Diss. ETH No. 20271

Motor Control with Graphical Models

A dissertation submitted to the
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
DENNIS GÖHLSDORF
Dipl.-Inf. Eberhard Karls Universität Tübingen
Dipl.-Biochem. Eberhard Karls Universität Tübingen
born April 18th, 1978
citizen of Germany

accepted on the recommendation of
Prof. Dr. Rodney J. Douglas, examiner
Dr. Matthew M. Cook, co-examiner
Prof. Dr. Jörg Conradt, co-examiner

2012

b

Abstract

Biological nervous systems can deal with many real-world natural stimuli and outperform any computer algorithm available today in most tasks involving interaction with real objects as required for example in interactive motor control problems. Moreover, they are able to learn an internal model of a controllable system solely from experience, unlike today's most successful control algorithms. The computational architecture expressed by brains is fundamentally different from modern computer hardware, in that it is composed of billions of small computational units with information stored only in the effectiveness of the communication channels between these units. Despite this restricted form of information storage, brains are able to devise complex sequences of actions which span many time scales.

We investigate neurally inspired algorithms that learn how to control systems of unknown mechanical structure solely from data collected from experience. These algorithms are designed to run on distributed computational architectures composed of many small and locally operating units. Further, we explore how motions spanning multiple time scales can be planned on such architectures.

Graphical models with discrete-state variables are used to model the relation of both input and output variables of a controllable system. After maximum likelihood learning within these graphs, the sum-product algorithm can be used to infer output values consistent with the current goal during live control. A hierarchical architecture is used to model multiple temporal and spatial scales in order to carry out long sequences of actions without the need to plan the entire action at once. All information is stored exclusively at the nodes of the employed graphical models and both during learning and live control, all computation takes place locally.

We demonstrate how factor graphs modelling instantaneous dynamics can be used to control a simple compliant robotic system and how complex robot kinematics can be factorized into products of simple functions, which can form the core of a basic control algorithm. Furthermore, we describe the concept of hierarchical navigation networks (HNN), which form a new class of graphical model topologies that are based on a hierarchical decomposition of state space. In these models, instantaneous dynamics at different

spatial and temporal scales are modelled at different hierarchy levels. Given a distant goal, higher levels of this hierarchy impose subgoals that are closer to the current position onto lower levels, which eventually allows the determination of an immediate action at the lowest level. Detailed inference algorithms for these models are found, as well as local learning rules for the model parameters.

All models that we employ are self-contained - they learn solely from experience and without the necessity for any prior information or reward signal. By relying on a distributed computation scheme in which the computational units have access only to local messages, our approach to motor control and planning problems shares important features with biological nervous systems. The HNNs we develop introduce a new concept in the modelling of temporal processes and hierarchical plans using graphical models. We believe that the three approaches to motor control and planning presented in this thesis form an important step towards self-contained distributed computational architectures that can learn to control robotic systems on their own.

Zusammenfassung

Nervensysteme von biologischen Organismen und im speziellen das menschliche Gehirn sind in der Lage, eine Vielzahl von natürlichen Stimuli zu verarbeiten. Dabei übertreffen sie alle derzeit verfügbaren Computeralgorithmen bei der Lösung von Aufgaben, die eine Interaktion mit der realen Welt voraussetzen, wie zum Beispiel der Kontrolle von motorischen Systemen. Erstaunlicherweise sind sie in der Lage, ein internes Verständnis von beliebigen steuerbaren Systemen einzig durch die Interaktion mit diesen aufzubauen, was sie von den performantesten Steueralgorithmen unterscheidet. Gleichzeitig ist offensichtlich, dass Berechnungen im Gehirn auf eine Weise durchgeführt werden, die sich fundamental von der Rechenart auf moderner Computerhardware unterscheidet. Anstelle einer oder weniger zentraler Steuereinheiten ist das Gehirn aus vielen Milliarden winziger Recheneinheiten aufgebaut. Informationen werden in diesen Recheneinheiten gespeichert indem die Effektivität der Kommunikationskanäle moduliert wird. Trotz dieses scheinbaren Nachteiles, nur lokale Berechnungen durchführen zu können und keine globale Steuereinheit zu besitzen, sind Nervensysteme in der Lage, eine grosse Vielfalt an Bewegungen von zu planen, welche sich über verschiendene Zeiträume erstrecken können.

In dieser Arbeit untersuchen wir Algorithmen, welche von biologischen Nervensystemen inspiriert sind und allein durch Analyse von beobachteten Zuständen eines zuvor unspezifizierten steuerbaren Systems lernen, dieses zu kontrollieren. Diese Algorithmen sind darauf ausgelegt, auf verteilten Rechenarchitekturen zu laufen, welche aus vielen kleinen und lokal operierenden Einheiten aufgebaut sind. Weiterhin betrachten wir Möglichkeiten, auf die sich lange Sequenzen von Bewegungen auf solchen Rechenarchitekturen planen lassen.

Wir verwenden Graph-basierte Wahrscheinlichkeitsmodelle, die Variablen mit diskreten Zuständen enthalten, um den Zusammenhang zwischen Eingabe- und Ausgabevariablen eines steuerbaren Systems abzubilden. Diese Modelle werden zunächst nach dem Verfahren der grössten Wahrscheinlichkeit (engl. 'maximum likelihood') trainiert. Ein einfacher Algorithmus basierend auf dem Summenprodukt zweier Vektoren kann dann verwendet werden, um Ausgabewerte zu berechnen, wenn nur die Zustände der Eingabevariablen bekannt sind, wie zum Beispiel bei der Steuerung des Systems. Eine hierarchis-

che Modellarchitektur wird eingesetzt, um verschieden grosse zeitliche und räumliche Grössenordnungen zu repräsentieren und um Bewegungen von unterschiedlicher Länge zu planen. Alle eingesetzen Rechenmodellen bestehen aus vielen kleinen Recheneinheiten, welche jeweils nur auf lokal gespeicherte Informationen zugreifen können. Sowohl während der Lernphase als auch der Anwendung dieser Modelle findet die Informationsverarbeitung ausschliesslich in diesen Recheneinheiten statt.

Wir führen vor, wie Faktor Graphen, welche die unmittelbare Bewegungsdynamik eines Systems abbilden, zur Steuerung eines einfachen nachgiebigen robotischen Systems eingesetzt werden können und wie die komplexe Kinematik eines Roboterarms als Produkt einfacher Funktionen dargestellt werden kann, um dann den Kern eines simplen Steueralgorithmus' zu bilden. Darüber hinaus beschreiben wir eine neue Klasse von Graph-basierten Rechenmodellen, welche auf einer hierarchischen Aufteilung des Raumes aufbauen. In diesen hierarchischen Navigations-Netzwerken (HNN) werden die unmittelbare Veränderungen eines Systems und ihre Ursachen in unterschiedlichen räumlichen und zeitlichen Massstäben betrachtet. Soll eine Veränderung herbeigeführt werden, welche viele Einzelaktionen benötigt, so ermitteln die höheren Hierarchieebenen in diesen Netzwerken Zwischenziele, welche den unteren Hierarchieebenen vermitteln werden. Auf der untersten Hierarchieebene kann auf diese Weise eine unmittelbare Einzelaktion bestimmt werden, welche im Sinne der erwünschten Veränderung ist. Wir erklären die genauen Regeln, nach denen in diesen Netzwerken Schlussfolgerungen getroffen werden und beschreiben lokal funktionierende Lernregeln, die zum Aufbau dieser hierarchischen Netzwerke führen.

Alle Kontrollalgorithmen, welche wir in dieser Arbeit verwenden, sind in sich geschlossen und benötigen keine Vorabinformationen über die Struktur des zu kontrollierenden Systems. Die Parameter dieser Algorithmen werden ausschliesslich durch Analyse von Beobachtungsdaten trainiert. Ferner können alle Berechnungen in diesen Algorithmen verteilt auf viele kleine Recheneinheiten durchgeführt werden. In diesen Punkten ähneln die von uns vorgestellten Methoden den Rechenabläufen in biologischen Nervensystemen. Die HNNs, welche wir entwickelt haben, stellen ein neues Konzept dar, wie zeitliche und hierarchische Zusammenhänge in verteilten Rechenstrukturen dargestellt werden können. Zusammenfassend sind wir überzeugt davon, dass die in dieser Arbeit vorgestellten Ansätze zur Steuerung von motorischen Systemen und zur Bewegungsplanung eine wichten Schritt darstellen auf dem Weg zu vollständig in sich geschlossenen, auf parallelen Architekturen durchführbaren Rechenalgorithmen, welche von sich aus lernen können, robotische Systeme zu steuern.

Contents

Chapter 1

Introduction

1.1 Purpose of this thesis

From a very abstract point of view, brains can be reduced to being powerfull and almost universal inference machines. Millions of sensors provide a continuous stream of high-dimensional input to the brain, which itself generates an equally complex stream of motor output signals. In this vast amount of data, the brain discovers functional dependencies and thereby gains an understanding of its environment. This knowledge can be interpreted as an internal model of the world, which enables the brain to infer properties of the world whenever direct observations about these properties are unavailable. Such a non-observable property could be a prediction about a future state of the environment. For example, we can predict the future position of a flying ball after seeing its recent trajectory.

