point, which is also referred to as its end-effector position. The arm is placed with its base in the middle of a square room with the dimensions as specified above. The size of the base of the arm is exaggerated in the image for clarity, but is in reality infinitely small.

6.1 Experimental design

6.1.1 Physical description

The system we look at consists of a planar arm made of a freely specifiable number $k$ of uniform segments, adapted so that the total length of the arm is 1 unit. The segments are, in turn, connected by joints with angles $\alpha_i \in [-\pi, \pi]$, where the first joint is directly connected to a fixed base. We consider the posture of the arm as the state, which provides a complete description of the system (as discussed in Sec. 2.2.3), so that $s = [\alpha_1 \ \alpha_2 \ \ldots \ \alpha_k]^T$. For task space, we consider the end-effector position $x = [e_1 \ e_2]^T$ of the arm, which corresponds to the Cartesian coordinates of the end of the last segment of the arm. The arm is placed with its base in the middle of a square room with sides 1 unit, with additional walls of length 0.2 reaching inwards from the centre of each wall, see Fig. 6.1.

6.1.2 Spaces

The state space $S \subseteq \mathbb{R}^k$ of the robot arm corresponds to the posture space of the arm, where each dimension corresponds to the values of a joint $\alpha_i \in [-\pi, \pi]$. The robot is controlled by goal states $\hat{s}$, which consists of target angles $\hat{\alpha}_i$ that describe the state we want the robot to change into. Since this action is also a state we have an action space $A = S$. The task space $X \subseteq \mathbb{R}^2$ is the 2 dimensional surface of the room of Fig. 6.1. This limits each element $e_i$ of the end-effector position $x = [e_1 \ e_2]^T$ to a span $0 \leq e_i < 1$. 70
Figure 6.2: The dynamics of the system. An agent in a state $s$ is given a goal state $\hat{s}$, which the agent will start moving towards until reaching the goal state, or until breaking one of the rules of the environment. These rules are that the arm is not allowed to pass through walls or itself, or move any joint position more than $\frac{1}{100}$ units between two steps in the trajectory. This last rule limits the arch length $l_k$ any joint $k$ can trace in a single trajectory to 1 unit. If either rule would be broken in the next step of a trajectory, the arm is stopped.

6.1.3 Dynamics

The robot arm is as previously mentioned controlled using goal states $\hat{s}$. Given such a goal the robot controller will attempt a linear trajectory in 100 steps, so that:

$$s_t = (\frac{100 - t}{100}) s + (\frac{t}{100}) \hat{s}$$

at the $t$:th step. This motion proceeds until the 100th step is reached, or until the next state $s_{t+1}$ would break one of three restrictions of the environment:

1. The agent is not allowed to intersect any of the walls.
2. The agent is not allowed to intersect itself.
3. No joint is allowed to move its position more than $\frac{1}{100}$ units between two steps in the trajectory.

These restrictions are illustrated in Fig. 6.2. The state the agent finally ends up in is denoted $s'$, where we see that $s' = \hat{s}$ if neither rule is violated. We thus have a forward dynamics state description $f : S \times S \rightarrow S$ so that:

$$f(s, \hat{s}) = s'$$

For the task space, note that the end-position $x$ is uniquely defined by the state of the arm. There is thus a function $\psi : S \rightarrow X$ so that:

$$x = \psi(s)$$
Figure 6.3: How the starting state $s'$ is found. The arm starts coiled up and then attempts to straighten every joint angle out, until hitting a wall.

Although it is possible to find this relationship explicitly knowing the structure of the robot, we do not assume this knowledge. What we do know is that the end-effector position $x$ is defined by the state $s$, so that if we can find a fully connected state set $S_n$ (states that can always be reached from each other), then each such state $s \in S_n$ also corresponds to a unique task outcome $x = \psi(s)$, in some task space $X_n = \psi(S_n)$. This will be important to continuous manifold approaches, see Sec. 5.4.1.

6.1.4 Procedure

The objective is for the robot to learn to move the end-effector to any goal $x^* \in X$ within a finite number of iterations, using the observations it generates by attempting to reach goal states $\hat{s}$ from different states $s$, as described in Sec. 4.1.1. This must be learned online, where the current state of the robot is always the state the previous action left it in. In order to find a physically plausible starting configuration (as most random choices of joint angles would lead to self collisions) the arm starts coiled up and then uncoils until hitting a wall, which constitutes its starting position. This procedure is illustrated in Fig. 6.3.

6.2 Approach 1

Our first approach combines the home state approach described in Sec. 5.3 for the GB models, with the learn-by-doing approach for the Markov Decision Process (MDP) model, as described in Sec. 5.2. The main idea here is to provide a simple and general implementation for tackling the control problem. We will thus favour algorithmic simplicity over performance in this case. For this reason we will use a very simple recovery routine, and simply let an evaluation terminate if this routine fails. This will, in addition, give us some feedback of how prevalent recovery failures are, which will help us design methods for how to handle such situations.

Another simplification is that we will ignore the trajectories between iterations, and only use the final positions $s'$ and $x'$ for training. The ability to access the trajectory data is specific to the case of the planar arm, and we are interested in seeing the learning capabilities in the more general cases where data is sparse and limited to one sample per iteration.

When evaluating the performance, the effects of different radii $r$ of reach out regions $X_n$ will be investigated, as well as the impact of varying numbers of DoF of the arm. The full code can be found at: github.com/Loviken/MCGB
6.2.1 MDP model

For MDP model we consider probabilities $P(n, \hat{n})$ which approximate the probability to move from a home state $s_n$ to $s_{\hat{n}}$, i.e., so that $f(s_n, s_{\hat{n}}) = s_{\hat{n}}$. Whenever a new GB model is created using a home state $s_n$, we initiate the probabilities to move to or from that node, so that:

\begin{align}
  P(n, \hat{n}) &= 1 \\
  P(\hat{n}, n) &= 1
\end{align}

if $s_{\hat{n}}$ was the last home state the agent visited before discovering $s_n$, or if $s_{\hat{n}}$ is one of the 2 nearest neighbours to $s_n$ in the task space, e.g., with respect to the distance:

\begin{align}
  ||\psi(s_{\hat{n}}) - \psi(s_n)||
\end{align}

In all other cases we initiate:

\begin{align}
  P(n, \hat{n}) &= 0 \\
  P(\hat{n}, n) &= 0
\end{align}

Each new node $n$ will thus initially be connected to 2 or 3 other nodes, depending on if the new node was created from one of its closest neighbours or not. Note that new nodes will later be able to make additional edges to these previous nodes. The purpose of only allowing a few transitions is to keep the size of the MDP down, since considering all possible transitions between all nodes would create an explosion of combinations as the number of nodes would increase. It would thus soon become impossible to try out all possible transitions in order to approximate their probability of success. The addition of an edge to the node from which a new node was found assures that all nodes are connected to each other in some way. The main problem here might be that edges are not symmetrical, so that an agent can move directly between two home states in one direction, but not the other. Imagine a robot falling over for example. That robot is unlikely to be able to move directly back to the standing state. For a planar arm however, the dynamics should be more or less symmetrical.

Given these initial approximations for probabilities, we update every probability $P(n, \hat{n})$ every time a transition $n \rightarrow \hat{n}$ is attempted, so that:

\begin{align}
  P(n, \hat{n}) &= \alpha P(n, \hat{n}) + (1 - \alpha) \text{I}(s' \in S_{\hat{n}})
\end{align}

where $\alpha = 0.5$, and $\text{I}(s' \in S_{\hat{n}})$ is 1 if $\hat{n}$ was reached and 0 otherwise. For $\hat{n}$ to be reached the resulting state $s'$ needs to be in $S_{\hat{n}}$, which we assume to be true if $||s' - s_{\hat{n}}|| < 0.01$.

For the reward, we define a function:

\begin{align}
  R_{x^*}(n) &= \begin{cases} 
    1, & x^* \in X_n \\
    0.01 \exp\left(-\frac{||x^* - \psi(s_n)||}{r}\right), & \text{otherwise}
  \end{cases}
\end{align}

where $||x^* - \psi(s_n)|| < r$ means that $x^* \in X_n$, so that $r$ is the radius of the regions $X_n$. This will help the agent plan a transition to a home state close to the target $x^*$ if no region $X_n$ includes
the goal, while at the same time being small enough to not compete with any region $X_n$ that does include the goal $x^*$. Using these rewards and probabilities an action-value function $Q_{x^*}(n, \hat{n})$ can be found using value iteration, as in Alg. 4. Goal nodes are then chosen greedily, so that:

$$\Pi_{x^*}(n) = \operatorname{argmax}_{\hat{n}} Q_{x^*}(n, \hat{n})$$

6.2.2 GB models

For the GB models $\mathcal{M}_n = \{D_n, \pi_n\}$, we use the home state approach, described in Sec. 5.3. Each GB domain $D_n$ thus represents a state $s_n$ with an associated end-effector position $x_n = \psi(s_n)$ in the task space. We define $X_n$ to be a disk of radius $r$ around this point $x_n$. New GB models are created whenever the agent reaches an end-effector position $x$ that is outside of all previous domains $\{X_n\}_{n=1}^N$. In order to travel between different GB models, the agent simply uses the home state $s_n$ of that model as a goal state $\hat{s}$. For state recovery (whenever the agent is not in any home state) the agent will attempt to move back to the last home state it visited. Since transitions are symmetric in the planar arm case, this should theoretically always be possible. However since every transition trajectory is simulated in 100 steps, unless stopped by breaking a physical law, see Fig. 6.2, the return trajectory might be simulated in a different number of steps which could lead to slightly different properties of the return trajectory. In practice this could make the arm unable to return. If this happens the agent will try to change its current position $s$ slightly by adding a small noise to give a goal state:

$$\hat{s} \leftarrow s + 0.01 \pi U(0, 1) \mathcal{N}(O, I_k)$$

where $\pi (= 3.14 \ldots)$ is the scalar constant, $U(0, 1)$ is a uniformly random scalar in the span $[0, 1]$, and $\mathcal{N}(O, I_k)$ is a vector where each element in $s$ is given a normal distributed value around 0. The motivation for the two terms is to create variation in each DoF, while also varying the distance $||s - \hat{s}||$. After this re-positioning the agent attempts to move back to the home state again, and if this does not work it re-positions again and tries to return. If the agent is still unable to recover after 100 such attempts the whole evaluation is terminated. Later we will solve such situations by adding or moving existing GB models, but for this evaluation we do not want to analyse the effects of such additions or changes. Terminating the runs also gives a better understanding of how prominent the problem is.

For the reach out policy $\pi_n$, each GB model holds a dataset $D_n = \{(s_i, x_i)\}_{i=1}^{k_n}$ of observed outcomes when doing reach-outs from $s_n$. We thus know that:

$$\xi(s_n, s_i) = x_i$$

for any pair $(s_i, x_i) \in D_n$. Similarly we can also add the result of any action $\hat{s}$ taken from $s_n$, so that if we observe $\xi(s_n, \hat{s}) = x'$ we can simply add $(\hat{s}, x')$ to $D_n$. When creating this dataset we
do not really care about the region $X_n$, and we will add any observation, regardless of where $x'$ ends up. In a way the regions $X_n$ are more to help the MDP planner plan transitions to a reasonable home state, and from that home state the reach out policy will attempt to reach the goal regardless of if it happens to be in the corresponding region or not. This gives a natural way for discovering new home states, as we can use samples $(s_i, x_i)$ near the edge of the region $X_n$ of one home state, in order to reach task space positions outside the domain of any previous GB model.

Once $\Pi_{x^*}(n) = n$ we know that the agent is in an optimal home state for a reach out, as no further transitions can increase the estimated reward for the MDP. When this happens the arm will attempt a reach-out to $x^*$. This is done by selecting the state $s_{\text{min}} = s_i$ of the sample $(s_i, x_i)$ that minimises the distance $d = \|x_i - x^*\|$. We then add some exploratory noise finally getting:

\[
\pi_n(x^*) = s_{\text{min}} + \min(d, 0.1) \cdot \pi U(0, 1) N(O, I_k)
\]

with a random term $U(0, 1) N(O, I_k)$ computed as in Eq. 6.10, with cut offs if any joint angle should fall outside of the span $[-\pi, \pi]$. The term $\min(d, 0.1)$ means that the noise level increases up until a distance $d = 0.1$ between the goal $x^*$ and the closest sample $x_i$, to stay constant after that. The other values should be seen as an arbitrary constant. During our testing we found the method to be relatively robust to the precise choice of these meta-parameters. A more complete statistical evaluation could however be performed to precisely analyse their impact.

### 6.2.3 Evaluation

Given the specified models above we get the solution seen in Fig. 6.4. Alg. 5 describes how this can be done, where a new goal $x^*$ is chosen uniformly randomly in the full task space $X$ every 20th iteration. For each goal $x^*$ we define a reaching error $\epsilon$ to be the shortest distance between the end-effector and the goal throughout these 20 iterations. This is done since the agent could attempt a reach-out during any of these iterations, and in some cases it would not even reach a home state from which to do a reach-out.

When evaluating the model we are interested in the effect of different radii of the regions $\{X_n\}_{n=1}^N$, and the effect of different number of DoF of the arm. The effect of the number of DoF is of particular interest since it ties to the core challenges of dimensionality, in Sec. 2.3.2. Most machine learning methods, such as Reinforcement Learning (RL), scale badly with such an increase in motor and state space dimensionality. The GB approach has been shown to be able to handle large action spaces, but without the ability to plan trajectories between different states. Testing different numbers of DoF is thus a good way to test this ability. Learning with many DoF could potentially be even more efficient, as that provides more ways to reach any given goal.

For the choice of radius $r$, we see that if the radius is large enough to encompass the whole room the method turns into classical home state GB, where the arm constantly returns to the same home state after every action. This comparison gives us some measure of the utility of
The home state approach summarized. Each domain $D_n$ is defined by a home state $s_n$ and a region $X_n$ which is a disk around the home state end-effector position $\psi(s_n)$ with a radius $r$. Given a goal $x^*$ the MDP model plans transitions to a home state from which $x^*$ can be reached, in this case $s_5$. From that home state a reach-out is made using GB model $\pi_5(x^*)$.

Multiple home states compared to one. When $r$ becomes very small on the other hand, planning will become more complicated as the amount of GB models grows as well. This means that the planner might be unable to move the agent to a relevant home state $s_n$ for a reach out within the time allowed, since we only consider transitions between neighbouring nodes in order to keep the size of the MDP model down.

To evaluate the performance we look at radii:

$$r = \{ 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.5, 2.0 \}$$

and DoF:

$$DoF = \{ 5, 10, 30, 50, 100, 200, 1000 \}$$

When comparing different radii, we use a 100 DoF arm, and when comparing different DoF we use radius 0.2. Each setting is evaluated over 100 independent runs, which goes on for 10,000 iterations or until the agent is unable to recover to a home state.

6.2.4 Results

When running the arm first uncoils into its starting position (Fig. 6.3). Once there, a home state $s_1$ is created with an associated domain $X_1$ around the end-effector position $x_1 = \psi(s_1)$, since no
Figure 6.5: The reaching error over time for arms with different number of DoF, with \( r = 0.2 \), over 100 independent runs. The mean error is shown in red, with each sample as black dots. All means are geometric and taken over these independent runs, and then smoothed out using a window of 20 time steps to reduce noise further. Each such time step is 20 iterations, which is the time to reach one goal \( x^* \). In “Error vs. DoF” each line shows mean reaching error collected up until the indicated iteration, and 1,000 iterations back (so that the curve 5,000 is the average error computed between iterations 4,000 – 5,000, etc). Finally in “Evaluations running” we see to what extent evaluations are terminated due to an inability to return to a home state.

previous home state covers that position. After that it starts to attempt reach-outs to the goal \( x^* \), since it is already in the best (the only) home state. When doing so it starts by deviating slightly from the home state, since that is its only sample so far. After each deviation it returns to the home state to try again. If a deviation gets closer to the goal \( x^* \) than \( x_1 \), the next attempt will be a deviation of that deviation. Unless the region radius \( r \) includes the whole task space, at some point the arm deviates enough from \( x_1 \) to reach a position \( x' \not\in X_1 \), i.e. \( ||x' - x_1|| > r \). At this point a new home state is created from that state, with a region \( X_2 \) centred around \( x' \). Now the planner starts to play a more important role by planning transitions between the home states in order to always reach out from the home state closest to the goal. Whenever a new home state is discovered, the planner assumes a connection between the new home state and the home state from which it was discovered, as well as to the 2 home states that are the closest in the task space (which could include the one it was discovered from). Whenever a transition is attempted and fails the approximated probability to do that transition is sharply reduced (see Eq. (6.7)) which encourages the planner to attempt another path if such exists.

Moving on to the collected data, observe that all averages are geometrical, which averages over the order of magnitude, rather than the average of absolute value as in the more common case of
**arithmetic** mean. Geometrical mean of a set \( A = \{a_1, a_2, \ldots, a_n\} \) can thus be described as follows:

\[
\text{mean}_\text{geo}(A) := \exp \left( \frac{1}{n} \sum_{i=1}^{n} \ln a_i \right)
\]

Fig. 6.5 show the impact of different number of DoF for a fixed radius of 0.2, measured over 100 independent runs for each setting. The number of DoF does not seem to impair the ability to reach each random goal \( x^* \) within 20 iterations (this is referred to as the “reaching error”). More DoF rather seem to improve this ability, up until around 1,000 DoF where the reaching error increases again. A possible reason for this is that the exploration rate, Eq. 6.12, was tuned using a 100 DoF arm. It is possible that optimal noise levels depends on the number of joints affected. Overall the arm manages to reach every goal with an average precision of less than \( 10^{-2} \) units for almost every setting. This means that if the room was 1 × 1 meters, it would reach any goal with an error of less than 1 cm on average.

Occasionally the arm gets stuck and unable to recover, usually as the consequence of barely avoiding stroking a wall before hitting it and being stopped. Since the trajectory was terminated before doing the 100 planned steps of its reach-out trajectory, the recovery trajectory attempts smaller steps as it attempt to return in 100 steps. The trajectory back is thus not identical to the reach-out trajectory, and could sometimes hit the wall it previously just missed, on the way back. If unlucky when re-positioning and attempting to return again it might get stuck indefinitely. This would lead to that particular evaluation being terminated. It seems like an arm with fewer DoF is more prone to get stuck this way, which is likely due to the decreased flexibility of such an arm.

When investigating the impact of different radii \( r \) for an arm with 100 DoF, there seems to be a sweet spot for a radius around 0.2 – 0.3, in terms of the reaching error, as seen Fig. 6.6. The difference in reaching error performance is however not striking and are at the end bellow \( 10^{-2} \). When visually observing the different settings we however observe that larger radii manage to cover the task space faster, but at the price of including positions in a region \( X_n \) that cannot be reached directly from the home state \( s_h \), typically by including positions on the other side of a wall relative to the home state. This leads to a set of goals that cannot be reached with higher precision than their distance to the wall, and this distance increases with the radius \( r \) of the regions \( X_n \). We can especially see this split in the distribution for \( r = 2.0 \). The reason why reaching error does not decrease more is that most goals can still be reached with high precision. Another issue of large radius seems to increase the risk of getting stuck. This is probably due to the fact that a large radius means longer reach-outs and recoveries, which more opportunities of getting stuck along the way.

The problem with smaller radii seems to be that they lead to more home states, as can be seen in Fig. 6.7. Since transitions are only assumed between neighboring home states and the state from which it was discovered, this leads to more transitions to any goal. Since only 20 iterations are allowed for any goal \( x^* \) it means that some goals are too far away for the arm to have a
Figure 6.6: Variation of radius, for 100 DoF, over 100 independent runs. The mean error is shown in red, with each sample as black dots. Note the split in distribution for \( r = 2.0 \). All means are geometric and taken over these independent runs, and then smoothed out using a window of 20 time steps. Each such time step is 20 iterations, corresponding to the attempt to reach one goal \( x^* \). In “Error vs. radius” each line shows mean reaching error collected up until the indicated iteration, and 1,000 iterations back (so that the curve 5,000 is the average error computed between iterations 4,000 – 5,000, etc). Finally in “Evaluations running” we see to what extent evaluations are terminated due to an inability to return to a home state.

...chance to get close to it. A way around this would be to allow transitions between home states that are far from each other. The problem then is how to learn between what home states such transitions are possible, since 200 home states (as for \( r = 0.05 \)) results in around \( 200^2 = 40,000 \) possible transitions to try out.

Instead of only considering the global average of the reaching error we can also see how the error is distributed over the task space, as seen in Fig. 6.8. In this case we clearly see the problem of traditional home state GB. Since there is only a single home state (which happens with \( r = 2.0 \)), this leads to areas that cannot be reached in a single motion from that home state. This is the explanation for the split in error distribution for \( r = 2.0 \) in Fig. 6.6. Similar splits seem to happen for smaller radii as-well (but to a lesser degree), since every disk \( X_n \) that overlaps a wall leads to attempted reach-outs that cannot get closer to the goal than the wall allows it to.

### 6.3 Approach 2

For the second approach, we want to evaluate the continuous manifold approach. Ideally this would allow the arm to follow trajectories smoothly in the task space. To do this we create GB models using the continuous manifold approach described in Sec. 5.4. For training of the resulting...
Figure 6.7: The average number of home states created over time for an arm with 100 DoF for different radii over 100 independent evaluations. Blue lines encompasses 1 standard deviation.

Figure 6.8: Average over 100 independent runs. Each pixel shows the geometrical mean of the reaching error for the 10 closest goals $x^*$ to the position of that pixel, collected between the indicated iteration and 1,000 before it. Iteration 5,000 thus shows errors collected between iterations 4,000 – 5,000. “GB” here refers to $r = 2.0$, since such a radius means that all of the task space is within the region $X_n$ of a single home state, so that no new home states are created.
Figure 6.9: The continuous manifold approach summarised. Every model $M_n$ is defined by a region $X_n$ and a mapping $\pi_n : X_n \rightarrow S_n$ that maps every task space position $x \in X_n$ to a specific state for which the end-effector is ideally $x$. Given a goal $x^*$ the MDP model plans transitions between the regions $\{X_n\}_{n=1}^N$ by reaching for random goals $\hat{x} \in X_n$ in each, until reaching for the region of the goal $x^*$ (in the example of the image $X_{12}$). In this case the agent reaches for the final goal $x^*$ using goal posture $\pi_{12}(x^*)$.

MDP model we will compare the different approaches mentioned in Sec. 5.2, which includes selecting goal nodes $\hat{n}$ randomly, to select $\hat{n}$ in order to reach random goals $x^*$ (learn by doing), and to select actions by what improves the model the most (intrinsic motivation). In this case the main goal is to see if such continuous manifolds can be created while allowing the agent to reach any goal even if it would mean planning a trajectory around walls. In this case the agent will be given every data point in each trajectory $((s_t, x_t))_{t=1}^{100}$ in order to provide more data for the manifolds. This means that reaching errors here are not completely comparable to those in Approach 1. The main goal is not to find the lowest reaching error however, but to see if the robot problem can also be solved in a way that allows it to smoothly follow goal trajectories. Finally, since the continuous manifold approach is used for the GB models, we segment the task space in advance into a set of uniform square regions $\{X_n\}_{n=1}^N$, where $d \times d$ is the resolution of the segmentation. The planner is only allowed to attempt transitions between adjacent regions (Moore neighbours). This leads to an overall model as shown in Fig. 6.9. The full code can be found at: github.com/Loviken/FEGB/tree/icdl-epirob-2017
6.3.1 MDP model

In this case we do not need to consider the addition of new regions, as they are all defined in advance. We thus initiate probabilities:

\[ P(n, \hat{n}) \left\{ \begin{array}{ll} 1, & X_n \text{ adjacent to } X_{\hat{n}} \\ 0, & \text{otherwise} \end{array} \right. \]  

In this case each region \( S_n \) corresponds to a continuous manifold \( \pi_n : X_n \rightarrow S_n \). The probability \( P(n, \hat{n}) \) should then ideally converge to the average probability to move from a state \( s = \pi_n(x) \) to a state \( \hat{s} = \pi_{\hat{n}}(\hat{x}) \), if \( x \) and \( \hat{x} \) are uniformly sampled from \( X_n \) and \( X_{\hat{n}} \). Once the MDP model is initiated using Eq. (6.16), it is updated so that:

\[ P(n, \hat{n}) \leftarrow \alpha P(n, \hat{n}) + (1 - \alpha) I(s' \in S_{\hat{n}}) \]  

This is identical to Approach 1, with the exception that \( \alpha = 0.8 \) here. This makes model changes slower, which is advantageous in this case as the probability should average over multiple random states in \( S_n \) and \( S_{\hat{n}} \) (compared to the home state approach where every transition was between the same states).

When reaching for a goal \( x^* \) we set the reward:

\[ R_{x^*}(n) \left\{ \begin{array}{ll} 1, & x^* \in X_n \\ 0, & \text{otherwise} \end{array} \right. \]  

If instead the agent is using intrinsic motivation exploration we initiate an intrinsic reward:

\[ R(n, \hat{n}) \left\{ \begin{array}{ll} 1, & X_n \text{ adjacent to } X_{\hat{n}} \\ 0, & \text{otherwise} \end{array} \right. \]  

This reward is then updated every time the transition \( n \rightarrow \hat{n} \) is successfully performed, so that:

\[ R(n, \hat{n}) \left\{ \begin{array}{ll} \beta R(n, \hat{n}), & s' \in S_{\hat{n}} \\ R(n, \hat{n}), & \text{otherwise} \end{array} \right. \]  

with \( \beta = 0.68 \). This value was chosen so that if the estimated probability of success \( P(n, \hat{n}) \) is lower than 0.3 and the transition succeeds, the estimated return \( P(n, \hat{n})R(n, \hat{n}) \) should increase, i.e. the increase in success-probability compensates for the decline in reward.

Given \( R_{x^*}(n) \) or \( R(n, \hat{n}) \) the planning policy \( \Pi(n, x^*) \) can be defined as before by finding an action value function \( Q_{x^*}(n, \hat{n}) \), using value iteration (Alg. 4). For this value iteration the future reward discount is set to \( \gamma = 0.95 \). Given an action value function we define:

\[ \Pi(n, x^*) \left\{ \begin{array}{ll} \arg\max_{\hat{n}} Q_{x^*}(n, \hat{n}) \end{array} \right. \]  

If, instead, a random exploration strategy is used, \( \Pi(n, x^*) \) selects one of the neighbouring nodes uniformly randomly.
Should a node \( n \) become unreachable for some reason during exploration, this is solved by re-initiating the GB model of that node. How this is done can be found in the following GB model section, but it also has implications for the planning since the probability to move between nodes is based on underlying GB models. The initiative to reinitialise a model is made when a probability to reach a node \( n \) from all other nodes is lower than 0.1. For the planner, this means that any intrinsic rewards \( R(n, \hat{n}) \) and probabilities \( P(n, \hat{n}) \) associated to the region are reset to their initial values.

### 6.3.2 GB models

For GB models \( \{M_n\}_{n=1}^{d \times d} \) we consider the continuous manifold approach described in Sec. 5.4. This means that for every region \( X_n \), we search a manifold \( \pi_n : X_n \rightarrow S_n \) that both defines the state space \( S_n \), and provides a model for reaching each goal \( x \in X_n \). As a more detailed description can be found in previous Sec. 5.4, the focus here is rather to provide details of this specific implementation.

In this implementation each manifold \( \pi_n : X_n \rightarrow S_n \) is approximated using the samples \( D_n \) observed in the region \( X_n \), including the memory factors as previously described. To generalise over this data and approximate a manifold, Local Linear Regression (LLR) is used so that:

\[
\pi_n(x) \leftarrow \text{LLR}(x, D_n)
\]

where LLR selects the up to 100 samples closest to position \( x \) in task space, and 1 in memory space, on which regular Linear Regression (LR) is then used. In this final regression each sample is weighted by the distance to \( [x^\top 1]^\top \) in combined task-memory space. After a sample \( (s_i, x_i, m_i) \) in \( D_n \) has been used this way, it is decayed by a small amount \( \sim 0.001 \) so that:

\[
m_i \leftarrow (1 - 0.001) m_i
\]

to make the sample drift away slightly from any goal \( [x^\top 1]^\top \), to make newer samples preferable. This decay level was found empirically and would probably need to be re-tuned for another agent and environment. The full reasoning behind the memory-factor practice is described in Sec. 5.4. Once this regression estimate is made and unless the learned model is evaluated, an additional random noise is added for some exploration, when creating the final goal state \( \hat{s} \). This additional noise is made so that:

\[
\hat{s} \leftarrow \pi_{\hat{n}}(x) + U(0, 1) h\left(P(n, \hat{n}), k_{\hat{n}}\right)
\]

for an attempted transition from \( n \) to \( \hat{n} \). \( U(0, 1) \) is a uniform random scalar in the span \([0,1]\), and \( h(P(n, \hat{n}), d_{\hat{n}}) \) is a normally distributed noise vector that is proportional to the probability \( P(n, \hat{n}) \) of reaching \( \hat{n} \) from \( n \), and inversely proportional to the number of samples \( k_{\hat{n}} \) found in the goal region \( X_{\hat{n}} \). This allows for greater exploration when first discovering a region, while allowing it to converge over time.
In this implementation “State recovery” is made so that whenever the agent attempts to reach a region \( X_n \) and fails, either by not reaching \( X_n \) or reaching a goal state \( \hat{s} \), it is assumed to be in an inconsistent state. If the agent failed by not leaving its original region \( X_n \), it attempts to move to some goal \( \hat{x} \) in that current region \( X_n \). If it did reach another region \( X_n' \), two things can happen. If no previous samples have been observed in \( X_n' \) the dataset \( X_n' \) of the region can be initialised using every sample in the trajectory in \( X_n' \) as they are assumed consistent with each other. If a model is already defined there the MDP model will act as if the node \( n' \) was reached, while remembering that the agent is in inconsistent states. This means that the observed data will not be used to train either the MDP nor the GB models, since the current state of the agent does not represent the dynamics a consistent state would. A probability \( P(n, \hat{n}) \) should only represent the probability of moving between \( S_n \) and \( S_{\hat{n}} \), and not just from any state with an end-effector in \( X_n \).