Another type of non-observable data brains need to infer is required in motor control: In order to cause a transition from an observed body posture to a desired future posture, the brain needs to infer sequences of motor output signals, which are of course non-observable before execution.

Amazingly, the brain seems to be able to learn an internal model of the world solely from experience. Concerning motor control abilities for example, a newborn human struggling with its limbs appears to have very little conception about the effects of its motor output signals. By executing thousands of seemingly random movements, its brain learns to correlate neural output signals with limb movements and from these statistics gains basic control over its skeletal muscles within short time. As the newborn's control abilities improve, movements will become more directed and less noisy, which allows the brain to refine and specialize its internal model of the motor system. The self-containment of this process of acquiring new abilities is impressive: random movements lead to a basic model, which allows to generate better movements, leading to a better model and so on.

While it is an open question how much prior information about the motor system of

its own body is available to the brain at birth, it seems like brains can learn arbitrary tasks purely by monitoring the relation between action and effect. Interestingly, the brain learns to control its body at a large number of various timescales, with motions directed towards goals that are reachable within tenths of a second or within up to several days.

This thesis aims at the development of self-contained algorithms for control that learn to control robotic systems through a self-developed internal model, without having any prior information about the structure of the system. We use graphical models like factor graphs in order to learn the statistics between input and output data obtained from robotic systems and employ these models to infer motor control outputs in control tasks. In contrast with methods from reinforcement learning, these models are constructed solely from data points collected while interacting with the system and do not require a teaching signal, such as a reward signal.

We demonstrate that both the instantaneous dynamics of a robotic system and the kinematics of a complex robot can be modelled as a factorized function and outline algorithms for control that are based on this representation. In order to solve control tasks that require the execution of long sequences of actions, we introduce a new concept of how temporal processes can be modelled as a graphical model. This approach uses a hierarchical decomposition of a state space to partition a complex control task into simple tasks at different spatial and temporal scales.

The models we employ are neurally inspired in terms of their underlying computational paradigms: All information is processed locally at the nodes of a distributed computational network. Both learning and inference calculations result as a coordinated action of these small computational units. In addition, the presented algorithms are of non-procedural nature: Information is communicated between the computational units, but is never stored as intermediate results. Instead, a single communication between all nodes is sufficient to infer actions. Even during the execution of long action sequences, immediate actions are computed with a single communication between all nodes.

1.2 Human control compared to control algorithms

For several decades already, there have been tremendous efforts to develop control algorithms for robotic systems that would mimic the abilities of brains. Despite these efforts, human control over the motor system still outclasses any available algorithms for robot control as of today. While modern robotics can provide systems that outperform humans in terms of precision, speed and strength, no one has yet been able to demonstrate a robot that can compete with humans in terms of adaptability to a dynamic environment. This superiority of human movement control results from three important properties:

First, humans are able to dynamically respond to unpredictable obstacles and thereby avoid collisions while still achieving their goal under the unexpected constraints, which is something robot control is struggling to do. Modern approaches allow to avoid obstacles, but the spatial position of such obstacles needs to be defined precisely for example in terms of smooth cost function, which requires cumbersome parameter tuning. In addition, a full trajectory around the obstable needs to be precomputed before the initiation of movement, which is a time-consuming procedure [Tou09].

Second, human motions are passively compliant. The forces that humans exert with their muscles in order to perform a movement normally matches precisely the required strength for executing this movement. As a consequence, our motions can be modified by external forces. In comparison, most robotic systems are stiff and use servo controllers in order to precisely follow a defined trajectory. This stiffness, which seems like an advantage at first sight, can be problematic if the robot should interact with humans in dynamic environments. Particularly when combined with an inability to avoid collisions, this non-compliance of a robot can cause damages of the robotic system, the environment, or injuries of interacting humans. Typical factory robots must therefore be turned off before a human can safely approach it.

Third, humans are able to quickly adapt to changes of the motor system. As such, we never experience a handicap caused by slow changes as caused by body or muscle growth. Sudden changes caused for example by injuries or by bulky clothing are also adapted within short time. In contrast, robotic control algorithms need to be designed and adapted for a specific robotic system.

One of the major challenges modern robotic control algorithms are facing is to control a humanoid robot. Approaches to this problem are usually bound to a precise description of the robot in terms of its topology and its actuators and cannot be learned from experience [KKK+03, HKK+07]. In addition, these approaches are designed for the control of almost deterministic systems with little noise in sensor data and action execution, which restricts their application to precisely controlled and non-compliant systems. Probabilistic approaches to motor control naturally extend to noisy systems. However, they also require a precise parameterization of the robot to control [Tou09]. Modern approaches involving reinforcement learning allow adaption to modified systems, but nevertheless require the definition of motion primitives, which in turn are tied to a specific robotic system [SBTS10, PVS03, TBS10, KP09].

1.2.1 Human abilities to plan sequences of actions

In order to solve complex tasks, we often need to exert long sequences of coordinated short actions. As an example, imagine a person that is leaving from home and heading to

a local airport in order to catch a plane. Solving this task requires the person to execute thousands of short muscle actions. While the complete sequence of actions is necessary in order to reach the defined goal, it is impractical to plan the complete trajectory of all body limbs in advance.

Interestingly, we are able to outline a path to the goal within a few seconds, which allows us to almost immediately initiate a movement. Instead of planning a detailed trajectory for each of our joints and predicting the precise position of our body at all timesteps in the future, we rather follow more abstract goals and subgoals. In the example, the abstract goal the person is pursueing could be the state of being at the airport. Even if the final goal would be defined in terms of a precise position and body pose, these details are irrelevant for reaching it as long as the distance to the goal is high.

An abstract formulation of the goal will in most cases not on its own be sufficient to determine suitable muscle actions, but we can easily formulate less abstract subgoals that need to be reached in order to get to the final goal. These, in turn, can guide the formulation of even less abstract subgoals and so on. In the example, a possible series of such subgoals might be 'reach the bus stop', 'leave the house', 'leave the room', 'stand up' and 'shift weight forwards'. This subgoal is the least abstract and the only one that will have an effect on the precise muscle actions the person needs to execute immediately. Future muscle actions can be computed as soon as the state of the body has changed. In this way, the complete sequence of actions unfolds only during the process of solving the task.

1.3 Towards neurally inspired control algorithms

1.3.1 Computational properties of brains

Obviously, brains are able to solve classes of motor control tasks, which today's algorithms struggle to solve. A lot of research is invested into improving existing methods from control theory or reinforcement learning in order to get closer to the brain's performance. Instead of following the existing approaches to motor control and attempting to gradually improve today's algorithms, we choose to follow a different path towards motor control solutions, in the hope to make a step into a new direction that might eventually lead to a new class of control algorithms. Since brains are the only known computational devices which can provide robust control in interactive environments, we followed several computational paradigms in the design of our approaches, which were inspired by the computational properties of brains.

The human brain is composed of approximately 10^{11} neurons, each of which is connected to about 10^4 neurons. Each neuron in a brain forms a distinct computational

unit and can only communicate with the connected units. All information it can access is stored locally in its activation state or in the synaptic connection strengths to other neurons or is transmitted to it through spiking signals from connected neurons. Remarkable about the organization of the brain and in particular the mammalian cortex is its uniformity, which is both expressed in terms of neuron densities [HW74, RHP80] and neuron connectivities [DM04, DMW89].

1.3.2 Factor graphs

Interestingly, graphical models such as factor graphs share several qualitative properties with brains: They are composed of many computational nodes which can communicate with each other through communication channels. Information is stored locally at the nodes and computations at each node are performed based on this locally available information and messages that are received from other nodes. Moreover, these local parameters can also be trained from experience by locally evaluating incoming messages [Rol06]. Besides these computational similarities, factor graphs and brains share the property of having a uniform structure. Algorithms like the belief propagation algorithm rely entirely on the repetitive application of the same and simple operations.

One might argue that the computational power of nodes in a graphical model is much higher than that of single neurons. However, ensembles of neurons could be regarded as the neuronal counterpart of these nodes. This idea was applied in several recent studies, that inspect how calculations on a factor graph could be implemented with simulated neurons [SMD09, Rao04, Rao05, LU09, DIPR07, MBLP06].

Factor graphs can be regarded as powerfull machines for the approximation of high-dimensional functions [KFL01]. By factorizing such high-dimensional functions into products of lower-dimensional functions, they allow a significant reduction of the number of parameters. At the same time, interesting properties of the global function can be computed efficiently using Pearl's belief propagation algorithm [Pea88]. One such property are the marginals of the global function with respect to one parameter. These marginals are of particular interest if the global function is a probability distribution, because the marginals will be equal to the conditional probability distribution for one variable.

The possibility to distribute computations in a factor graph to many small computational units is of course interesting with regard to standard multi-core hardware architectures that are available today, but also makes factor graphs an interesting subject for implementation on massively parallel architectures, like the SpiNNaker architecture [KLP+08].

1.3.3 Control algorithms based on graphical models

Motivated by the apparent similarities between the computational properties of brains and of factor graph, we developed control algorithms based on graphical models. In our approach, a graphical model is used to create an internal representation of a robotic system, which learns to relate input and output variables of the system through experience. In two different approaches we present, the internal model is trained to represent either the instantaneous dynamics or the kinematics of a robot. Notably, these internal models are represented in terms of sums and products of small functions. During live control, motor commands are inferred using the belief propagation algorithm on these internal models.