The agent will move around like this until it attempts to reach a GB model and succeeds, or fails 10 times - after which the model of the current region is reinitiated. After this the state is once again consistent with a model \( \pi_n(x) \) and the learning resumes.

### 6.3.3 Evaluation

Given the specified models above we get the solution seen in Fig. 6.9. It is trained using Alg. 6.

When evaluating the different settings we will mainly consider the impact of different exploration strategies for the MDP planner, the effect of different DoF, and the effect of different resolutions \( d \times d \). For different exploration strategies we compare:

\[
\text{(6.25)} \quad \text{Exploration strategies} = \{ \text{random, learn-by-doing, intrinsic motivation} \}
\]

as described in Sec. 5.2. For the learn-by-doing approach, a goal \( x^* \) is chosen uniformly randomly in the task space \( X \) every 20th iteration. For different DoF we compare:

\[
\text{(6.26)} \quad \text{DoF} = \{ 5, 30, 50, 100, 200 \}
\]

where the 1,000 DoF setting (tested in the previous approach), was excluded as the program would run too slow in that case. For resolution we looked at:

\[
\text{(6.27)} \quad \text{res} = \{ (1 \times 1), (4 \times 4), (6 \times 6), (7 \times 7), (10 \times 10) \}
\]

where \( 1 \times 1 \) is the same as the continuous manifold approach for regular GB, while each region in the others could be seen as an independent such problem. Fig. 6.10 shows the different segmentations.

These settings are evaluated using the “reaching error” which, as before, is how close to a random goal \( x^* \) the arm manages to get in a number of iterations. Note that no noise term is used for the goal states \( \hat{s} \) in Eq. (6.24) in this case, as we are not interested in exploration here, but purely evaluation. During evaluation the agent is allowed \( 3d \) iterations to reach every goal, in order to compensate the fact that a high resolution \( d \times d \) would require more steps to reach any
region, which we saw could be a problem, in Approach 1. Aside from the reaching error we will also consider the “inverse error”, which captures how close to any goal \( x^* \) the inverse estimate \( \pi_n(x^*) \) is, without the requirement to reach that state to begin with. We can thus define the inverse error \( \epsilon_{inv} \) so that:

\[
\epsilon_{inv}(x^*) = ||x^* - \psi(\pi_n(x^*))||
\]

where \( n \) is defined so that \( x^* \in X_n \). In the previous approach this measurement would not make as much sense, since in that case we had an overlap of different regions \( X_n \), and some positions might not be in any region \( X_n \) at all. In this case however every position \( x \in X \) will be mapped to exactly one domain \( X_n \).

Finally we will consider the smoothness of the inverse models \( \pi_n : X_n \rightarrow S_n \), in order to see how well the agent can follow trajectories within a region \( X_n \), but also on the border between neighbouring regions \( (X_n, X_{n'}) \). We measure it in the amplitude of the gradient \( \nabla \pi(x) \), where \( \pi(x) \) is the combined policy for all regions so that \( \pi(x) = \pi_n(x) \) for \( x \in X_n \). We thus get smoothness:

\[
s(x) = ||\nabla \pi(x)||
\]

where in practice each partial derivative is approximated so that:

\[
\frac{\partial}{\partial e_1} \pi \left( \begin{array}{c} e_1 \\ e_2 \end{array} \right) \approx \pi \left( \begin{array}{c} e_1 + \delta \\ e_2 \end{array} \right) - \pi \left( \begin{array}{c} e_1 \\ e_2 \end{array} \right)
\]

and similarly for \( e_2 \), given \( x = [e_1 \ e_2]^\top \). In this case we set \( \delta = \frac{1}{100} \). To finally express the smoothness we use approximation:

\[
||\nabla \pi(x)|| \approx E_{\Delta x} \left[ \frac{||\pi(x) - \pi(x + \Delta x)||}{||\Delta x||} \right]
\]

This is calculated by picking \( \Delta x \) as the Moore-2 neighbourhood.

### 6.3.4 Results

To give an idea of the developed behaviours, see Fig. 6.11 for some particular behaviours that were developed for different settings. Most notable here is the fact that the agent needs more
Figure 6.11: Some examples of different behaviours developed for different settings, when asked to reach a goal on the other side of a wall. Most notable is the failure in the $1 \times 1$ case which corresponds to the assumption of continuous manifold GB that any goal can be reached from any starting condition. By using multiple smaller regions the agent is able to take the dynamics of different starting conditions into account, which allows it to plan trajectories around the wall in order to reach the goal.

than a single region for reaching goals that require some planning. The number of DoF seems to have a limited effect on the outcome.

When considering the full learning process, as for the home state approach, the arm starts by uncoiling into the starting position (Fig. 6.3). This initialises the dataset for the first region $X_n$ in which the agent ended up. The agent then chooses a goal $\hat{x}$ in the current region or one of its neighbouring regions, and tries to reach it. In the beginning this often leads the agent to end up in other regions $X_{n'}$ by accident, but as long as they have not been visited before this allows these regions being initiated as well. Over time attempted transitions succeeds more often and at this point the exploration strategy starts to become important. Fig. 6.12 compares the learning rate for different exploration strategies for a $6 \times 6$ segmentation, and Fig. 6.13 for a $10 \times 10$ segmentation. Both cases evaluates an arm with 100 DoF.

We see that for a lower resolution, such as $6 \times 6$, there is no real difference between an intrinsically motivated exploration and a random exploration, while the learn-by-doing approach performs slightly worse. The probable reason for this is that random actions and intrinsic motivation are more or less equally likely to try all transitions out, since there are only a small amount to begin with. The learn by doing approach however is likely performing worse due to its greedy behaviour. It is only doing what seems best for any iteration of the MDP model, effectively stopping it from finding more efficient planning strategies.

For $10 \times 10$ we see that the intrinsic rewards really begins to pay off with respect to the reaching error. The number of possible transitions here have increased so much that it starts to become important to actively seek out the contexts where a given MDP model can be improved the most. Interestingly we see an initial advantage for the learning-by-doing approach here. This is probably because such an approach would rapidly find one way to reach every region $X_n$, although it might not be the best. This gives it an initial advantage, which is later lost when the other methods learn the MDP model better and are thus able to plan more efficiently. The
intrinsic motivation approach manages to surpass the learning-by-doing approach after around 5,000 iterations after which it is clearly dominating, with the random exploration approach and the learning-by-doing performing essentially the same for the final iteration. An interesting observation is the difference in distributions for the different cases. For random exploration we see a band of positions that the agent never manages to reach, making the distribution bimodal. For the learn-by-doing approach, this other mode is somewhat reduced over time, while the intrinsic motivation approach is able to seek out more challenging areas and thus remove the second mode almost completely.

As the intrinsic motivation exploration works as well as, or better than the alternatives, this is the strategy that will be used henceforth in this experiment.

Secondly we look at the impact of different segmentations $d \times d$, shown in Fig. 6.14. Errors seem to be lower the higher the resolution, although it is worth noting that this trend cannot go on forever as a higher resolution leads to a more complicated MDP model to learn. A $100 \times 100$ segmentation would for example be impossible to learn within 10,000 iterations, as that would mean 10,000 nodes to visit, and around 90,000 different transitions to attempt. A $10 \times 10$ resolution is apparently not a high enough resolution for this to become a problem, and as long as this is not a problem it makes sense that a higher resolution leads to better results in this case. A higher resolution leads to smaller regions, which makes it easier to find fully connected sets, so that the outcome of an action is independent of where in the region the agent starts. For a smaller region this is clearly easier to achieve, whereas the case where the full task space is seen as a single region ($1 \times 1$) is the opposite. Although we see that this inverse model keeps improving throughout the training session, it is limited by the fact that the agent is not able to account for its current state. The $7 \times 7$ resolution is another interesting case as there are regions with a
Figure 6.13: Reaching error for $10 \times 10$ segmentations with 100 DoF arm, for different exploration approaches for the MDP model. Each pixel column represents error distribution of 20 independent runs, with 200 such columns for inverse error, and 100 for the reaching error. Lines show geometrical mean.

Figure 6.14: Reaching error for different segmentations with 100 DoF arm. Each pixel column represents error distribution of 20 independent runs, with 200 such columns for inverse error, and 100 for the reaching error. Lines show geometrical mean.

Wall passing right through them, see Fig. 6.10. This did not seem to matter too much however, as the performance is equivalent to $6 \times 6$. What seemed to happen was that the GB models would focus on developing the manifold on one side of the wall and reach for every goal from that side, leaving a small error for goals on the other side.

To see the impact of different DoF we chose a segmentation $6 \times 6$, leading to Fig. 6.15. What is significant in this case is that the number of DoF does not seem to be of any greater significance. This result is consistent with other results from ordinary GB [Rolf et al., 2011], and clearly shows that learning complexity is untangled from the size of the action space of the agent.
Figure 6.15: Reaching error for $6 \times 6$ segmentation with different DoF of the arm. Each pixel column represents error distribution of 20 independent runs, with 200 such columns for inverse error, and 100 for the reaching error. Lines show geometrical mean.

Figure 6.16: Reaching error for a longer run with $10 \times 10$ segmentation and a 100 DoF arm. Each pixel column represents error distribution of 20 independent runs, with 200 such columns for inverse error, and 50 for the reaching error. This means that the inverse error is first evaluated after 250 iterations, and the reaching error after 1,000 iterations. Lines show geometrical mean. The rightmost image shows postures used to reach some of the most challenging end-effector positions.

In order to see how stable the learning process is over time a longer evaluation was made with over 50,000 iterations. For this evaluation an arm with 100 DoF was used, and a $10 \times 10$ segmentation, seen in Fig 6.16. In this case we see that the reaching error catches up with the inverse error after roughly 10,000 iterations, which means that at this point, the agent will reach the region of any goal $x^*$ so that the inverse model can be used for each goal. At the end of this evaluation the arm could reach any goal with an (geometric) average error of $10^{-3}$, which is within 1 mm in a room with sides 1 m.

Finally we measured the smoothness $s(x)$ across the task space, as specified in Eq. (6.29). Fig. 6.17 shows the smoothness for different choices of DoF for a $6 \times 6$ segmentation, and Fig. 6.18 shows the smoothness for different segmentations, using a 100 DoF arm. Each result here is based on the learning of a single arm, as each run will allow direct transitions between different neighbouring regions. Averaging over many arms would obscure such results, as one arm might
Figure 6.17: Examples of smoothness $||\nabla \pi(x)||$ for different DoF, for a $6 \times 6$ segmentation, for single runs. Each image shows smoothness values for $100 \times 100$ task space positions, which corresponds to the pixels. In general the different arms performed similarly. Some anomalies include the 5 DoF arm failing to find manifolds for two regions, and the 200 DoF arm accidentally accepting two inconsistent sets of data in its lower left corner, creating two separate manifolds with a discontinuity in-between.

Figure 6.18: Examples of smoothness $||\nabla \pi(x)||$ for different segmentations for a 100 DoF arm, for single runs. Each image shows smoothness values for $100 \times 100$ task space positions, which corresponds to the pixels. Worth noting is how a discontinuous edge stretching out from the arm’s base, was created in the $1 \times 1$ case, and how the $7 \times 7$ segmentation was only able to find smooth manifolds for half the regions when the region had a wall in the middle.

enable a smooth transition between two regions $(X_n, X_{\tilde{n}})$ while another arm with the same resolution would use other regions to transition between them. The reason that not all regions can be connected to each other is a physical property of the system. If every region was connected directly to every neighbour, the arm would be able to travel $360^\circ$ around its base and be back in the first region. However, since every region only has one manifold $\pi_n : X_n \to S_n$ this means
that it would also need be back in the same state. This is however impossible since moving its
dend effector one turn around its base would leave it coiled up one turn, which means that it
cannot be back to the original uncoiled state. This means that somewhere when moving around
the base there must be an edge, so that the arm must go one turn around the base in the
opposite direction to reach the other side of that edge. This is clearly seen in the $1 \times 1$ case, where
a discontinuity is developed in the manifold from the centre and out to the upper left wall. In this
case the arm would attempt to reach any position above the discontinuity in a counter-clockwise
(CCW) fashion, and below it in a clockwise (CW) fashion. Once the space is segmented into
smaller regions these discontinuities are moved to the edges between the regions in order to
keep every single region smooth and fully connected (which is not physically possible in the $1 \times 1$
case). Note that even though some regions have an edge between them, it might still be possible
for the agent to transition directly between them. The edge only means that the arm cannot
move across it smoothly, without considerable posture change. In most cases though we see that
the proposed methods lead to large patches of continuous manifold, allowing trajectories to be
smoothly followed both within regions, and between them.

## 6.4 Discussion

When comparing the results of the home state approach (Sec. 6.2) and the continuous manifold
approach (Sec. 6.3) it is clear that they are approximately equivalent when it comes to reaching
error, with an error in the order of $10^{-2}$ units after $10,000$ iterations (see Fig. 6.19). This is a
reasonable outcome, considering that $10,000$ samples were collected in a task space of dimension
$1 \times 1$. Since $10,000 = 100 \times 100$ there would be a sample for every $\frac{1}{100} = 10^{-2}$ unit of the room,
provided evenly spaced samples. Any random goal $x^*$ would thus have some closest sample in
a distance of that order of magnitude. This assumes perfectly spaced samples, and that each
previously seen sample could be reached again without any further generalisation from the robot.
This is of course not exactly the case, but it provides some intuition on the order of magnitude
one could expect for the error using $10,000$ samples in a $1 \times 1$ unit$^2$ room.

The fact that both methods performed so similarly, both in terms of reaching errors and
the ability to handle in principle any number of DoF, despite how differently the GB models
where created, speaks to the power of the general concept behind both of them, described in Sec.
4.1. With this in mind it is fair to point out that despite these similarities there are also some
important differences. The home state approach seems relatively unaffected by the region radius
with respect to the reaching error, while the continuous manifold approach is strongly influenced
by a change in resolution. The reason for this is that the difficulty of finding fully connected state
sets scales with the size of regions, while the home state approach only needs to consider the
outcomes from a single home state. On the contrary we can also see in Fig. 6.15 that there is no
clear relationship between DoF and performance in the continuous manifold case, while the home
state approach seems to perform better for higher DoF. In this case a possible explanation is how
exploration was done for reach-out policies $\pi_n$ in the two cases. For the home state approach, where new samples are made as deviations of old, these deviations are likely to scale with the number of DoF of the arm, as noise is added to each joint. Noise levels optimized for a particular number of DoF would then naturally perform increasingly worse as the number of DoF changed. In the continuous manifold approach the difference in effect of goal state noise could have been compensated by the use of gradients with LLR, making step sizes adapt to the effect of noise levels.

When it comes to implementation, the continuous manifold approach required a significantly more complicated setup, with memory factors, predefined segmentations and large amount of additional data as the whole trajectory was used in this case. The setup is also less adaptable for learning multiple task spaces in parallel, as it is hard to know how to segment these additional task spaces so that a manifold in one task space also creates a manifold in the other. In the home state approach it is instead possible to just save the outcome $x^{(i)}$ in every task space $X^{(i)}$ given a goal state $\hat{s}$ from a home state $s_n$, as was done in [Forestier and Oudeyer, 2016] for a single home state. The continuous manifold did on the other hand produce the lowest reaching error, for the $10 \times 10$ segmentation. It is however not a completely fair comparison, as this approach got 100 samples for every transition, when the home state approach only received one. The overall
conclusion is thus that unless following trajectories smoothly is important, then a home state approach would be preferable.

It is worth pointing out that the home state approach comes with many unanswered questions. We still have not seen examples of how to handle situations where a recovery cannot be made. It is also not entirely clear how to connect different home states to each other when creating the MDP. Allowing attempted transitions between all home states will lead to a combinatorial explosion and subsequently an MDP that cannot be even approximately learned, leading to inefficient planning. Allowing too few connections on the other hand runs with the risk of making some home states unreachable all together, if neither assumed connection is usable. Another issue is that the home state approach does not allow a home state to change over time, in order to find a similar but, by some metric, more efficient position. Shifting a home state $s_n$ even slightly means that all samples in the corresponding dataset $D_n$ might become obsolete, as they represent the outcomes when starting from the previous value of the home state. In the continuous manifold case we did not really have this problem as the memory factor would allow the whole manifolds to drift over time, based on whatever goal states that managed to reach the region.
In this chapter we turn to the problem of controlling the torso orientation of a humanoid robot, acting in a real environment. Besides describing a completely different system, learning to control a physical robot acting in a real environment introduces a whole new set of problems. Running experiments on a real robot risks damaging it, which would in turn change the dynamics of the damaged robot, making any data it has collected so far potentially obsolete. It might also lead to a loss of control of some motors, so that some joints cannot be controlled any more. Each experiment is also more time consuming and is not easily parallelizable, which limits the number of iterations a robot can use for learning, and also limits the number of experiments that can be run in order to find appropriate settings for those experiments.

Body orientation control is here referred to as the control of a vector $\omega$, corresponding to the direction of gravity, relative to the reference frame of the robot's torso, see Fig. 7.1. Assuming perfect control this means that: a $\omega$ pointing straight forward (relative to the robot) would make the robot lie down on its belly; straight back - lie on its back; and straight down - to sit up. Note that some directions might not be physically reachable here. Gravity pointing straight up would require the robot to stand upside down, for example.

Learning to control the orientation of one's body is among the first skills human children need to master [Asada et al., 2009; Cangelosi and Schlesinger, 2015], and is fundamental to many other skills that first require the child to move into an appropriate starting position. Despite this, examples of successful learning of body orientation control are essentially absent in existing literature. It is however a suitable problem to the methods proposed in this work. It describes a system with a low dimensional task space (the orientation), high dimensional continuous action spaces, in this case the 25 degrees of freedom (DoF) of the robot, and a high dimensional state space, composed by both the 25 DoF posture, together with the orientation. The system is
Figure 7.1: The state of the robot at rest is characterised by a posture $q = [\alpha_1 \ \alpha_2 \ \ldots \ \alpha_{25}]^\top$ of 25 joint angles, and a torso orientation vector $\omega$ pointing in the direction of gravity. This direction can either be expressed in Cartesian coordinates $[e_1 \ e_2 \ e_3]^\top$, or Spherical coordinates $[\theta \ \phi]^\top$, where $\theta$ is the orientation latitude, $\phi$ the longitude.

Furthermore characterised by dynamics so that the outcome of an action depends on the state in which it was made, and many task space outcomes can only be achieved through a series of intermediary actions [Kuniyoshi et al., 2004]. A robot cannot go straight from lying on its back to lying on its belly, for example. Furthermore, learning is limited by the sparse data the robot is able to collect in a single run, which is here restricted to 1,000 iterations.

The body orientation problem will be tackled using the two basic approaches described in previous chapter, with minor adaptations. The first approach is once again home state based, the second is searching a continuous manifold. This allows us to evaluate how both approaches translate over different problems.

7.1 Experimental design

7.1.1 Physical description

For this experiment we consider the Nao robot from SoftBank Robotics, in versions v3 and v5, seen in Fig. 7.1. The robot is 58.0 cm tall and weighs 4.3 kg. It has 25 DoF, and a 3D accelerometer in its torso. All measured values can be accessed through a wireless connection. To protect the robot’s fingers from damage the hands were covered with paper and tape. Full documentation can be found at: http://doc.aldebaran.com/2-1/home_nao.html

7.1.2 Spaces

The task space $X \subseteq \mathbb{R}^l$ is here based on the orientation of the torso, given by the accelerometer $\omega \in \Omega$, so that $X = \Omega$, seen in Fig. 7.1. This direction can be expressed either in Spherical...
coordinates $[\theta \ \phi]^\top$ or Cartesian coordinates $[e_1 \ e_2 \ e_3]^\top$, where:

$$(7.1) \quad \sqrt{e_1^2 + e_2^2 + e_3^2} = g$$

for a system at rest, where $g$ is the acceleration of gravity. We normalise this data so that $g = 1$. Unless stated otherwise, we will use the Cartesian formulation, as it preserves distances between accelerometer readings on the surface $\sqrt{e_1^2 + e_2^2 + e_3^2} = 1$. The Spherical formulation on the other hand inflates distances near the poles $\theta = 0$ and $\pi$ (for the same reason Greenland is disproportionately large on most world maps), and includes periodic boundary conditions $\phi = \pm \pi$ that would need to be accounted for when comparing accelerometer readings.

The state space $S \subseteq \mathbb{R}^k$ of the robot is defined as a union of the posture space $Q \subseteq \mathbb{R}^{25}$, defined by postures $q = [\alpha_1 \ \alpha_2 \ \ldots \ \alpha_{25}]^\top$, and the orientation space $\Omega$ defined by orientation $\omega$, so that $S = Q \cup \Omega$. This gives us states:

$$(7.2) \quad s = \begin{bmatrix} q \\ \omega \end{bmatrix}$$

To include the orientation $\omega$ is, as discussed in Sec. 2.2, necessary in order to fully describe the configuration of the robot, since the same posture $q$ can potentially be compatible with many different orientations $\omega$, at rest. See Fig. 2.2 for an example. In addition, including $\Omega$ into the state space, which is also our task space $X$, ensures a relationship $\psi : S \rightarrow X$, so that every state corresponds to exactly one task space position.

For the action space we are here interested in a goal state description, so that $A = S$. In practice the robot is controlled using goal postures $\hat{q} \in Q$. This is practically solved by only using the posture part $\hat{q}$ of any given goal state $\hat{s} = [\hat{q}^\top \ \hat{\omega}^\top]^\top$, when sending a motor command to the robot.

### 7.1.3 Dynamics

Given a current state $s = \begin{bmatrix} q \\ \omega \end{bmatrix}$ and a goal state $\hat{s} = \begin{bmatrix} \hat{q} \\ \hat{\omega} \end{bmatrix}$, the agent will attempt to change posture to $\hat{q}$ for 1 second. At this point the posture has changed into a new posture $\tilde{q}$ which is set as the new goal state (so that the robot will hold it). After that the robot waits until reaching a steady state. This is assumed when the accelerometer measures two readings $\omega_t$ and $\omega_{t+1}$ with 0.1 seconds apart, so that:

$$(7.3) \quad ||\omega_{t+1} - \omega_t|| < 0.02$$

At this point we read the new state $s' = \begin{bmatrix} q' \\ \omega' \end{bmatrix}$, and a task space position $x' = \omega'$. This process is assumed to follow some forward dynamics equations:

$$(7.4) \quad f(s, \hat{s}) = s'$$
$$\zeta(s, \hat{s}) = x'$$
Note that for this problem we will not have access to any trajectory of intermediary points during the motion. The reason for this is that our states describes the robot at rest. Any measurement in-between might not be stable, and on top of this, any measured accelerometer reading during a motion would include other sources of acceleration than gravity, making them inaccurate.

**Speed and stiffness**

In order to make the robot attempt to move to a goal posture $\dot{q}$ it is not only necessary to specify the posture to the robot, but also a *stiffness* value\(^1\) for the motors, together with a *speed* at which the robot attempts to reach the goal posture. Both values are defined in the span $[0,1]$. It was found that a motor speed of 0.4 worked well for this problem. The stiffness was however more difficult to adjust properly. The main issue is that a high stiffness results in great stress on the robot’s gears, making them likely to break during the exploration that learning from scratch entails. A low stiffness, on the other hand, severely reduces the postures a robot is able to reach, decreasing its mobility. Both of these effects need to be balanced against each other. As each of our model solutions use different approaches to handle this problem, these descriptions are referred to the respective sections of each approach (Sec. 7.2 and 7.3).

**The fall manager**

In addition to defining speed and stiffness, there is also an in-built safeguard referred to as the “fall manager” in the robot. This is a preprogrammed behaviour designed to protect the robot if accidentally falling over. As we are only interested in the behaviours the robot can develop by itself, this fall manager was deactivated. Instructions for how to do so can be found at: http://doc.aldebaran.com/2-5/naoqi/motion/reflexes-fall-manager.html.

### 7.1.4 Procedure

The experimental procedure is described in Fig. 7.2. In addition it should be noted that great care is taken whenever the robot is moved to not change the posture or the orientation of the robot. Doing so might trick the learner into considering that a transition is possible although it is not. Even if that was not the case, the goal is to keep human interference to a minimum. While being moved the robot is frozen, as it is only allowed to continue once no additional forces than gravity is measured, see Eq. (7.3).

It is also worth pointing out that after every break in training after 200-250 iterations, the robot starts its next session from the original position on its back. Each robot will furthermore be trained independently, so that the training data from one robot will not be used by another. In the end, each robot will be allowed to train 1,000 iterations in total.

---

\(^1\)http://doc.aldebaran.com/2-5/naoqi/motion/control-stiffness.html
Figure 7.2: The experimental procedure. The robot starts each run in a neutral pose on its back, in the centre of a 150 × 150 cm$^2$ mat. Its hands are protected by paper and tape, and the robot is connected to a power outlet, so that the battery does not deplete. It is then allowed 200 − 250 iterations of free movement, during which the only occasional human interference is to move the robot back to the centre of the mat, reinsert the power plug of the robot, or prevent a joint from evident damage. After these iterations, training is paused and the robot is allowed to let its motors cool down and its battery to fully recharge, before later resuming the training.

7.2 Approach 1

We here consider the same home state based approach that was described in Sec. 6.2, with a growing set of home states $\{s_n\}_{n=1}^N$ together with assumed reachable regions $\{X_n\}_{n=1}^N$ defined by a radius $r$ from each home state’s position $x_n = \psi(s_n)$. The same code is used here as for the arm and can be found at: github.com/Loviken/MCGB

7.2.1 General adjustments

Although the method is kept as similar as possible to Chapter 6, some minor changes and adaptations are nonetheless made.

Generation of task space goals

Task space goals $x^*$ are generated by sampling a point $\tilde{x} = [\tilde{e}_1 \; \tilde{e}_2 \; \tilde{e}_3]$ with each element $\tilde{e}_i$ randomly sampled in the span $[-1, 1]$. The task space goal is then created by normalisation:

\[(7.5) \quad x^* \leftarrow \frac{\tilde{x}}{||\tilde{x}||}\]

corresponding to the sphere surface of 1 (with $g \equiv 1$), see Fig. 7.1. The robot is then allowed 10 iterations in order to reach a home state $s_n$ for which $x^* \in X_n$. Once a reach-out has been made from such a home state $s_n$, or once the robot has not been able to reach such a state within 10 iterations, a new goal is created.
Motor stiffness

To begin with, we need to define the motor stiffness when reaching for goal states $\hat{s}$. A low stiffness decreases the risk of damage for the robot, but limits the amount of states it would be able to reach. To handle these constraints we concluded two things. Firstly, the risk of damage is the greatest when the robot attempts to reach a state it has never observed before, which happens during reach-outs. Secondly, the importance of reaching a goal state is the greatest when attempting a transition, or when recovering, to specific home states $s_n$. As every home state has previously been observed by the robot, the risk of damage is lower. With this in mind we set reach-outs to a stiffness of 0.5, and transitions and recoveries to a stiffness of 1.0.

Test for assigning reached home states

Another thing to consider here is when to assume that a home state is reached. In a simulation it is possible for the agent to reach a previous state exactly, while in any real case there will be some deviation, both due to imprecise sensors, and imperfect motors. For this reason we will assume that a home state $s_n$ is reached if the robot attempted to reach the home state to begin with, and ended up in a task space position $x$ so that:

\begin{equation}
||x - x_n|| < 0.1
\end{equation}

where $x_n = \psi(s_n)$. The reason why positions in the task space were compared, and not the full states, is that very often some joint of the robot would stop working, meaning that it would be impossible for the robot to ever reach that state perfectly. If the robot still ended up in the same orientation, although it could not reach the proper posture exactly, we concluded that the reached state was similar enough to the original one to be assumed to have the same basic properties. In essence this allowed the robot to continue learning even if for example the neck rotation of the robot broke so that it could not control the head’s yaw.