The control algorithms we present are self-contained: they learn behaviour solely from observations at several data channels and have no prior information about structure within the data. After learning, complex functions like the inverse kinematics of a robot arm are computed by simple multiplications of low-dimensional functions.

1.3.4 Representing multiple timescales in a graphical model

The control of robotic systems often involves motor planning, which requires the identification of usefull sequences of motor commands. This problem is related to navigation tasks, where sequences of actions are required to navigate from a starting point to a destination. We therefore chose navigation problems in order to investigate how temporal sequences could be generated using graphical models.

We present a hierarchical approach to navigation that is inspired by the way in which humans seem to plan a path between a source and a destination. As described in the last section, we do not require a precise definition of a distant goal in order to execute immediate actions. Instead, it is apparently sufficient to define distant goals at an abstract level, which allows to deduce less abstract subgoals. Our approach uses an abstraction hierarchy, which arises as a result from learning through experience. The relation between the current state of the agent and a given goal is evaluated at different abstraction levels. As a result of this evaluation, subgoals at lower abstraction levels are chosen, which ultimately guide the immediate behaviour of the agent.

We show how to implement the described scheme of computation in terms of a *hierarchical navigation network* (HNN), which is related to graphical models and as such is composed of a network of many small computational nodes. Each of these nodes belongs to an abstraction level and is connected to a small number of other nodes at the same or an adjacent abstraction level. Nodes can communicate through these connections by message passing, where each message consists of vectors of real numbers. The messages

sent to a node compose the only information that is accessible to that node. While we assume that the network topology is predefined by the programmer, each node can learn its own message-producing behaviour from its experience of the messages it receives. The nodes are organized in a hierarchical structure, which reflect the layering of abstraction levels mentioned above. We describe the topology of HNNs and show how these can be used to solve navigation tasks by applying them to maze navigation tasks. The nodes of the network use learning rules that rely solely on local messages available to the nodes.

HNNs have several interesting properties: First, they demonstrate how a simple on-line learning rule can lead to the emergence of a hierarchical decomposition of space. Second, the self-emerging hierarchical decomposition of space allows the network to quickly converge when given a source and a destination and to select an appropriate basic action at every point in the journey. While the generated paths are not necessarily optimal in terms of length, the resulting path lengths get close to optimal as the system learns. Third and most importantly, HNNs represent an alternative way to model temporal processes with graphical models. Processing across different time scales arises naturally as a consequence of the hierarchical spatial decomposition.

Chapter 2

A software to allow interaction with factor graphs

2.1 Motivation

In order to obtain the ability to use a factor graph for controlling a systems of any kind, three elements need to be implemented: First, the factor graph itself, together with the possibilities to run message passing algorithms on it and to adjust its internal parameters. Second, a controllable system, such as a robotic system. Third, a bridging interface between the first two components. All three parts could be implemented in arbitrary form, for example as a hardware or a software solution.

As of now, there exists no hardware implementation of a factor graph, which makes a software solution the prefered choice for the first component if the goal is rapid productivity. If the type of robotic system that should be connected to such an implementation of a factor graph could change, it is desirable to design the interface as flexible as possible in order to allow it to interact with both real and software-simulated robotic systems as well as more abstract controllable agent-based environments.

Before this thesis project, there existed no such flexible interface that would allow to easily connect different types of data sources such as robotic systems to factor graphs of arbitrary topology in order to explore solutions for various control problems. Therefore, we decided to implement a software library that would allow to setup factor graphs, to efficiently run message passing algorithms on these graphs and to link nodes from these graphs to arbitrary data sources.

In order to develop and explore new concepts within the framework of factor graphs, it is of fundamental importance to truly understand both its capabilities and restrictions. While it is fairly easy to understand the mathematics underlying factor graphs and the belief propagation algorithm, it is often times already difficult to picture the effects a

single parameter change in one factor node can have on a message to an adjacent node. It is of course an even more complex task to imagine the effects of various learning rules applied to one factor node on whole series of messages arriving at distant nodes.

Gaining a natural understanding of any complex system can be greatly alleviated by a tool that allows to visualize its properties and to easily interact with it. This insight motivated us to integrate the required factor graph and interfacing software into a powerful visualization and editing framework.

2.2 Software characteristics

We developed a Factor-graph editing environment called *FGControl* which offers much more functionality beyond the main function of interactively creating and modifying factor graphs. It is designed as a modular multi-purpose framework that can easily be extended with any desired capabilities. A screenshot of a possible setup can be seen in figure 2.1.

The software provides its own serialization mechanism which allows to detect new classes and user-controlled instantiation of members of these classes. This allows devlopers to quickly extend the software with new modules, which are easily integrated into the existing framework. The properties of the new modules can be edited from within the software and can be stored in and restored from XML-files and a binary file format. New modules can define their own user interface and visualization, which will automatically become availabe in the interface. Section 2.4.3 guides through an example class that extends the software.

2.2.1 Factor graph functionality

Despite its design as a multi-purpose framework, the main function of FGControl is of course the interaction with graphical models and in particular factor graphs. Therefore, the center window in the graphical user interface is currently reserved for displaying such graphical models (see figure 2.1). Three aspects were guiding the implementation of the factor graph functionality.

High performance computation

Messages inside the factor graph are communicated through simple array structures and data sharing between the nodes reduces the amount of storage overhead. Tensor calculations at the factor nodes, which make up the major part of the computational complexity in a factor graph are highly optimized in low-level code. This code takes advantage of cache-coherency and as such aims to reduce the amount of memory address jumps.

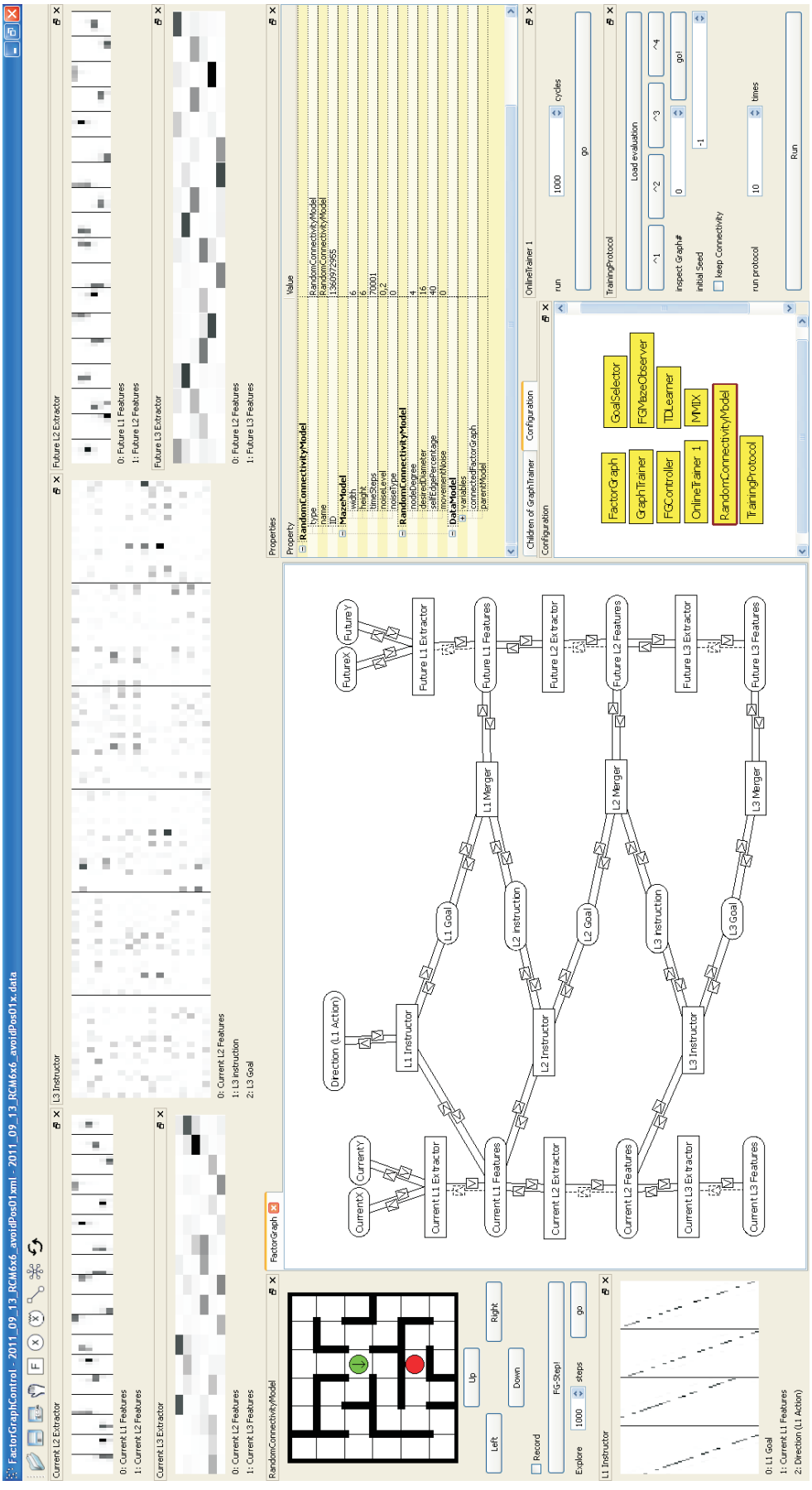

Figure 2.1: *A sample screenshot of FGControl. The center shows a factor graph which is connected to a 6x6 maze as a data source. Several dockable widgets shows the contents of some of the factor node contents, the data source and few other modules. The properties of one module are displayed in the properties widget.*

In addition, it avoids to recompute memory addresses by reducing the amount of array indexing.

Message passing in a factor graph is currently implemented in a single-threaded version only. This single-threaded version proved to be sufficient for all control setup that were tested in the course of the thesis project. Nevertheless, it would be desirable to extend this code to a multi-threaded or a distributed version.

Flexibility

FGControl allows to easily setup factor graphs of arbitrary topology. Factor nodes can have any desired dimensionality and variable nodes can have user-defined numbers of discrete states. Connections between nodes can be added at any time and have as little impact as possible. Similarly, the number of states for variable nodes can be adjusted at any time, with as little impact as possible.

As shown in section 2.3.1, messages in a factor graph can be inspected by single mouse clicks and the contents of factor nodes can be continually observed while they are being modified by factor node training algorithms.

Interfacing to data sources

Variable nodes in a factor graph can be *observed* variables. These variables are typically linked to some observation which originates from some sort of data source. The messages these observed variable nodes send to their neighbors should depend solely on this observation. In FGControl, variable nodes can interactively be linked to data source variables, which can be part of various observations. New data sources can easily be added to the software by reimplementing an abstract class for data sources.

2.2.2 Choice of programming language

FGControl is implemented in *C++* and makes heavy usage of the cross-platform application and user interface framework *Qt* [Nok11]. As such, the software can easily be compiled on several platforms (Linux, Windows, MacOSX). While being almost as portable as a *Java* application, this choice is advantageous in terms of computation speed, because the code runs without virtual machine and can be optimized for a specific architecture by the compiler.

Implementation in the *C++*-language also provided the advantage of simplifying communication with several application programming interfaces for various robotic systems. Libraries for communication with a compliant pendulum (see section 3.2.1) or a Katana-robot (see sections 3.7) were both available as *C++*-libraries. A software implementation

in *Java* or *Python* would have required additional interfacing work. Interfaces to native libraries also would have reduced the advantage of these two languages of being designed for cross-platform usage.

The usage of *Qt* as a application framework allows cross-platform compilation, but in addition provides the advantage of future software interfacing to the *Python*-language, because all *Qt*-objects can be used from within *Python* via *PyQt*. During this thesis project, it was successfully tested to interface objects from FGControl through *PythonSIP* and to use them from within a *Python*-script. This approach was however not pursued any further because it provided no additional benefit for this project.

2.3 Features and usage

2.3.1 Interface basics

The user interface of FGControl mainly consists of 4 elements: First, a configuration window displays the currently instantiated objects. Second, a center area that is reserved for factor graph manipulation and inspection. Third, a properties window allows to edit the properties of objects. Fourth, a user-selectable number of windows is used to visualize the contents of selected objects. All windows except the factor graph editing area are dockable windows, which allows to freely configure the interface.

Configuration window

The configuration windows displays items for all instantiated top-level objects (see 2.4). New objects can be instantiated by double-clicking, which causes a dialog to appear. This dialog offers a list of known classes and allows to create a new instance of any of these classes and to assign it a name. Instantiated abjects are then shown as a movable item in the configuration window. Selection of an object by left-mousebutton click causes its properties to be displayed in the properties window, where these can also be edited. Double-clicking an item calls an object-specific dialog. This dialog depends on the object implementation. An example for such a dialog is shown in figure 2.2, which allows to edit the properties of a factor node. Right-clicking an item displays a small menu. The menu entries are object-dependant, but two options are always available: First, a visualizing window for the respective object can be created, which will add a new dockable window to the user interface. Second, a window can be created that displays the child objects of the selected object. This window behaves similar to the configuration window. Mouse scrolling in the configuration windows allows to zoom in and out, which is beneficial for setups with many objects.

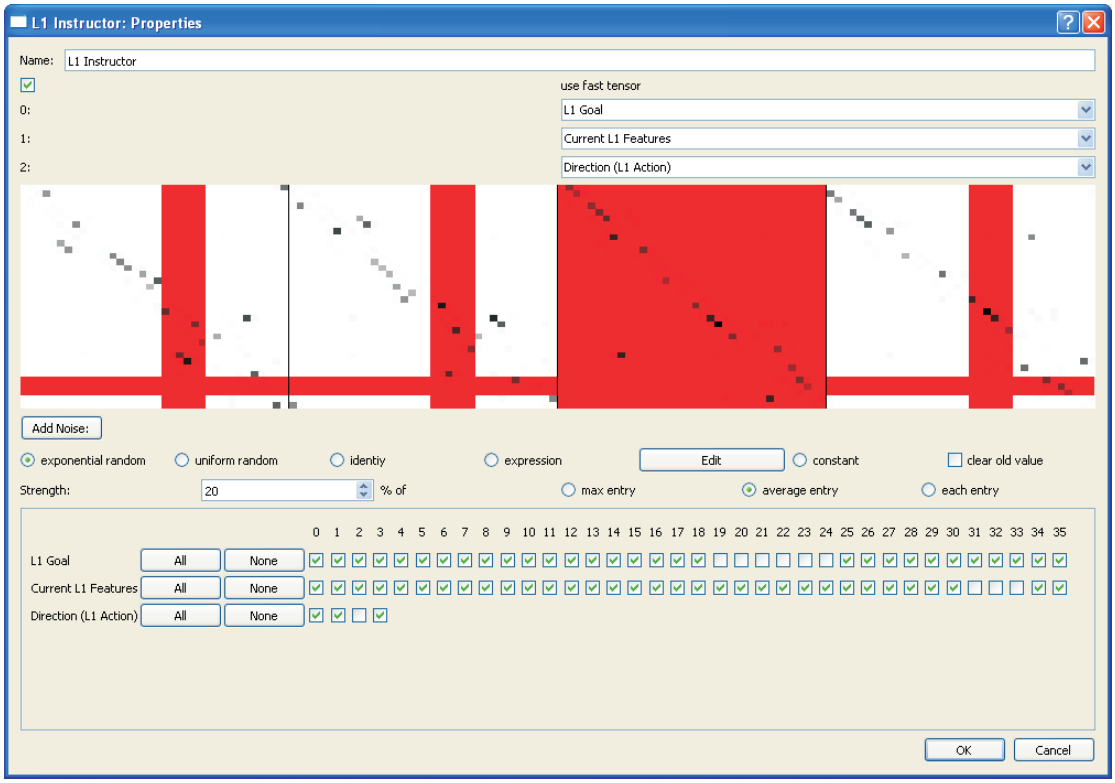

Figure 2.2: *A dialog for editing the contents of a factor node. Here, a three-dimensional factor node is shown. Grey values in the center area indicate the relative value of a factor node entry, with darker colors representing higher values. The combo-boxes above this area allow to change the order of connected variable nodes, which will affect the display.*

Factor graph editing

The factor graph editing area behaves very similar to the configuration window. As such, mouse buttons are assigned the same function in normal editing mode. However, users are only allowed to instantiate classes that are of type `FactorNode` or `VariableNode` or are childclasses of these. Factor nodes are displayed as rectangular items with sharp edges in the area, while variable node items have rounded edges. Connections between factor nodes and variable nodes are displayed as two lines connecting their respective items, one line for each message direction between the nodes. A small arrow box indicates the message direction. Right-dragging the mouse from these arrow boxes adds a small message display to the factor graph editing area, which will automatically display the message sent along this connection. An example setup with message inspection is shown in figure 2.3.

A small toolbar in the software admits to enable three more editing modes: One mode allows to add factor nodes with a single click and another to similarly add variable nodes. The third mode is used for connecting nodes in the network. If enabled, left-dragging the mouse from a variable node to a factor node or vice versa establishes a connection between these nodes. This operation automatically increase the dimensionality of the connected factor node. The contents of the factor node are expanded in such a way that the additional connection will have no immediate effect on the global function the current factor graph is representing.

In standard editing mode (activated by small 'hand'-icon), items as well as connections between nodes can be selected in the factor graph editing area. Both elements can be deleted by using the 'Delete'-key on the keyboard. Deletion of connections will cause a reduction of dimensionality in the connected nodes. Deleting nodes also causes the deletion of their connections.

Properties window

The properties windows shows the parameters of a selected object in either the factor graph editing area or the configuration window. The parameters displayed are the same that are saved to a XML-file when storing the state (see section 2.3.1). Some of these parameters such as numbers, boolean values or strings can be edited in the properties window.