State recovery

Whenever the robot ends up in a state that cannot be assigned to any home state (which happens after every reach-out), a state recovery is made. This is made by first reattempting to reach the last home state $s_n$ it attempted to reach. If the last action was a reach-out, this home state is the home state from which the reach-out was made. If this fails the robot will attempt to reach whatever home state is closest to it in the task space, 3 times. If still unable to reach a home state, the situation will be solved by redefining the home state closest to the robot in the task space to the current state of the robot. That is, if the robot is in a position $(s, x)$ and a home state $s_n$ minimises the distance $||x - x_n||$, then $s_n$ is changed so that:

\begin{equation}
s_n \leftarrow s
\end{equation}

When this is done, any dataset $D_n$ or probability $P(n, \bar{n})$ relating to the previous $s_n$ is also reinitialised, as the dynamics are assumed to have changed. Effectively this is equivalent to
adding a new home state at the current position of the robot, and removing the old one. Another option would be to simply add a new home state without adding a new one, but it was found that this would often lead to an explosion in the number of new states, making planning significantly harder.

**Addition of new home states**

As for the arm experiment, new home states are added whenever a part of the task space is reached that is not covered by any previous home state, i.e. if for the new task space position \( x \):

\[
(7.8) \quad ||x - x_n|| > r
\]

where \( x_n = \psi(s_n) \), and \( r \) is the region radius. This concept is expanded here (compared to the arm case) to also discard the regions \( X_n \) of any home state \( s_n \) that for any reason seems to have become unreachable, according to probabilities \( P(n, \hat{n}) \). This can happen for a number of reasons. The robot might have changed its dynamics during training, either by wearing down of motors/cogwheels or by different properties of motors with temperature or battery level. Another possibility is that the home state was not reachable to begin with, from any of the neighbouring home states. This can happen if the home state was first discovered during a state recovery, when trying to recover from some state that is also not a home state. It can also happen if the home state was discovered as a failure to reach some other state (which is usually the case). The “reach by accident” approximation, is, as described in Sec. 2.2.3, not always reliable. Excluding such unreachable home states from the test allows the agent to find new home states in regions only covered by seemingly unreachable home states.

**Accumulation of datasets**

Whenever an agent attempts to reach a goal state \( \hat{s} \) from a home state \( s_n \), it will observe a new state:

\[
(7.9) \quad s' = f(s_n, \hat{s})
\]

and a new task space position:

\[
(7.10) \quad x' = \xi(s_n, \hat{s})
\]

Since the point of the samples \((s_i, x_i)\) in the data set \( D_n \) is to suggest goal states \( s_i \) that bring the agent as close as possible to a task space position \( x_i \) (when starting from home state \( s_n \)), we append the goal state \( \hat{s} \) together with the task space result \( x' \) to the dataset:

\[
(7.11) \quad D_n \leftarrow D_n \cup \{(\hat{s}, x')\}
\]

Note that, as discussed in Sec. 4.2.1, this means that we mainly view \( \hat{s} \) as an action signal, and not as an actual state. We are only interested in the effect the goal has on the robot controller,
with respect to the task space outcome \( x' \). Intuitively it might seem better to instead associate \( x' \) to \( s' \), since clearly \( \psi(s') = x' \). However there is no guarantee that the robot will reach \( s' \) by attempting to reach \( s \). We do know however that the robot can reach \( s' \) by attempting to reach \( \hat{s} \), which is why we want to use \( \hat{s} \) as our action signal. This becomes increasingly important as reach-outs are done at a lower stiffness setting. In this case it was found that the robot would often undershoot its given goal states. This might mean that if a joint attempted to move from a value 0.0 to 1.0 it would stop at 0.8, but if instead asked to move from 0.0 to 0.8 it would stop at 0.6. Saving the goal state instead of the actually reached state circumvents this problem.

7.2.2 Evaluation

In this experiment we are mainly focusing on the effect of different radii \( r \) for the regions \( X_n \) of the home states, and their impact on the reach error. The reach error is here computed in two different ways. First we compute it as the Cartesian distance \( \epsilon_c \) between the goal \( x^* \) and the closest position \( x_{\text{min}} \) the agent had for some iteration, before a new goal was chosen. We thus have:

\[
\epsilon_c = ||x_{\text{min}} - x^*||
\]

Since task state positions \( x \) are computed in Cartesian coordinates \( x = [e_1 \ e_2 \ e_3]^\top \) on a unit-sphere \( ||x|| = 1 \), this means that we can also find the reach error angle \( \epsilon_a \) using the dot product identity:

\[
x_{\text{min}} \cdot x^* = \cos(\epsilon_a)
\]

By expanding the expression \((x_{\text{min}} - x^*) \cdot (x_{\text{min}} - x^*)\) we find the “Law of cosines” which for unit vectors becomes:

\[
||x_{\text{min}} - x^*||^2 = 2 - 2(x_{\text{min}} \cdot x^*)
\]

This means that:

\[
\epsilon_a = \cos^{-1} \left(1 - \frac{\epsilon_c^2}{2}\right)
\]

This is measured in radians \([\pi]\), which also corresponds to the geodesic distance between the two points on the unit sphere. In many ways this is a more natural measurement than the direct distance between two points as the robot will always need to travel along the surface of the sphere. For this reason this is our preferred measurement of errors, and it will also sometimes be used to display the radius \( r \) of the regions \( X_n \) (although the Cartesian distance is used by the robot for simpler computations).

As for the arm, one of our main questions of interest is the effect of the radius \( r \) of the regions \( X_n \). For this reason we will examine the the following radii:

\[
r = \{0.1, \ 0.2, \ 0.3, \ 0.4, \ 2.0\}
\]
Figure 7.3: Robots 2 and 3 had the cogs on one of their hip joints ground down during the setup of the experiment, during which experiments were made to find the relatively safe stiffness settings finally used. This was a great opportunity to evaluate the robustness of our approach to complex control dynamics, as our method does not assume a particular robot body in advance.

These are all measured in the Cartesian distance from any given home state position \( x_n = \psi(s_n) \). Observe that this means that a radius 2.0 includes the whole task space \( X \), since that is the maximum distance between any two points on a unit sphere. A radius \( r = 2.0 \) is thus equivalent with the Goal Babbling (GB) approach, where only a single home state is used. Each setting is run one time on each of 4 different robots, which will be referred to as robot 1, 2, 3, and 4. Of these, robot 2, 3 had broken gears, as seen in Fig. 7.3, which made them unable to use the hip roll angle properly. Since the behaviours are learned by interaction with the environment, and does not depend on some initial knowledge of how the robot ought to work, this should not affect the outcome more than that these robots might not physically be able to move to some states which the other robots can.

As a final illustration of the impact of the morphology of the robot in terms of performance, a final experiment was made where the body of the robot was altered by adding a cylinder to its back. This allowed us to see how the reachable parts of the task space depends on the particular morphology of the robot, and it also provides an example of the advantage of a learning paradigm that does not rely on the exact design of a robot being known in advance. See Fig. 7.11 for an example of this alteration.

As the order of experiments might be of importance here (as each experiment poses a risk of wearing the robot down), we start by testing the different radii in the order \( \{2.0, 0.2, 0.4, 0.1, 0.3\} \), followed by a single run with the adapted morphology.

### 7.2.3 Results

During the evaluation of the effect of different radii, it became clear that the method allowed the robot to discover many interesting states and large parts of its task space, as seen in Fig. 7.4. From the preprogrammed behaviours of the robot we know that it is physically able to sit up. We
Figure 7.4: The figure shows all task space positions ever discovered by any robot, together with some examples of the states with which the robot reached it. The examples 1-4 are all from a single run with robot 2 using radius 0.3. Example 5 is from a run with robot 3, using a radius 0.1 and was the only run close to sitting. A small radius seemed to help the development of this behaviour, but at the cost of other behaviours. The task space is here shown in spherical coordinates measured in units of $\pi$, where $\theta$ is the latitude and $\phi$ is the longitude. In all results $\theta = 0$ is defined so that the torso is pointing straight up, and $\phi = 0$ is the robot lying flat on its back.

never observed this however, and it is not clear if it is even possible considering that a lowered stiffness was used. To sit up requires the use of arms, and these seemed to be particularly weak and fragile in the context of exploring unseen goal states.

Task space coverage

The seen task space positions after 1,000 iterations is displayed in Fig. 7.5, for every robot and radius. The reason why radius 2.0 has more task goals $x^*$ is because $x^*$ changes after every reach-out, which is after every state recovery in a case with just one home state. Another observation is that robots 2 and 3 do not seem to be significantly influenced by their broken gears (see Fig. 7.3). At least not with respect to their coverage of the task space. Note that although this coverage mainly seems to be a central band, this band represents the equator of the unit sphere, whereas the large uncovered areas correspond to the poles, which are inflated due to the spherical projection. To get a clearer understanding of the coverage of the unit sphere, Fig. 7.6 illustrates the average surface distance to the nearest seen sample, as the learning progresses.
Figure 7.5: Every single run, by robot and radius, after 1,000 iterations. Blue dots (●) shows the distribution of reached positions $x_t$, and cyan crosses (×) shows the distribution of task space goals $x^*$. The reason why $r = 2.0$ has more goals is that the goal $x^*$ changes after each reach-out, which for a single home state happens after every state recovery.

**Reach error**

From the point of view of task space coverage, it seems like the radius $r = 0.4$ performed the best. What positions are reached is however not the most important consideration. What is more important is to what extent the robot is able to reach a given goal $x^*$ using its previous experiences, i.e. the “reach error”. In this experiment it is defined as the error after one reach-out to the goal, after which the goal moves. This means that the reach error here shows the performance when reaching for a goal that have never been attempted before, compared to the planar arm example, where the arm was allowed 20 iterations during which it could make multiple reach-outs. In this case, if the robot fails to make a reach-out within 10 iterations, the reach error is set to the distance to the closest position the robot come to the goal. The evolution of the reach error over all 4 robots can be seen in Fig 7.7, whereas Fig. 7.8 shows the reach error distribution.
Figure 7.6: The average surface distance to the nearest observed task space position for independent runs, measured by 10,000 uniformly random samples over the surface of the unit sphere. All averages are geometric. Gray background shows the span of the 4 different runs. The final image shows the average up until the indicated iteration, where the x-axis shows the region radii in radians.

When analysing the reach error, once again it seems like \( r = 0.4 \) is the best setting. In particular with respect to Fig. 7.7. This was however not the experience when later testing the robots by manually giving them different goals \( x^* \). The problem with such a large radius seems to be that it is often very hard for the robot to find transitions through the narrow reachable bridges that connect the states on the back to the belly. Often this rolling-over motion requires many smaller intermediary states, which was not permitted by such a large radius. The large radius also led to significantly fewer home states, which made it harder for the robot to find a reachable home state to recover to. As a home state is moved and reinitialised when the robot cannot return to it, a situation arose where many home states were constantly oscillating between different positions, without finding any that always worked. Why the reaching error is still low for \( r = 0.4 \) seems to be because the goals it can reach, it can reach with higher accuracy. To analyse the effect of small radius we consider \( r = 0.1 \). By most metrics this setting seems to be doing the worst after \( r = 2.0 \) (which is a single home state). The main problem here seems to be the difficulty of reaching any goal \( x^* \) before it moves. The robot also seems to have problems in general to plan between the large number of home states such a small radius entails. The setting was, however, the best setting for finding, and preserving, some of the states that seems the hardest to discover. Robot 3 with radius 0.1 came the closest to sitting up, as was seen in Fig. 7.4.
Figure 7.7: The reach error over time for 4 different robots. The black lines show geometric means, using a Savitzky-Golay filter with a window size of 100 iterations. Note that $r = 2.0$ covers the full unit sphere (in euclidean space) and corresponds to a single home state. The last image shows the geometric mean of the reach error vs. different radii. The reach error and the radii axes are here showed in radians. Note that $r = 0.3$ was the last experiment, which means that the robots were significantly worn down at that point.

Figure 7.8: The reach error over the unit sphere, approximated as the geometric mean of the reach errors of the 4 closest task space goals $x^*$ to each pixel. The heat map as well as the axes are in units of $\pi$.

**Overall performance**

In practice, radius $r = 0.2$ and $r = 0.3$ seemed to work the best in terms of having a robot that would be able to move to a large part of its task space if asked to. It should be noted however
Figure 7.9: Example of a complete 360° turn performed by robot 4 with $r = 0.3$, after 500 iterations ($\sim 30$ min). The uppermost image shows the model learned by the robot, and the lower images show the corresponding states when moving around in that model. The model here displays the home states together with the average probabilities to move in either direction between them. This means that if a transition can only be done in one direction this average becomes too low to be seen as an edge, which explains why there does not seem to be an edge between step 3 and 4. The regions $X_n$ are not shown here, as the spherical projection transforms euclidean distances in non-trivial ways, with singularities for $\theta = 0$ and $\pi$. The displayed circles simply indicate the centres $\{x_n\}_{n=1}^N$ of the home states, projected in spherical coordinates.

that since $r = 0.3$ was the last radius to be tried out, the robots where significantly worn down at this point. In addition, robot 2 and 4 had gears break down after 500 iterations during this last run, significantly decreasing their performance in subsequent iterations as it changed the dynamics on which previous learning was based. Prior to this many of these robots showed a great task space control, and were, for example, able to move freely between back and belly. Fig. 7.9 shows how robot 4, using $r = 0.3$, was able to perform a 360° rotation after 500 iterations. This behaviour was created by asking the robot to move from back to belly, and the back to the back. The figure also shows the set of home states, and how the rotation was planned and performed by the robot. Also $r = 0.2$ allowed such behaviours to be learned, as can be seen in Fig. 7.10 for the same robot. Note that more steps are used in this case, as a smaller radius puts neighbouring home states closer to each other.

**Morphology**

To see an example of the effect of a different morphology, a cylinder was attached to the back of robot 1, as seen in Fig. 7.11. Robot 1 was chosen as it was the least worn down, with no apparent
Figure 7.10: Alternative behaviour for turning $360^\circ$ with robot 4, radius 0.2 after 1,000 iterations. Steps 6 and 8 seems identical, but on closer inspection one can see that the robot used step 7 to remove load from the left arm, allowing it to be moved to a position from which the robot could make a push against the floor. At step 11 the robot had to be manually brought back into the centre of the mat. Note that a smaller radius means that the robot needs more steps for the transition in general.

Figure 7.11: The design of the altered morphology of Robot 1. A paper cylinder was added to the back of the robot with length 14.0 cm radius 2.8 cm.

Figure 7.12: The positions reached for robot 1 using a radius 0.3 after 1,000 iterations. The altered morphology is compared to the default one. Note that all points with a latitude near 0 are at the north pole of the unit sphere, and are thus very close to each other.

damage or malfunction. Using a radius of $r = 0.3$ it was soon clear that the change in morphology had a major effect as the robot was now able to use this new structure in order to move into a
sitting posture, which was discovered after around 400 iterations. Being able to move to a sitting position also provided a new means of moving from back to belly, as the robot could sit up and then fall forward with the legs to the sides.

In the end the positions of Fig. 7.12 were found by the agent, here compared to the results of the same robot and radius with the original morphology. It is very clear in this case that the set of reachable positions was profoundly altered by the change of morphology. Figure 7.13 gives some examples of particular home states and sequences of home states the robot learned.

### 7.3 Approach 2

Applying a continuous manifold solution to the body orientation problem is associated with some difficulties that did not apply to the home state approach, or when applied to the arm case. In the arm case it was possible to collect intermediary samples from each transition, which allowed the continuous model to smooth out the manifold in the intersection of different GB models. In this case such intermediary samples cannot be collected as our states $s$ correspond to stable states of the robot, i.e. when at rest. Even if we did collect such samples they would not be reliable, as the accelerometer only returns the direction of gravity in the absence of other forces. This means that the robot in this case is limited to only the final state of every transition.

Compared to the home state approach of the body orientation problem the issue is more
practical. In a home state approach we made a difference between transitions and reach-outs. This allowed us to use a low stiffness for more experimental goal states, and a higher stiffness when moving to states already seen (and thus relatively safe to attempt to reach). The continuous manifold approach does not make such a distinction as the manifold covers the full task space, and every task space position $x$ must be reachable with the state $\pi_n(x)$ on the manifold. This means that we need to treat every goal state the same way we previously treated transitions to home states, i.e. with as high a stiffness as possible. This however poses a great danger to the robot, as it would mean that it would attempt to reach states never observed with high stiffness. Such goal states can potentially lead to great stress on the robot cogs, especially the shoulders, hips and neck rotation. This poses a conundrum. Unless full stiffness is used, many necessary postures cannot be reached, but if full stiffness is used on the other hand, the risk is that many of the robots would break down during the training.

In the end these difficulties made us decrease the task space significantly to only include orientations between its back and sides. As this setup cannot be fairly compared to the previous where the full orientation space was considered, we here instead put our focus on simply establishing to what extent a continuous manifold approach is applicable to the problem. To investigate this, one of the most important factors is the importance of consistency. Great care is taken in the continuous manifold approach to only add new samples to a dataset if they are consistent with previous samples, as described in Sec. 5.4. The goal of this experiment is thus first to confirm whether a continuous manifold approach can gain control in the limited task space, and secondly to compare this result to what would happen if no consistency check was made at all. The full code can be found at: github.com/Loviken/FEGB/tree/iros-2018

7.3.1 General adjustments

Limitation of task space

Because of the especial risk of breakage, we decided to reduce the task space to only include the longitude $\phi$, and only in the span of back and sides, see Fig. 7.14. This means that the task space becomes 1D, so that every task space position $x$ corresponds to an orientation $\phi$. The reason for this particular choice was that this was the safest span in which high stiffness could be used without considerable risk of damage. States on the belly would often lead to transitions where the arms would take the full weight of the robot, and without the compliance of lower stiffness this would create great stress on the shoulder gears. Motions related to the latitude $\theta$ on the other hand were futile as all such motions would naturally require a higher stiffness (see example 5 of Fig. 7.4).

Segmentation of task space

The task space was segmented into 9 regions. Each of the 9 regions $X_n$ were defined by a centre placed evenly in span $\phi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$, with borders right between all centres (with a similar radius
Figure 7.14: Segmentation in 9 regions \( (X_n)_n=1^9 \) of the reduced task space, spanned by longitude \( x = \phi \) between back and both sides. The two sets of postures seen here were developed in two separate runs.

This leads to a radius of \( \frac{\pi}{16} \) in radians, meaning that the set of regions \( X_n \) cover a task space of \( \phi \in [\frac{-\pi}{2} - \frac{\pi}{16}, \frac{\pi}{2} + \frac{\pi}{16}] \) in total. Note that a region radius (in radians) of \( \frac{\pi}{16} \) corresponds to a radius of \( r \approx 0.2 \) measured in euclidean space on the unit sphere, to compare the setting to the home state approach. This comparison is not straightforward however, as there is no overlap of regions \( X_n \) in this case. A fairer comparison might be to consider the average distance between home states which is also the radius in the home state example, which, in this instance, becomes a euclidean distance of approximately 0.39. The best comparison, in terms of the average distance the robot will move in any transition, would thus be to a home state approach with radius \( r = 0.4 \).

**Wrapper to compensate for lower motor stiffness**

Given the limited task space it was possible to reduce the necessary decrease in stiffness from 0.5 (in previous experiment) to 0.8, where 1.0 is maximum stiffness. Although the decrease was not big, it lead to the robot controller starting to undershoot goal postures. This is a problem as in order for the robot to follow the task space manifold, it needs to actually reach the postures of that manifold. The problem is not necessarily that any goal posture cannot be reached, given this lower stiffness, but that the motor controller would decrease its torque when close to a goal, which means that the goal would not be reached completely. A joint angle requested to change from \( \alpha \) to \( \hat{\alpha} \) would undershoot and end up in an angle \( \alpha' \):

\[
\alpha' = \rho \alpha + (1 - \rho) \hat{\alpha}
\]

with \( \rho > 0 \). We thus know that the motor stiffness is sufficient for changing the angle \( \alpha \) into \( \alpha' \). However, if we then request a transition from \( \alpha \) to \( \alpha' \), it would end up in an angle \( \alpha'' \):

\[
\alpha'' = \rho \alpha + (1 - \rho) \alpha'
\]

It is thus clear that the controller of the robot is able to reach \( \alpha' \) at a lower stiffness setting, but not if asked to do so. Because of this, a wrapper was created around the controller of the
robot, so that instead of sending a goal posture $\hat{q}$ directly to the controller of the robot, it was sent to an intermediary wrapper that in turn sent its own “fake” goal postures to the robot’s controller, in order for it to end up as close to the actual goal $\hat{q}$ as possible. The wrapper is a PI-regulator with $P = 0.4$ and $I = 0.05$, where the posture of the robot was continuously sampled in a control loop at 100 Hz. Although no rigorous tests were made to capture the exact effect of this regulator, it was concluded from visual inspection that the alteration overcame many of the issues of undershooting that had previously stopped it from properly reaching many goal states.

Consistency and state recovery

In this experiment the robot is assumed to have reached a node $x$ if it is in a state $s = [q^\top x^\top]^\top$ for which $\hat{x} \in X_n$, and

$$\max_{\hat{q}}(|s - \pi_n(x)|) < \delta$$

where $\max$ only compares the posture part $Q$ of the state $S = Q \cup \Omega$, while $\delta$ sets a margin of how much any joint angle is allowed to deviate from the GB model $\pi_n$. We set $\delta = 0.5\pi$. If the robot is not in a state consistent with the GB model, a state recovery is made. This is done in the same way as was the arm experiment, described in Sec. 6.3.2.

7.3.2 MDP model

We are using the same intrinsically motivated Markov Decision Process (MDP) as for the arm. As in that case the robot is only allowed to attempt transitions between neighbouring states, which in this case includes the choice of staying in a region, or to attempt a transition to the region to the left or right of the current region. The outermost states are an exception here, as the only choices are to stay or move back inwards, towards $\phi = 0$ (see Fig. 7.14). Transitional probabilities are approximated identically to the arm experiment. If the probability to reach a given region falls under probability 0.2, it is forgotten and reinitiated.

7.3.3 GB models

The GB models $\{\pi_n\}_{n=1}^N$ are slightly simplified here compared to the arm case, in that the memory factor was dropped, and that Linear Regression (LR) was used rather than Local Linear Regression (LLR) (so that the entire dataset is always used). The memory factor was dropped mainly due to simplicity. To find appropriate memory decay rates would take significant testing, and the focus was mainly to confirm that the robot could learn to cover its designated task space. The reason why LR was used rather than LLR had more to do with the fact that we in this case got much fewer samples, as no trajectories could be collected. It was thus preferable to use all samples in the datasets $D_n$ rather than the subset closest to any given goal. We thus got inverse models/manifolds:

$$\pi_n(x) \leftarrow LR(x, D_n)$$
Before defining this as a goal state some exploratory noise was also added, so that:

\[
\hat{s} \leftarrow \pi_n(x) + \left( \frac{1}{d_n(x)} + \eta \right) U(0,1) \mathcal{N}(O,I)
\]

(7.21)

where \(\mathcal{N}(O,I)\) is a normally distributed noise-vector and \(U(0,1)\) is a uniformly random scalar to modulate the overall noise amplitude. The expression \(\frac{1}{d_n(x)} + \eta\) decreases the noise rate with the density \(d_n(x)\) of samples around \(x\) in \(D_n\), and \(\eta = 0.05\) is a static noise level to prevent the models from completely converging. This was necessary as there would otherwise be a risk that the dynamics of the robot would change, making previously reachable regions unvisitable.

### 7.3.4 Evaluation

As we could assume in this case that the robot should be able to reach its full task space and transition between any neighbouring regions \((X_m, X_n)\), a natural measurement is the average probability to perform transitions successfully. Since there are 3 possible actions for each node (except for outermost nodes) this results in 25 unique transitions, referred to as “state-actions” (SAs) in MDP terminology. These values can be computed the same way as probabilities are estimated by the MDP model, with the only difference here that we initiate these probabilities \(P(n, \hat{n})\) pessimistically to 0. For this evaluation 10 independent runs were made on 5 different robots (2 runs/robot). Each robot was trained for 1,000 iterations where the robot was allowed at least 10 minutes of rest after every set of 200 iterations (similar to the home state approach).

Secondly we want to see the importance of assuring state consistency when adding new samples \((s, x)\) to the datasets. This is done by running an independent run using pure Motor Babbling (MB), i.e. selecting goal states \(\hat{s}\) uniformly random. The datasets \(\{D_n\}_{n=1}^9\) were here constructed by assigning every seen outcome \((s', x')\) to the dataset \(D_n\) for which \(x' \in X_n\). The reason this method was used is that this type of data collection is among the most common for robots learning autonomously by interaction with an environment. This comparison was run for 600 iterations in 3 different runs with separate robots. Also in this case the robots were allowed to rest every 200 iterations. The reason why the total number of iterations were reduced was that motions based on random goal states, were much more damaging to the robot, than motions based on manifold estimates.

The MB approach was finally compared to the continuous manifold approach. Here, the MB approach is creating goal states \(\hat{s}\) in the same way as the continuous manifold approach, with the main difference that the two methods created their datasets \(D_n\) in different ways. The two methods are compared by observing how well each approach is able to do a random walk in the MDP, given goal nodes \(\hat{n}\) randomly. An agent always able to do a transition would statistically spend as much time in every node, whereas a robot unable to reach some nodes would be stuck in the subset of nodes it could easily transition between.
Figure 7.15: Two independent sets of postures found after 1,000 iterations, using the continuous manifold model. Most of the time, the robot would be able to transition directly between any neighbouring regions.

Figure 7.16: Probabilities of successful state-actions (SAs), where each SA corresponds to an attempted transition between two unique states. To the left: The mean probability of a state-action to be successfully performed, where the background indicates the distribution over all SAs where brighter colours represent higher density. To the right: The mean success probabilities of every SA over 10 different runs. The SAs are here sorted by size (after averaging).

7.3.5 Results

Fig. 7.15 shows found postures for two independent runs after 1,000 iterations. This corresponds to roughly 30 minutes of training time. Often the robot would have learned to cover the full task space significantly faster however. In https://www.youtube.com/watch?v=19zknXbXoD0 we can see a robot learning to reach all the regions already after 500 iterations. To see how well the robot manages to learn a manifold that allows it to transition between any neighbouring regions, Fig. 7.16 shows the estimated probability to perform each allowed action. We see that the robot starts by finding a smaller set it learns to control, to then expand this control out towards the remaining states. This corresponded to the robots first discovering postures related to lying on the back, and then progressively finding new postures that allowed it to progressively push further out towards the sides.

Finally, when comparing the advantage of using an online learning method like the one
proposed, compared to the more common approach of letting a robot collect data through random motor actions (MB), we see in 7.17a that the proposed method spends more time around its sides the more it learns, i.e. it is actively using the knowledge it previously stumbled over, in order to collect the data that will allow it to improve even further. For a MB approach however we see a strong tendency for actions to move the robot back to its back. This is significant since the outermost regions are (as far as we could tell) impossible to reach in a single transition. To reach them the robot must select a sequence of appropriate actions, where any exception would move it back to the back. For random actions such a sequence is very unlikely to happen, which is why we see such a sharp distribution towards the centre regions. Learning online thus seems to provide much better training data than random exploration. When evaluating the performance of each approach we see in Fig. 7.17b that the proposed method (which adds samples by consistency) allows the robot to transition more freely between all regions as more data is collected, meaning that more data gives it better task space control. The MB approach on the other hand (which does not ensure consistency in its data sets) gets more restricted to its middlemost regions, the more training data it collects. This confirms that in order to make generalisations over many data points, as is done with LR, separating different solution branches is essential.

Due to the effort to allow the robot to move at all between the back and sides in this case, no deeper analysis was made into the smoothness of the manifolds.

### 7.4 Discussion

Although both the home state approach (Sec. 7.2) and the continuous manifold approach (Sec. 7.2) allowed complex behaviours to emerge in matters of minutes, comparing the two is not completely straightforward. They considered different task spaces and they also used different measurements for proficiency. The reason why they used different task spaces and measurements...
is however very telling in itself. The home state approach could be applied relatively safely to the full orientation space, whereas the continuous manifold approach could not. The main reason for this inability was the need of the manifold approach to reach every goal state exactly. This was not possible for the full task space due to the fragility of the robot, together with its inability to reliably revisit previously reached states with a lowered stiffness. The home state approach was able to avoid this issue by using full stiffness for relatively safe transitions between home states, and a low stiffness when reaching out into unknown territories.

Considering the difference in evaluation methods: the fact that the home state approach learned by attempting to reach random goals \( x^* \), meant that reach error data could be collected continuously as the robot attempted to reach each goal. This was not possible in the continuous manifold approach as the intrinsic motivation approach was used (since this exploration method worked the best in the arm experiment). The problem with the intrinsic motivation approach is, however, that in order to approximate reaching errors, the training needs to be stopped so that the behaviour can be evaluated on a set of goals. This would considerably increase the time the robot has to interact with the environment, and in addition such evaluation would pose a risk of wearing the robot down, making whatever it had learned unusable.