Storing and restoring a setup

A setup in FGControl can be saved to disk by pressing the 'Save File'- or the 'Save File As'-Button ('Disc'-icons in the toolbar), the latter allowing to select the name of the desired output file. The program will ask for to files for saving, because contents of a

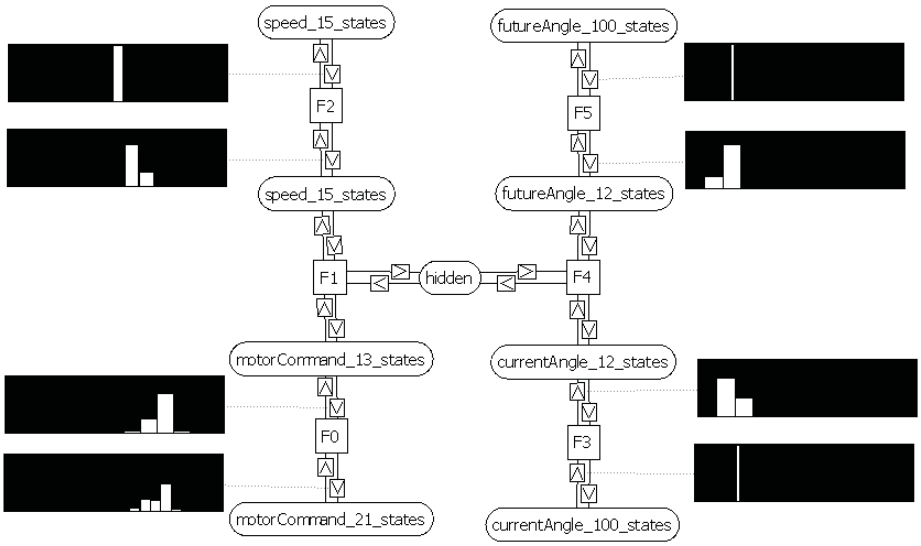

Figure 2.3: *Inspecting messages sent between nodes in a factor graph. The displayed graph was used to learn one-timestep dynamics of a simple pendulum (see section 3.4). Here, eight messsages are inspected, six of which are directed towards the inside of the graph and the remaining to pointing towards the observed variable node* `motorCommand_21_states`*. The user can inspect message by simply right-mouse dragging from the small arrow-boxes that indicate the message direction between two nodes.*

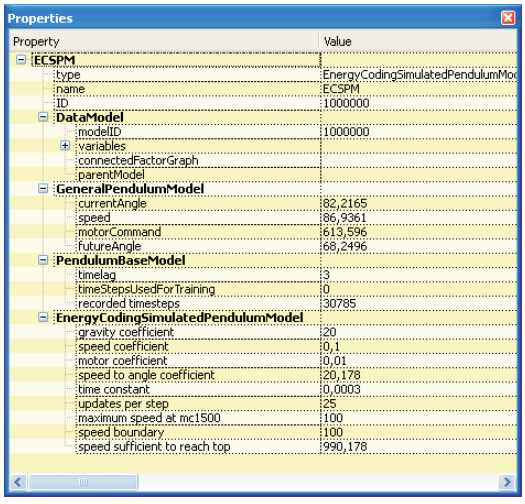

Figure 2.4: *The properties window displaying the parameters of an example object. The first line indicates the name of the object, the remaining bold lines indicate which parameters were inherited from which parent class.*

setup are stored in two separate files: An XML-file is used to store structural paramters of objects in the setup as well as the user interface setup. A binary '.data'-file is used to store other data associated with the objects, such as recorded data traces or factor node contents. As a rule of thumb, everything that would be to bulky to be edited in the properties window is stored in this binary file (see also section 2.4.2).

When opening a file (using the 'Folder'-icon), the user has the option to either load a XML-file or a '.data'-file. The former will basically restart FGControl with a new setup, while the latter will cause all objects in the current setup to attempt to load non-structural data from the binary file.

2.3.2 Factor node training

Parameters of factor nodes can be trained using many possible methods, such as gradient descent methods or an expectation maximization algorithm. In the course of this thesis, many different learning algorithms were applied. Particularly the development of HNNs required the devolopment of several specialized algorithms. Sometimes, several different algorithms need to be applied at the same time to different factor nodes.

FGControl provides a universal graph training module (class `GraphTrainer`) for training factor nodes with respect to observed data. It can be used to train a complete factor graph by adding multiple trainers for single factor nodes. Figure 2.5 shows several GUI elements associated with this module. A dialog allows to create new factor node trainers, which can easily be enabled or disabled for training. Parameters of the individual trainers can then be edited through the properties window.

During training, the graph training module will iterate over a selected range of data points the connected data source can provide. For each data point, all observed variable nodes in the network are fixed to their respective value. Messages are then passed in the network until convergence before the graph training module informs the activated factor node trainers that a new data point was presented.

2.4 Software architecture

2.4.1 Object hierarchy

As mentioned before, FGControl is designed as a multi-purpose framework that serves as a platform for a collection of many modules. New modules can easily created and integrated into the software. This modular design is enabled by a framework-specific serialization mechanism, which is described in the next section.

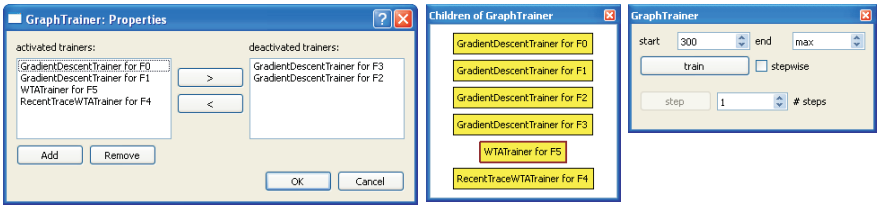

Figure 2.5: *Graphical user interface elements for graph training. Left: A dialog that allows to create new objects for training algorithms for selected factor nodes. Middle: the created node trainer objects appear as child object of the graph trainer object. Using a 'child inspector'-window, these node trainer objects can be selected, which allows their parameters to be edited in the properties-window. Right: The dockable window for a graph trainer object, which can be used to initiate the training process, either in continuous form or by training for a selected number of steps.*

Objects in FGControl are organized in a hierarchy. Each object can have one parent object and an arbitrary number of children. This hierarchical structure matches the modular design: A module usually consists of a collection of objects and it makes sense to bind these objects together, as they should be instantiated and restored together. In FGControl, objects loaded by another object will become its children and will be maintained by it. Accordingly, deletion of one object implies deletion of all its children. This allows to easily remove a complete module from memory. As an example, consider a factor graph, which contains several factor nodes and variable nodes. In FGControl, these nodes are considered the children of the factor graph. The nodes are tied to the factor graph because they lose their relevance without the existence of the factor graph.

All classes are children of one base object, which is a member of the main class `FGControl`. This base object holds together all loaded modules.

The organization as a tree is useful to express strong object dependencies as described above. However, it is often required that different modules can communicate with each other, although they are not tightly bound to each other. A data source needs to communicate to a factor graph, but deletion of the factor graph does not imply deletion of the data source and neither vice versa, as each object has a relevance without the other. In FGControl, modules can search for other modules along the tree structure and thereby establish communication links. These links can be stored together with other contents and will be re-established when restoring the state, if both involved objects are also restored.

2.4.2 Serialization in *C++*

The problem

Unlike more modern programming languages like *Java*, *C#* or *Python*, *C++* is lacking mechanisms for serialization or self-reflection of objects. This property can become a handicap when developing a software that should be able to save its current state and should later be able to restore the saved state.

In order to understand this problem, consider a simple example: Let a software contain a list of objects of type `A` that can change during execution. In order to store its current state, it needs to save the state of all these objects, for example by first storing the number of objects in the list and then successively calling a `save`-method of `A` for all objects in the list. As long as the object type in the list is static, the state of the software can easily be recovered by creating the correct number of instances of `A` and then calling a `load`-function for each of these.

If the list contains instances of a childclass `B` of `A`, storing the state remains straightforward: `B` should reimplement the function `save`, which will allow to correctly store the state of the list. However, recovering the state of the software has become much more complicated: The program needs to know whether it should create instances of `A` or `B`. Even if the stored state contains the name of the class, the program still needs to know how to create an instance of this class. While *Java* for example provides mechanisms for precisely this situation, *C++* is lacking any equivalent.

One parent class for all dynamically loaded objects

One of the design goals for our software framework was that it would allow interactive addition of objects of arbitrary type and would be able to store and restore its state, independent of the currently loaded objects. In addition, integration of new classes should be straightforward and also possible dynamically by loading libraries. As such, the software needed to remain flexible with respect to manageable object types and allow the integration of classes that were not known at compilation time.

In order to solve the problems described in the last section, we chose to make all dynamically loaded classes childclasses of one base class, `FWSerializable`. In the following, we will denote these childclasses by the term *serializable class*. By creating a childclass of the abstract creator class `FWSerializableCreator` and registering this class at a central factory class (`FWSerializableFactory`), new classes become visible to the software framework and can be created by the user and be stored to and restored from data streams. A creator class can automatically be created and registered by calling the macro `REGISTER_SERIALIZABLE`, which will be demonstrated in section 2.4.3.

The class `FWSerializable` provides a function `getType`, which needs to be reimplemented by each serializable class. While saving the state, the program stores the types of all instances of serializable classes by calling this function. When restoring the state, the central factory class is queried for a creator class that allows to instantiate a class of requested type.

Serialization of object properties

Each serializable class can define the contents that need to be stored and restored by reimplementing the functions `serializeParameters` and `serializeData`. The difference between these is explained in table 2.1. Both functions are given an instance of an abstract archive class as parameter. These abstract archives provide overloaded `synchronize`-functions that allow to synchronize various data types. An archive can either store or restore variable contents, depending on its implementation.