In conclusion, it seems clear that a home state approach is preferable in this particular case. The advantage of having the robot following a continuous manifold is also not as obvious when controlling the body orientation. A continuous manifold still holds the promise of smoother motions of the robot, but as long as the robot is not strong and robust enough to follow such a manifold that cannot really be exploited. This might be the case for a smaller and lighter robot, a robot with more compliant joints, or even a soft robot, where breakage is less of a risk.
This work has introduced a novel online learning approach that is based on a combination of Reinforcement Learning (RL) planning in a sparse Markov Decision Process (MDP), and a set of local Goal Babbling (GB) models, able to learn exact motor actions for achieving outcomes within each model’s domain of expertise. Each component of the method is able to learn rapidly and independently. The GB part reduces the action complexity from that of the state-action space to that of the low dimensional task space, and the RL part allows the robot to plan transitions to reach the right contexts from which each GB models can be used. This makes the method applicable to real physical robots starting out close to tabula rasa, and allows them to learn to achieve freely specifiable goals in the task space after a training time in the order of only thousands of iterations, corresponding to half an hour to an hour of training in the humanoid robot case.

8.1 Summary

Chapter 2 provided a mathematical description of the systems we consider, and some important concepts were introduced such as goal state control and fully connected sets. We saw that goal state control can be useful even when parts of the state cannot be directly changed by the controller of a given robot. These uncontrollable parameters can often be influenced indirectly, but predictably, by the controllable ones, allowing some goal states to be reached. We finally described the control learning problem mathematically, and specified what acquiring task space control would entail to the described system. This allowed us to point out a set of particular challenges any solution needs to be able to cope with. These are:

- Sparse training data, as training time is limited.
• High dimensional continuous action and state spaces, as many robots have a significant number of degrees of freedom (DoF).

• The solution will need to be able to plan multiple transitions in a row, as many outcomes require the robot to first be in a small subset of states from which they can be achieved.

• The robot will not be able to automatically reset itself into some initial starting state, but must always act from whatever state it has previously ended up in.

• The solution must be able to handle a variable goal $x^*$, even if the exact goal has not been previously seen.

Chapter 3 investigated these learning problems from the point of view of some of the most popular learning paradigms in robotics, to see how they handle each of the identified challenges. The general conclusion was that although in theory, goal-conditioned RL set out to solve a similar problem to that described here, it would be fundamentally unable to do so here due to the large action space and short training time. Most RL methods depend on an exhaustive exploration and evaluation of every context a robot is likely to end up in, which is simply not feasible in this case. It was, however, found that another popular method, GB, was suitable exactly to the types of problems to which RL was unsuited. GB focuses on finding a single action to every desirable outcome in some low dimensional task space, and because of this it is able to reduce the complexity of the problem to the dimension of the task space, regardless of the dimensionality of the state or action space. The problem with GB on the other hand, is that it relies on an assumption that the outcome of an action only depends on the action, so that the state from which an action is made can be ignored, removing the need for planning. We thus have a situation where RL can solve the planning problem, but not the dimensionality problem, and GB can solve the dimensionality problem, but not the planning problem.

Chapter 4 introduced a new method, that combined a sparse RL model to multiple smaller GB models into a larger framework, in order to utilise the strength of each to overcome the weaknesses of the other. The proposed framework could then generate a global action-policy composed of four sub-policies: One recovery policy to reach the domain of some GB model; one transition policy to move from one GB model to another; one planning policy to plan transitions between GB models to reach a particular model as safely as possible; and one final policy for attempting to reach a particular task space goal $x^*$ from a particular GB model.

Chapter 5 went on to present different ways to implement and learn each sub-policy online. In particular it focused on ways to find a set of GB models, where each model was responsible for selecting appropriate actions from a certain domain of starting conditions, as well as an MDP model, that would learn how to transition between the domains of GB models, where each domain was represented as a discrete state in the MDP. For the MDP model learning problem a major decision was what exploration strategy to use, whereas learning of GB models could be reduced to a choice of which major branch of GB to use:
1. Branch one was referred to as the “home state approach”. In it, each GB model is defined by a single state $s_n$, called the “home state” from which every action is performed. This approach is easy to implement and has been shown to allow for the learning of multiple task spaces in parallel in traditional GB [Forestier and Oudeyer, 2016]. The general weakness is that the robot needs to reach the home state before every action, which means that the robot will not be able to follow a trajectory in the task space.

2. Branch 2 was referred to as the “continuous manifold approach”. The goal is here to find a manifold that maps every part of the task space to a particular state [Rolf et al., 2011]. This removes the need to return to a home state after every action, and allows the agent to follow a trajectory uninterrupted as long as it is able to reach every corresponding state of that manifold.

Chapters 6 and 7 finally investigated the capabilities of each approach on two distinct problems, both characterised by high dimensional state and action spaces, a low dimensional task space, in scenarios where planning was necessary in order to achieve many outcomes. Because of the nature of these problems they are both previously unsolved in literature. Chapter 6 investigated end effector control of a redundant simulated arm, acting in an environment with obstacles the arm has to plan transitions around. Chapter 7 investigated body orientation control of a real humanoid robot, with 25 DoF. The results showed that the proposed methods were able to achieve high levels of control in both cases, leading to complicated behaviours such as avoiding walls in the arm case, and planning body motions in many steps in the robot case, allowing it to move between back, sides and belly, and under some circumstances to sit up. These behaviours developed rapidly as well. The humanoid robot was only allowed 1,000 iterations, corresponding to 30-60 minutes of training, while the simulated arm was allowed 10,000 iterations. Results also showed that the methods were robust, as they worked equally well for an arm with 1,000 DoF or 5 DoF. They were similarly unaffected in its applicability to a robot with broken cogs or a significantly altered body. As the method would learn from scratch it would simply adapt to whatever body the robot had, requiring no previous knowledge or assumptions of its functionality. The greatest drawback for the physical robot was however its fragility, which meant that cogwheels could break down mid-training if full motor torque was used. This was particularly a problem in the continuous manifold approach, which required a significant simplification of the task space in this case in order to protect the robots. Overall, the fact that two quite different implementations of the underlying framework worked so well for two completely different control learning problems, speaks to the power of the underlying idea presented in Chapter 4.

8.2 Contribution

The main contribution of this thesis is to provide a framework for solving a control learning problem, previously more or less unexplored with no obvious methods of solving, but with strong
analogies to, for example, early learning of body orientation control in infants [Cangelosi and Schlesinger, 2015]. This thesis examined two quite different implementations of the proposed framework, and showed success in both cases for two different control learning problem.

### 8.2.1 Solutions to core challenges

When the task space control problem was first introduced in Chapter 2, we presented 5 particular challenges that would be necessary to solve in order to solve the full problem. These were: “Sparsity”, “no reset”, “dimensionality”, “planning” and “risk of damage”. We will now see how each of these were tackled by the final framework presented in this work.

#### Sparsity

Sparsity refers to the fact that only a few samples could be collected “in the lifetime” of a robot. The sparsity restriction was fundamentally tackled by the division of control between the MDP model and the GB models. The MDP model is a sparse tabular model, where the number of possible transitions is strictly limited, and the GB models have the advantage of only needing to find one action for every task space outcome, which means that samples only need to cover the low dimensional task space, and not the entire action space. Since each of these models can be reasonably approximated with relatively few samples, sparsity was manageable.

#### No reset

The problem of not being able to reset the robot into a previous state is that it is not necessarily possible to go back to a previous state in order to see what a different action would have done. To attempt different actions from the same state is often necessary in machine learning, in order to evaluate how to best act in a given situation. In this work this was solved by considering a whole set of GB models, which where defined so that the agent would always be able to “reset” itself to the domain of one of them. If that was not possible a new GB could simply be made (or an old could be moved) to the agent's current position. This effectively made the agent able to reset to a smaller set of states, solving the “no reset” problem.

#### Dimensionality

The issue with dimensionality is that the combined space of states and actions an agent could potentially attempt grows exponentially with the number of DoF of the agent. This was solved by not considering every possible state-action combination an agent can attempt, but rather to focus on a single action for every task space outcome, from a small set of initial states. As for GB, this effectively scales the learning complexity to the low dimension of the task space [Benureau, 2015; Rolf et al., 2011].
Planning

The problem with planning was how to decide on a sequence of actions, that would result in a sequence of states, so that the final action led to a desired task space outcome. The challenge here is the combinatorial explosion of such action sequences, and the need for a deep understanding of the effect of every action with respect to the subsequent state. The framework presented in this work solved this by reducing the planning problem to moving to the appropriate node in the MDP, created by the domains of the set of GB models. This meant that we did not need to consider planning in the entire state-action space, but only for a small MDP with a small set of possible actions at every node, in the form of allowed goal nodes. Efficient planning in such a small model could then be made using, for example, an optimal action-value function, as was done here.

Risk of damage

Since a robot learning from scratch cannot know in advance what actions are dangerous to it, there is always a risk with every attempted action that it might damage or wear itself down. One of the most important ways to reduce this risk is to limit its training time. Fewer actions means less wear down, which is also important if the agent should be able to rely on data it collected earlier. Secondly we saw that for the humanoid robot, motor stiffness was a big issue. Lower stiffness means less risk of damage, but at the cost of not being able to reach goal states. One solution to this was to separate actions that had previously seen from more exploratory actions, as was done in Sec. 7.2. Movements between home states could then be done with high stiffness, as they were already observed and safer (although not perfectly safe), while exploratory actions where the agent attempted to reach new task space goals could be done with softer motor settings. This work does not provide a clear answer for eliminating the risk of robot damage however, and many robots did wear down in the setting up of, and collecting results from the experiments of this work. It is not clear however if the risk of damage can be further reduced here from a software point of view, for a robot learning from scratch. Even animals and babies fall and hurt themselves considerably in the process of acquiring body control. It might therefore be more useful at this point to focus on the development of more robust robots. One promising such field is the field of “soft robotics” [Kim et al., 2013]. Soft robots are often very robust, but have previously been held back by a difficulty to model them properly, although this is a rapidly developing field [Duriez, 2013; Duriez and Bieze, 2017; Haddadin et al., 2018]. For this reason an important field of soft robotics is how such robots can learn skills autonomously [Cheney et al., 2014; Ishige et al., 2018; Nakajima et al., 2015; Rolf and Steil, 2013; Thuruthel et al., 2016], which makes soft robotics an interesting target for the framework developed in this work.

8.2.2 Implication for future applications

One great advantage of having a method that is able to rapidly access what a robot is able to accomplish with respect to a particular task space (and potentially multiple task spaces in

123
parallel) is that it could also be used as an automatic way to evaluate a particular morphology of a robot. As we saw when attaching a cylinder to the back of Nao in Fig. 7.12, changing the body of a robot can strongly influence what that robot is able to accomplish. A method like the one presented here could thus be a great aid in the design of robot bodies, as it provides a relatively quick feedback on the capabilities on any particular body [Müller and Hoffmann, 2017]. This ties to the field of evolutionary robotics, such as [Cheney et al., 2018, 2014; Sims, 1994] where brains and bodies are developed together, in order to optimise performance of various tasks.

Another particularly interesting implementation of the framework presented in this work, is as previously described in the context of soft robotics. Given that soft robotics are generally hard to model, it would be very useful if they could instead learn directly in the real world how to use their bodies to perform any given tasks. A particular issue here is to what extent such a robot could reach a limited set of domains \( \{S_n\}_{n=1}^N \) in order to limit the state spaces from which actions are evaluated. If this is not possible, and actions from every possible state would need to be considered, it would not be possible to break down the state space complexity which was necessary for framework of this work.

8.2.3 Limitations

The method presented here observes systems characterised by static states, where the outcomes of actions are assumed to be consistent with respect to the state in which the actions were made. It is not clear how such a framework can be applied to a system characterised by non-static components, with more dynamic actions such as torque control for example [Peng and van de Panne, 2017]. The proposed method instead deals with transitions from one stable state to another, although there is room for more dynamic effects during reach-outs [Baranes and Oudeyer, 2013; Forestier et al., 2017; Forestier and Oudeyer, 2016]. This ability can in practice be extended upon, so that the presented framework effectively becomes a “reset” routine for some other machine learning method. Consider for a humanoid robot, if a home state in the framework corresponds to a standing up state, which the robot is then always able to return to through a sequence of other home states. It would then be possible to use some other machine learning technique, for example deep RL [Heess et al., 2017; Lillicrap et al., 2015], to attempt to optimise a gate from this starting state. After each roll-out of such a network, the robot would be able to return to the starting state to try again. How the framework of this work would find such a set of home states, where one is a standing up state, is however not known. There is a big difference between sitting up and standing up, and it is not even clear what types of task spaces would require such states to be discovered.

The choice of task space itself is another limitation. In this work task spaces were manually chosen, typically with some set of potential behaviours related to that task space in mind. How to find interesting task spaces autonomously is therefore an interesting topic, with many unanswered questions. In particular, it would be necessary to clearly define what constitutes
a “good” task space, so that the behaviours learned by controlling such a task space might be considered useful or relevant by an outsider observer. Some previous work on this topic include the works of [Cartoni and Baldassarre, 2018; Jonschkowski and Brock, 2015; Laflaquière et al., 2015; Péré et al., 2018; Rolf and Asada, 2014], and could be a good starting point for such an endeavour.

Another limitation of this work is that the method primarily attempts to learn to find one way to achieve any outcome (although there might be an overlap of regions $X_n$ in some cases). This is the reason why it could learn so fast, but it is also a potential bottleneck in what the robot is able to learn. In the planar arm we would for example see that the arm sometimes found ways to reach some task space regions in such a way so that it would block itself from reaching regions even further away in the task space. It could therefore be useful for the arm to learn multiple ways to achieve the same outcomes, and then be able to learn which one allows further development the best. The risk with this is that it would easily explode in different alternatives, without a clear way to distinguish interesting new solutions from meaningless alterations of old ones. One idea for how to do this is to learn multiple task spaces in parallel. New GB models could then be created to overlap old ones in one task space, as long as they provide some novel control in at least one of the task spaces. This way the number of solutions is held down while at the same time allowing for overlapping GB models in some task spaces. Learning multiple task spaces in parallel has previously been shown effective in [Forestier et al., 2017; Forestier and Oudeyer, 2016; Oudeyer et al., 2007], since a new discovery in one space might open up for novel discoveries in other spaces.