This constructions bears two advantages: First, the programmer only needs to define the synchronization behaviour once, instead of defining both a `load`- and a `save`- function, which need to be consistent with each other. Second, the serializable object can be stored in and restored from arbitrary data structures. All that needs to be done to synchronize the state with a new data structure is to implement new archive classes. We took advantage of this construction in the current version: Structural properties of objects are typically stored in a XML-file but at the same time, they can also be edited in a properties-widget at runtime. In both cases, the same serialization-functions are called, but different archive classes manage the in- and output. Figure 2.6 shows an example for this process: Here, a simple class `Foo` is synchronized with two different data structures, which are both used in the software.

Dynamically loaded objects can contain pointers to other objects, which also need to be stored and restored. Of course, it makes little sense to store the value of these pointers, as the physical address of the linked objects will have changed after restoring the state. In FGControl, each dynamically loaded object is assigned a unique ID, which is stored as a property of the object and remains constant across different program sessions. This ID is also used to reestablish links between objects: In order to store its state, an object can create instances of `FWSerializableReference` for all pointers to other objects by calling the function `createReference` on these objects. These reference objects will store all information necessary to reidentify the connected object after restoring the software state.

This method allows to identify connected objects, but at the time an object is restored, a linked object might not be instantiated yet. To overcome this problem, child classes of `FWSerializable` can reimplement the function `establishConnections`, which will be

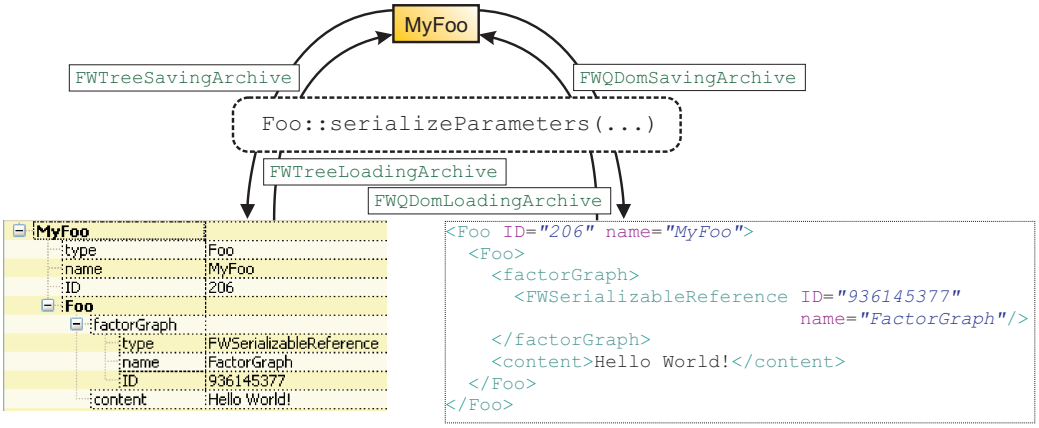

Figure 2.6: *Object parameter serialization in different contexts. Depending on the type of archive that is passed to the function* `serializeParameters`*, data is stored in different formats. Shown here is the result of storing an imaginary class* `Foo` *either to the properties window via* `FWTreeSavingArchive` *or an XML-file through* `FWQDOMSavingArchive`*. The contents of the object can be recovered from these data containers by using a different archive, in this case* `FWTreeLoadingArchive` *or* `FWQDomLoadingArchive`*. Data can be stored in any other format by implementing new archive classes. Source code for the function* `serializeParameters` *can be found in listing 2.2.*

Function name	Effects
`getType`	Returns the type of the class as a `QString`.
`serializeParameters`	Stores and restores structural properties of an object, such as dimensions of the object or links to other objects. Variables that are synchronized inside this function are typically stored in a XML-file and can be modified at runtime through the properties-tab.
`serializeData`	Stores and restores data contained in an object, such as recorded data traces or factor node contents. Variables are typically synchronized with binary data files.
`establishConnections`	This function is called after all objects have been created and should be used to re-establish links to other dynamic objects.

Table 2.1: *Functions of* `FWSerializable` *that should be reimplemented by child classes*

Function name	Return type	corresponding GUI element
`createVisualizer`	`SerializableVisualizer`	A visualization widget that will typically be embedded in a dockable window and might be a permanent part of the user interface. Can be used as a monitor to follow parameter changes of the object (such as factor node contents).
`createDialog`	`SerializableDialog`	A dialog window that will be shown as a modal window after the user double-clicked on the object's item.
`createItem`	`ConnectorClassItem`	A small icon representing the object, for example in the configuration window.

Table 2.2: *Functions of `FWConnectorClass` that can be reimplemented to define the object's behaviour in the graphical user interface.*

called after all objects have been loaded from a file.

2.4.3 Modular extensibility

User-instantiatable classes

The serialization mechanism described in section 2.4.2 allows it to easily extend FGControl. New child classes of `FWSerializable` can automatically be stored and restored by the program if the classes properly registered themselves at the central factory class. In addition, parameters of these new classes can easily be made accessible to the user by using the serialization functions described in section 2.4.2.

However, in order to allow dynamic instantiation of a new class by the user, new modules should be child classes of `FWConnectorClass`. Only classes of this type can be instantiated by double-clicking in the configuration window. `FWConnectorClass` defines three functions that can be reimplemented to define the user interface representation of an object; these are explained in table 2.2.

A simple example

This section provides an example of a minimal class `Foo` that allows to be interactively instantiated. The complete class declaration and source code are shown is listings 2.1 and 2.2.

The class `Foo` contains two parameters: `content`, which is of string-type and `factorGraph`, which links to an instance of `FactorGraph`. Both are synchronized in the function

```
1 #include <core/Framework/ExtendedConnectorClass.h>
2 #include <core/FactorGraph/FactorGraph.h>
3
4 class Foo: public ExtendedConnectorClass {
5 public:
6    FactorGraph * factorGraph;
7    QString content;
8    Foo(const QString& name = QString(), FWSerializable * parent = 0);
9    virtual ~Foo();
10   virtual QString getType() const;
11   virtual SerializableVisualizer * createVisualizer(
12     QWidget * parentWidget = 0);
13 protected:
14   virtual SerializableDialog * createDialog(
15     QWidget * parentWidget = 0);
16   virtual void serializeParameters(
17     FWParameterSynchronizationInterface &archive);
18 };
```

Listing 2.1: *Class declaration for a new dynamically loadable class* `Foo`.

`serializeParameters` (lines 30 and 31 in listing 2.2). Notice that before these variables are synchronized, the parent's implementation of the function is called in order to first synchronize the parameters of all parent classes in the hierarchy. For better style and for avoiding variable naming collisions, a new section is opened before synchronizing the parameters.

`Foo` is a childclass of `ExtendedConnectorClass`, which simplifies the maintainance of links to other objects. The link to an object of type `FactorGraph` is declared in line 4 in the constructor. Declaring this link has several effects: First, the object's dialog will automatically contain a drop-down box for establishing a new connection. Second, the internal pointer `factorGraph` will be automatically be set to null as soon as the linked object is deleted. Third, serialization is simplified and requires a single line (line 24). The connection to the object will automatically be re-established after the software state was restored.

The call of the macro `REGISTER_SERIALIZABLE` in line 4 makes sure that `Foo` will be integrated properly in the software framework, which takes as parameters the name of the class that should be registered and the name of its superclass. Internally, this macro creates two helper classes: The first one is a childclass of `FWSerializableCreator`, which provides a function `create` that can be called to create an instance of `Foo`. The second one will register the creator class at a central factory class in its constructor. The macro also declares an instance of this second class, which will cause the execution of this constructor. Taken together, the creator class will automatically be registered as soon as the code is loaded and this will allow the software to dynamically create new

```
1 #include "Foo.h"
2 #include <coreincludes.h>
3
4 REGISTER_SERIALIZABLE(Foo, ExtendedConnectorClass)
5
6 Foo::Foo(const QString& name, FWSerializable * parent) :
7    ExtendedConnectorClass(name, parent) {
8    factorGraph = 0;
9    declareConnectedSerializable("factorGraph", "FactorGraph",
10      (FWSerializable**)&factorGraph, "connected␣FactorGraph:");
11   content = QString("Hello␣World!");
12 }
13
14 QString Foo::getType() const {
15   return QString("Foo");
16 }
17
18 SerializableVisualizer * Foo::createVisualizer(QWidget * parentWidget) {
19   return new FooVisualizer(this, parentWidget);
20 }
21
22 SerializableDialog * Foo::createDialog(QWidget * parentWidget) {
23   return new FooDialog(this, parentWidget);
24 }
25
26 void Foo::serializeParameters(
27     FWParameterSynchronizationInterface &archive) {
28   ExtendedConnectorClass::serializeParameters(archive);
29   if (archive.openSection("Foo") == ARCHIVE_OK) {
30      synchronizeConnectedSerializable(archive,
31        (FWSerializable**)&factorGraph);
32      archive.synchronize(QString("content"),content);
33      archive.closeSection();
34   }
35 }
36
37 Foo::~Foo() {}
```

Listing 2.2: *Source code for a new dynamically loadable class* `Foo`.

```
1  class FooVisualizer : public BasicSerializableVisualizer {
2     QLabel * label;
3  public:
4     FooVisualizer(Foo * foo, QWidget * parentWidget);
5     virtual ~FooVisualizer();
6     virtual void serializableChanged(FWSerializable * serializable);
7  };
8
9  FooVisualizer::FooVisualizer(Foo * foo, QWidget * parentWidget)
10    : BasicSerializableVisualizer(foo, parentWidget) {
11    label = new QLabel(foo?foo->content:"");
12    this->getLayout()->addWidget(label);
13 }
14
15 FooVisualizer::~FooVisualizer() { }
16
17 void FooVisualizer::serializableChanged(FWSerializable * serializable) {
18    Foo * foo = dynamic_cast<Foo*>(getConnectedSerializable());
19    if (serializable == foo && foo)
20       label->setText(foo->content);
21 }
```

Listing 2.3: *Example code for a basic visualization class for* `Foo`*. The resulting user user interface element is shown in figure 2.7.*

instances of `Foo`. Necessary conditions for the macro to work are that `Foo` is a subclass of `FWSerializable` and that is declares a constructor with the same footprint as shown in this example. Abstract classes that to not allow instantiation should use the similar macro `REGISTER_ABSTRACT_SERIALIZABLE`, which is necessary for the framework to maintain a class hierarchy.

`Foo` defines its own user inferface by reimplementing `createVisualizer` and `createDialog`. These functions simply return new instances of `FooVisualizer` and `FooDialog`, respectively (lines 14 and 18). Listings 2.3 and 2.4 show the code for these two user interface elements.

`FooVisualizer` (listing 2.3) defines a simple visualization class for the class foo, which can be used to permanently monitor the state of the contents of `Foo`. As `Foo` is not very sophisticated, a single label is sufficient to visualize its state (lines 11-12). Note that `FooVisualizer` reimplements the method `serializableChanged`, which is called whenever the contents of a class changed. Here, the visualization is updated by changing the displayed text (line 20).

The dialog class `FooDialog` extends the class `ExtendedConnectorClassDialog` (listing 2.4). As such, the link to an instance of `FactorGraph` will automatically be maintained by the graphical user interface elements of the parent class. Hence, the function `addControls` is only used to add a simple line editor, which will be used to edit the string

```
1 class FooDialog : public ExtendedConnectorClassDialog {
2    QLineEdit * lineEdit;
3 public:
4    FooDialog(Foo * foo, QWidget * parentWidget);
5    virtual ~FooDialog();
6 protected:
7    virtual void addControls(QBoxLayout * layout);
8    virtual bool evaluateControls();
9 };
10
11 FooDialog::FooDialog(Foo * foo, QWidget * parentWidget)
12    : ExtendedConnectorClassDialog(foo, parentWidget) {}
13
14 FooDialog::~FooDialog() {}
15
16 void FooDialog::addControls(QBoxLayout * layout) {
17    ExtendedConnectorClassDialog::addControls(layout);
18    Foo * foo = dynamic_cast<Foo*>(getSerializable());
19    lineEdit = new QLineEdit(foo?foo->content:"");
20    layout->addWidget(lineEdit);
21 }
22
23 bool FooDialog::evaluateControls() {
24    Foo * foo = dynamic_cast<Foo*>(getSerializable());
25    if (foo) {
26       foo->content = lineEdit->text();
27       foo->contentsChanged();
28    }
29    return ExtendedConnectorClassDialog::evaluateControls();
30 }
```

Listing 2.4: *A simple dialog class for a new module* `Foo`*. The resulting user dialog is shown in figure 2.7.*

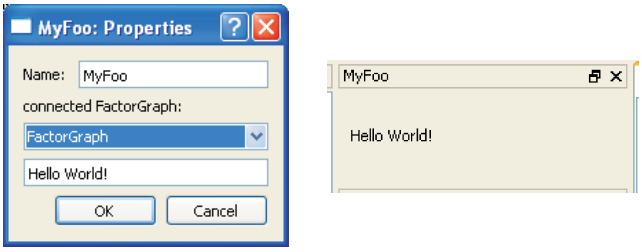

Figure 2.7: *Graphical user interface of* `Foo`*. Shown here are user interface elements corresponding to the two classes* `FooDialog` *(left, see listing 2.4) and* `FooVisualizer` *(right, see listing 2.3).*

contained in `Foo` (lines 19 and 20). The function `evaluateControls` is called when the dialog is closed using the 'Ok'-button. Here, the content of the line editor are transferred to the connected instance of `Foo` (line 26). Calling `contentsChanged` on this instance will inform all visualization classes about this change. This will for example cause an update in the permanent visualization of `Foo`, as described above.

2.4.4 Important classes

Table 2.3 shows a selection of classes that are important for extending the existing functionality of FGControl.

2.5 Discussion

FGControl was initially designed as a software to interface factor graph functionality to real robotic systems or any other controllable system. In the course of this thesis project, it became apparent that a more powerful tool was required.

In order to gain an understanding of how message passing on factor graphs can be used to exert control over a system like a simple pendulum, we needed to explore many different factor graph setups, change the number of states in variables, change the encoding of parameters of the robotic system, use different learning rules etc. In addition, we needed some method to reliably store setups together with parameters and recorded data traces. Of course, it was necessary that these setups could be restored and edited, which implies the possibility to inspect all parameters of a setup.

Because there existed no software that would have been flexible enough to meet our requirements, FGControl was extended to a larger interfacing and visualization framework. In order to later allow the usage of objects with arbitrary properties such as nodes with different message passing behaviour or new data sources, the software was designed

Class name	Properties of child classes
`FWSerializable`	Child classes can automatically be detected by the framework and contents of instances will be stored and restored to disk.
`FWConnectorClass`	Child classes can dynamically be instantiated by the user and can define their own user interface through functions described in table 2.2.
`ExtendedConnectorClass`	Convenience class that extends `FWConnectorClass`. The function `declareConnectedSerializable`, allows to create links to other objects that are automatically updated and are editable in a dialog of class `ExtendedConnectorClassDialog`. (Use `synchronizeConnectedSerializable`) when synchronizing these links. Connections will be established automatically.)
`DataModel`	Child classes can function as data sources that serve data points to a factor graph. A data source should contain instances of `DataVariable`, which can be added using the function `addVariable`.
`DataVariable`	Instance of `DataVariable` serve as the interface between a variable node in a factor graph and a data source by providing the state of one observed variable. Variable nodes in a factor graph can be connected to an instance of a `DataVariable` through their respective dialogs.
`Node`	Child classes can be instantiated as members of a factor graph. New nodes can define their own specific message passing behaviour.
`AbstractTrainer`	Child classes can be added as factor node trainers to an instance of `GraphTrainer`, which allows to add new learning algorithms to FGControl.

Table 2.3: *Important serializable classes of FGControl. Child classes of the given classes can be used in different contexts and thereby allow to extend the existing functionality.*

as a modular framework, which would allow to store, restore and edit arbitrary objects as part of a setup.

Emphasis was placed on the visualization and editing capabilities for factor graphs. Factor node contents can be permanently monitored, as well as messages between nodes in the graph. Nodes can be added to or remove from a graph at any time and variable nodes can always be resized. Factor node contents can easily be edited by the user, for example by using a simple expression evaluator.

This software proved to play a central role in the evolution of this thesis project. Its flexibility allowed to quickly set up new experiments involving different types of data sources or controllable systems and different factor graph topologies. Almost 200 different setups have been evaluated in this thesis project, which illustrates the importance of this software framework for this project.

At the same time, the integrated visualization mechanisms greatly helped to identify and understand problems that occured in these setups. First, visualization of factor node contents helped to understand the capabilities and restrictions of different learning rules. Second, message inspection was important to gain insights about different coding schemes, such as population codes (see section 3.5). Third, the user interface allowed to quickly search parameter spaces for sensible values, for example for finding suitable learning parameters. Fourth, the software made it easier to picture the effects of different factor graph topologies, which for example was important to develop the concept of HNNs (see section 4).

We believe that this software will be a very useful tool for future work related on graphical models. While it was originally designed to interface factor graphs to robotic systems, it could easily be extended to connect very different inputs to factor graph setups, such as vision sensors. Its versatility and modularity also allow it to form a foundation for many other software projects.

Future work

A software is never finished. It can rather reach a state at which it fulfills certain requirements. In its current version, the software proved to be a useful tool for this thesis project, but it might require extension for future projects.

Message passing between objects in FGControl could be standardized. In the current version, communication between objects is specialized for each connection. For example, nodes in a factor graph communicate with each other using certain functions, while a different set of functions is used in the communication between variable nodes and a data source. Standardization would allow to build generic classes that display messages, record these or can generate user-defined messages. At the moment, messages in a factor graph

can be inspected, but these inspection tools cannot be used to visualize other types of messages.

Another future extension could of course be the possibility to compute in multiple threads or on distributed machines. In the course of this thesis project, the performance of the single-threaded FGControl was sufficient, but larger setups could require to engage multiple CPUs. The speed of message passing on a factor graph can also be increased by using general processing on graphical processing units (GPGPU).

At the moment, the software is compiled into one executable file. It can be extended by adding new classes and compiling these together with the current version. Future versions could add a mechanism to dynamically load libraries. These libraries could then contain extension modules. The flexible serialization and factory mechanisms that form the foundation of FGControl will allow classes from dynamically loaded libraries to automatically become available in the software.