Finally, as this work introduces a new framework with a main goal of showing its feasibility to a previously unexplored set of problems, there is no detailed analysis of exactly how to best implement it. It is not even clear how to best evaluate the performance of any such implementation. In this work we attempted to mainly focus on the “reach error” when possible, but even how to measure this is not completely obvious. Should it be how close the robot gets to a random goal from a completely random starting state, or by following some distribution of states it is more likely to be found in? How many iterations should the agent be allowed to reach the goal? Should the goal move after a reach-out? What is the reach error if the agent never got to attempt a reach-out? Regardless of the exact definition of the reaching error, there seems to be some important properties it does not capture, with respect to the quality of a particular implementation. In principle a lower reach error should equate to a better model, but this was frequently not the experience when observing and testing the learned models manually (by providing different goals $x^*$ manually). For the continuous manifold solution on the planar arm for example, Sec. 6.3, a $10 \times 10$ segmentation got a better reach error than the $6 \times 6$ segmentation, although when manually testing the system for different goals the general experience was that the $6 \times 6$ was better to move to the appropriate region. The reason for this was likely that it was better at learning the properties of the MDP, whereas the $10 \times 10$ solution would occasionally move into dead ends. Similarly did a 0.2 or 0.3 radius seem better than 0.4 in the humanoid robot case, for
the home state approach Sec. 7.2. These smaller radii would be better at transitioning through the side states necessary to move from back to belly. Still, when observing the reach error a radius of $r = 0.4$ seemed like the better choice. A possible explanation for this disparity between measured and experienced performances is that most randomly drawn goals are by accident in places easy to reach, while a manual examination would focus on goals known to be particularly challenging to reach, given the current state of the agent. The worse implementation might then perform better at goals that are easy to reach, which compensates its worse performance on the statistically few goals that are more challenging.

8.3 Future work

Some different topics that would be interesting for future research follows:

• **Segmentation of the task space** in the case of a continuous manifold. At the moment this is a clear limitation of that approach. It requires outsider intervention with some initial understanding of the structure of the task space, so that regions get appropriate sizes and do not cross over potential barriers. An automatic segmentation would remove this element of human interference, but would include many challenges such as how to update these regions over time, both in terms of shape, as well as optimal number of regions.

• **Addition of new home states**, in the case of a home state approach. How can an agent find the best home states set $\{s_n\}_{n=1}^N$ in order to move around in a given task space and reach as large parts of the task space as possible? In this work this was done by simply adding a new home state whenever a new task space position was observed, outside the assumed reach $\bigcup_{n=1}^N X_n$ of previous GB models. This is however very simplistic, and it should be possible for the agent to optimise these positions over time. How to do so is however unclear, and a big challenge is that any dataset associated to a particular home state can only be assumed to work from that starting position. Moving a home state might therefore make previous data collected from that home state inaccurate.

• **Approximation of reach-out regions** $X_n$, in the case of a home state approach. This is essentially the “unknown range problem” discussed in Sec. 3.2.2, but extended to multiple GB models in parallel. As we later saw in Fig. 6.8 and 7.5, particularly in the single home state cases, actions are executed from a specific state only able to generate outcomes in particular parts of the task space. If the regions $X_n$ could be updated online to correspond to these parts, the agent would be able to remove the asymptotic model error that occurs when including unreachable goals into $X_n$. The problem here is how to determine whether a goal is truly unreachable, or if the agent has just not found an appropriate action yet.

• **State recoveries**. In this work state recoveries were performed by attempting to select GB domains in some given order, and then attempt to transition directly to them. How to choose
such an order is an open question, where a main problem for any type of generalisation, is the dimensionality of the space $S$ of possible states an agent might need to make recoveries from. Even if some GB domain would be reachable from every state $s \in S$ there is no clear way to determine which one. In addition, any failed attempt to reach a GB domain would lead to a new state $s' \in S$, meaning that it would not be possible to exhaust a list of potentially reachable domains from an agent’s current position.

- **Initialisation of MDP edges.** What nodes should be assumed to be connected to each other in the MDP? If the agent would be allowed to attempt transitions between any two nodes it would lead to an explosion in the number of possible transitions, which would be impossible to approximate with limited data. Too few transition on the other hand would limit an agents ability to plan. We saw this in the case of a small radius $r$ for $X_n$ in the home state case, where transitions were limited to the nearest neighbours in the task space. Since a small radius in this case leads to more transitions in order to get close to any goals $x^*$, the result was that the goal moved before a reach-out was even possible. Small radii however seemed useful in other ways as they allowed for more efficient exploration in some respects. For example $r = 0.1$ lead to the closest outcome of “sitting up”, in the unmodified humanoid robot case, shown in Fig. 7.4. One potential solution that would allow both long transitions, as well as detailed exploration, would be to select a few nodes between which longer transitions are allowed. This could allow the agent to quickly get to the general area of a goal $x^*$, and then move more precisely to the most appropriate node. Exactly how to select or identify such a set of more connected nodes is, however, not known.

- **Learning multiple task spaces in parallel.** As discussed in Sec. 3.2.5, learning of multiple task spaces in parallel can bootstrap the development of all spaces, in the standard GB case. It would therefore be interesting to see how this property carries over to scenarios where multiple home states are being used.

- **Ghost states.** It could be worth investigating how to increase an agent’s ability to transition between different home states. In the implementation presented here the robot would attempt to reach a home state $s_n$ by using $s_n$ as its goal state. Very often however, a home state would be discovered by attempting to reach some other state $\hat{s}$. As we saw in Fig. 2.3, sometimes, just because a state could be reached accidentally, does not mean that it can be reached intentionally. It could thus make sense to also add $\hat{s}$ as a home state, although it has never been observed and might not even be physically feasible. By adding it however, the robot could learn a probability $P(s_n | s, \hat{s})$ that would return the probability to end up in $s_n$ from $s$, by attempting to reach home state $\hat{s}$. The state $\hat{s}$ would then be some sort of a “ghost home state”, which can be used for useful failures.

- **Safe exploration.** How can a robot that is learning from scratch, select actions that leads to as little wear down as possible, without significantly impairing its learning capabilities?
How to do this from a software point of view is very challenging, and includes multiple factors to weigh against each other. A common approach when real robots are being used is to estimate “safety regions” in advance, but that removes from the whole element of autonomous learning. Less motor torque also reduces this risk, but decreases the set of effects a robot can achieve through its actions. To animals and humans, feedback from pain and touch receptors provides important feedback for such exploration. It is possible that similar feedback could be equally useful in the field of robotics.

In conclusion, it would be useful to see how widely a method as the one presented in this work can be applied. What are its limits? What other types of control problems can it be applied to? What types of task spaces can be considered? Could it for example be used to allow a robot to learn to stand up? What kind of task space would that require? The orientation space would be insufficient in that case, as standing would only be seen as having the torso pointing upwards, which can be done much easier by sitting up. Possible choices would be to add the head’s height over ground, or the amount of feet contact with the ground. Such spaces could potentially be used in parallel with the orientation space used here.
Hindsight experience replay.

Cognitive developmental robotics: A survey.
IEEE transactions on autonomous mental development, 1(1):12–34.

Active learning of inverse models with intrinsically motivated goal exploration in robots.

Deep imitation learning for playing real time strategy games.
Cs229. stanford. edu.

Transfer learning for reinforcement learning on a physical robot.

Bechtle, S., Schillaci, G., and Hafner, V. V. (2016).
On the sense of agency and of object permanence in robots.
In 2016 Joint IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob), pages 166–171. IEEE.

Self Exploration of Sensorimotor Spaces in Robots.
PhD thesis, Université de Bordeaux.

Resilient machines through continuous self-modeling.
Inter-finger coordination and postural synergies in robot hands via mechanical implementation of principal components analysis.

Sequential composition of dynamically dexterous robot behaviors.

*Developmental robotics: From babies to robots.*
MIT Press.

Autonomous discovery of the goal space to learn a parameterized skill.

A survey on policy search algorithms for learning robot controllers in a handful of trials.

Scalable co-optimization of morphology and control in embodied machines.

Evolved electrophysiological soft robots.

Hand posture subspaces for dexterous robotic grasping.

Locally weighted regression: an approach to regression analysis by local fitting.

Evolving coordinated quadruped gaits with the hyperneat generative encoding.

World Scientific Publishing Company.
Feudal reinforcement learning.

A survey on policy search for robotics.


Novel plasticity rule can explain the development of sensorimotor intelligence.
*Proceedings of the National Academy of Sciences*, 112(45):E6224–E6232.

Learning modular neural network policies for multi-task and multi-robot transfer.
In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2169–2176. IEEE.

Learning inverse kinematics.
In *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No. 01CH37180)*, volume 1, pages 298–303. IEEE.

One-shot imitation learning.

Control of elastic soft robots based on real-time finite element method.

Soft robot modeling, simulation and control in real-time.

*The Cambridge dictionary of statistics*.
Cambridge University Press.
Model-agnostic meta-learning for fast adaptation of deep networks.

Intrinsically motivated goal exploration processes with automatic curriculum learning.

Modular active curiosity-driven discovery of tool use.
In 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 3965–3972. IEEE.

Curiosity driven reinforcement learning for motion planning on humanoids.
Frontiers in neurorobotics, 7:25.

The free-energy principle: a unified brain theory?

Learning actionable representations with goal-conditioned policies.

Continuous deep q-learning with model-based acceleration.
In International Conference on Machine Learning, pages 2829–2838.

Exploiting elastic energy storage for “blind” cyclic manipulation: Modeling, stability analysis, control, and experiments for dribbling.

Inverse reward design.
In Advances in neural information processing systems, pages 6765–6774.

Mosaic model for sensorimotor learning and control.
Neural computation, 13(10):2201–2220.

Geometrical motion planning for highly redundant manipulators using a continuous model.
A continuous approach to robot motion planning with many degrees of freedom.
In IROS, pages 1935–1942.

Emergence of locomotion behaviours in rich environments.

Hierarchical approaches.
In Reinforcement learning, pages 293–323. Springer.

Using slow feature analysis to extract behavioural manifolds related to humanoid robot postures.
In Tenth International Conference on Epigenetic Robotics, pages 43–50.


Learning oscillator-based gait controller for string-form soft robots using parameter-exploring policy gradients.
In 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 6445–6452. IEEE.

Learning state representations with robotic priors.

Unsupervised real-time control through variational empowerment.

Soft robotics: a bioinspired evolution in robotics.

Embodied basis of invariant features in execution and perception of whole-body dynamic actions—knacks and focuses of roll-and-rise motion.
Task-agnostic self-modeling machines.

Learning agent's spatial configuration from sensorimotor invariants.

Le Ny, J. and Pappas, G. J. (2012).
Sequential composition of robust controller specifications.

End-to-end training of deep visuomotor policies.

Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection.

Continuous control with deep reinforcement learning.

Lindley, D. V. et al. (1956).
On a measure of the information provided by an experiment.

Curiosity based self-organization of humanoid robot.
Master's thesis, Chalmers University of Technology.

Safe reinforcement learning with model uncertainty estimates.

Passive dynamic walking, the international journal of robotics research, 9 (2), 62-82.

Gaussian processes for regression: a tutorial.
Human-level control through deep reinforcement learning.

Middleware for robotics: A survey.
In *RAM*, pages 736–742.

*Frontiers in psychology*, 4:1006.

Mouret, J.-B. (2016).
Micro-data learning: The other end of the spectrum.

Autonomous learning of high-level states and actions in continuous environments.

What is morphological computation? on how the body contributes to cognition and control.

Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning.
In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE.

Information processing via physical soft body.
*Scientific reports*, 5:10487.

Effect of exploratory perturbation on the formation of kinematic synergies in goal babbling.
In *2015 Joint IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*, pages 86–91. IEEE.

Model learning for robot control: a survey.
Intrinsic motivation systems for autonomous mental development.

Learning locomotion skills using deeprl: Does the choice of action space matter?

Unsupervised learning of goal spaces for intrinsically motivated goal exploration.

Policy gradient methods for robotics.

Parameter space noise for exploration.

Temporal difference models: Model-free deep rl for model-based control.

Gaussian processes in machine learning.
In *Advanced lectures on machine learning*, pages 63–71. Springer.

Learning inverse statics models efficiently with symmetry-based exploration.
*Frontiers in neurorobotics*, 12:68.

Reservoir computing with output feedback.

Learning versatile sensorimotor coordination with goal babbling and neural associative dynamics.
In *2013 IEEE Third Joint International Conference on Development and Learning and Epigenetic Robotics (ICDL)*, pages 1–7. IEEE.
Attractor-based computation with reservoirs for online learning of inverse kinematics.
In ESANN, pages 257–262.

A 3d steady-state model of a tendon-driven continuum soft manipulator inspired by the octopus arm.
Bioinspiration & biomimetics, 7(2):025006.

Goal babbling with unknown ranges: A direction-sampling approach.
In Development and Learning and Epigenetic Robotics (ICDL), 2013 IEEE Third Joint International Conference on, pages 1–7. IEEE.

Where do goals come from? a generic approach to autonomous goal-system development.

Efficient exploratory learning of inverse kinematics on a bionic elephant trunk.

Rolf, M., Steil, J. J., and Gienger, M.
Goal babbling permits direct learning of inverse kinematics.

Online goal babbling for rapid bootstrapping of inverse models in high dimensions.
In Development and Learning (ICDL), 2011 IEEE International Conference on, volume 2, pages 1–8. IEEE.

The robots are coming.
Foreign Aff., 94:2.

Artificial intelligence: a modern approach.
Malaysia; Pearson Education Limited.

Empowerment—an introduction.
Evolution strategies as a scalable alternative to reinforcement learning.

Universal value function approximators.
In International Conference on Machine Learning, pages 1312–1320.

Goal-directed learning of hand-eye coordination in a humanoid robot.
In 2015 Joint IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob), pages 168–175. IEEE.

Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts.

Schmidhuber, J. (2010).

On-line regression algorithms for learning mechanical models of robots: a survey.

Evolving virtual creatures.

‘improving ratings’: audit in the british university system.

Proximodistal exploration in motor learning as an emergent property of optimization.


Learning global inverse statics solution for a redundant soft robot.
Domain randomization for transferring deep neural networks from simulation to the real world.  

Mujoco: A physics engine for model-based control.  

Feudal networks for hierarchical reinforcement learning.  


An action perspective on motor development.  

Interaction between learning and development.  

*Applied linear regression*, volume 528.  
John Wiley & Sons.

Simple statistical gradient-following algorithms for connectionist reinforcement learning.  

Dcob: Action space for reinforcement learning of high dof robots.  