Finally, it will be useful to interface FGControl to the programming language *Python*, which is becoming more and more popular for software projects in science. Programs in *Python* could benefit from the fast implementation of factor graph computation in native machine code. Several classes such as `FactorNode`, `VariableNode` and `FactorGraph` were already partly interfaced to *Python* using the *PythonSIP*-framework during this thesis project, which proved to be a straightforward task. The *PythonQt*-framework would also allow integration of python-functionality into FGControl, such as a python console. Such an integration of a scripting language would largely extend the possibilities for flexible experiment setups.

Chapter 3

Motor control with factor graphs

3.1 Motivation

Robotic systems are commonly composed of several joints that are connected through hinges. Actuators allow to modify the position of the joints. Interestingly, the spatial position of a joint depends exclusively on the positions of connected joints and the current angles at the connecting hinges. Hence, given the position of connecting joints, the position of a certain joint is independent of all other joints. Such conditional independences can also be identified in the human body: The position of a hand is independent of the shoulder's position if the forearm's position is known.

These conditional independences imply that global functions describing the state or dynamics of a robotic system might be factorizable. One example for such a global function is the joint probability distribution (JPD) describing the spatial positions of the robot's joints. Since the individual joint positions only depend on connected joints, this global function can be defined by a set of local functions, where each of these local functions captures the effect of one joint. Intuition suggests that this approach could also extend to functions that are used to control a robotic system. Consider for example a hand-shaped gripper and assume a model for its kinematics and dynamics would be given, which would allow precise control. If this gripper would be extended by a few joints to a robotic arm, the model representing the gripper should be reusable, which would simplify the task of learning to control the full arm. While the complete model for the arm might factorize into a product of small functions, it could consists of several modules that need to communicate very little information between each other.

High-dimensional functions that can be factorized into products of lower-dimensional functions can naturally be represented by factor graphs, which by definition merely are visualizations of such functions. For example, in order to represent the JPD over joint angles and joint positions mentioned above, a factor graph can easily be constructed: For

each joint angle, a factor node could be added which would connect three variable nodes, with one of them representing the joint angle itself and the other two the positions of the adjacent joints. Belief propagation on the resulting graph would allow to infer all joint positions when given a single joint position and all joint angles.

In order to plan movements of a robot, though, it is necessary to consider its dynamics, whereas the JPD in the last example decribes the kinematics of a robot, which is a static relationship between angles and spatial positions.

In this part of this thesis, we will investigate general approaches of how dynamical systems can be modelled with factor graphs and how such factor graphs can be used to directly control robotic systems or be used as part of a robotic controller. In different scenarios, we employ factor graphs with discrete state variables that are trained based on recorded data traces of a robotic system.

While the topology of all factor graphs we employ is assumed to be given, the parameters of the factor nodes are initialized with flat priors and are trained from recorded data.

Discrete state variables bear the advantage that message passing is straightforward and, more importantly, a range of learning algorithms are available that allow to train the parameters of a factor graphs based on experience. Furthermore, their use is in many aspects more intuitive because the computations performed in message passing can be more easily followed. Finally, concepts developed for discrete state spaces can later be mapped to continous spaces.

The application of factor graphs to control problems requires the solution of several challenging problems: First, a suitable encoding for parameter values needs to be found. Since we are using variables with discrete state spaces, we need to map continous parameter values to discrete states. Outputs of the applied factor graphs will be distributions on discrete state spaces, which need to be converted back into specific control parameters. Second, data sampling in real robotic systems is time-consuming, which causes training data to be sparse. In addition, data usually cannot be sampled uniformly. Factor graphs trained on sparse and nonuniformly sampled data are likely to behave different from expectation. Third, we will inspect in which ways factor graphs can be used to model the dynamics of a robotic system.

3.2 Controlling a toy robot

In this section, control concepts based on factor graphs will be explored in the context of a small robotic system with a single degree of freedom. The simplicity of such a system allows to explore various types of variable encodings and to more easily understand the

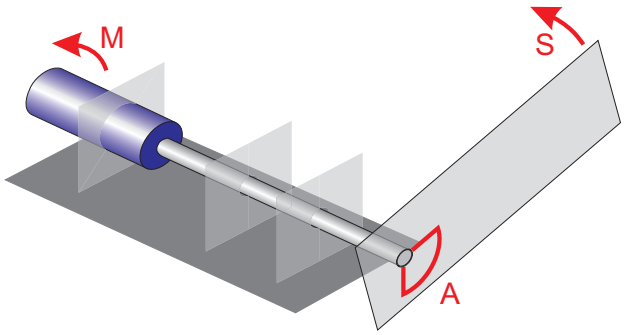

Figure 3.1: *Schematic view of a compliant pendulum with one degree of freedom. Input to this small robotic system consists of one parameter M, which controls the motor and thereby relates to the torque that will be applied to the shaft. The output of the pendulum defines the current angle A, which can be used to also compute the angular speed S.*

effects of factor graph training with sparse and nonuniformly sampled data.

3.2.1 The robot

A small compliant pendulum was chosen as a robotic system in order to define a first task for factor graph based motor control. Figure 3.1 shows a schematic picture of such a compliant pendulum, which was designed by Andreas Keller and Jorg Conradt [Kel08]. The input to the pendulum consists of a single parameter, which defines a torque that a motor will apply to the shaft of the pendulum. The pendulum is subject to several other forces: Friction, centripetal force, gravity and inertia, which all influence its movement.

Two circular potentionmeters at the shaft of the pendulum are used to measure the angle of the pendulum's blade. A microcontroller that also controls the voltage applied to the motor continuously samples the two sensors and transmits these values at 1000 Hz via a simulated serial port through a USB connection to a computer. The sensor readouts are then decoded into angular values and the angular speed is calculated as a low-passed first derivative of the angle. The software used to evaluate these angles reads out the values for angle and speed at 20 Hz.

While the angle is measured at a precision of approximately 0.07°, the temporal precision for reading out values is low. Data points are read out every 50 ms with a precision of about $\pm 5\ ms$. More importantly, the time that passes between sending a motor command and detecting its effect varies between 20 and 80 ms. This temporal variation is the main source of noise in the system.

One might ask why it would be necessary to use a real pendulum instead of a simulated version. Simulated robotic systems bear the advantage that their creation is much easier, which also makes it cheaper. Much more important though, simulated systems allow to

quickly sample large amounts of data points and this sampling can be performed in a uniform way.

However, the intention in this project was to explicitly deal with the shortcomings of real world systems, including time-consuming and non-uniform data sampling, which will generate very noisy data. While these shortcomings can of course also be simulated, it is easy to neglect certain problems of a real system. For example, the temporal imprecision described above introduces high amounts of noise that might be ignored in a simulated system. Additionally, a real robotic system is often more appealing to spectators, because it allows to demonstrate the practicality of an approach.

3.2.2 The task

A simple control task was created for the pendulum described in the last section: Given a target angle, apply angle-dependent torques to the shaft such that the pendulum arm moves to the desired angle and remains there. A factor graph should be used to model the dynamics of the system and be used to generate the controlling output. Importantly, the controller should have no prior knowledge about the robotic system - all parameters of the model should be trained based on data collected in sample runs.

As described in the last section, data collected from the pendulum is noisy, in particular with respect to the temporal precision when responding to a command. Obtaining training data is time-consuming and therefore, this data is sparse. Moreover, the input space is sampled non-uniformly. The chosen task requires to understand the dynamics of the robot in spite of these restrictions.

In all setups, we use a simple exploration function to generate training data. This function is called every 50 ms and at each timestep, it modifies the current torque applied to the pendulum with a probability of 10%. New torques are chosen randomly with a bias towards torques that have been less explored in states similar to the current one.

3.2.3 Discretizing continuous observations

Variables in a robotic system are most commonly of continuous nature, whereas we are using factor graphs with discrete state variables, ss mentioned in section 3.1. This makes it inevitable to discretize values from continuous observations in order to yield messages that can be used in the factor graphs that we consider.

The most straight-forward discretization strategy can be outlined as follows: Assume a continuous variable x is given which can take up values between x_{min} and x_{max} and should be encoded as a message with N elements. The intervall $[min, max]$ is split into N intervalls of equal length and each intervall is assigned to one of the elements of the

message. For a specific x, all entries in the message will be set to 0, except the entry which corresponds to the intervall containing x, which is set to 1. We assume that the elements of the message are sorted in order of their corresponding intervall.

Another interpretation of this discretization is that each element of the message is assigned a specific center of activity in the intervall $[min, max]$ and is only activated by an observed value x if its center of activity is closer to x than all the other centers of activity. This interpretation allows to define a *similarity* or a distance measure between an element of a message and an observed variable x as the distance between the element's center of activity and x.

3.2.4 Simple approach: Joint probability distribution in one factor

Three variables are sufficient to describe the state of the compliant pendulum described above: angle A, angular speed S (in the following referred to as 'speed') and the current motorcommand M which corresponds to a torque the motor applies to the shaft. These could be divided into input and output variables, but this separation is not necessary when analyzing the statistics of the system.

Further derivatives of the angle other than the speed could be considered as state variables, but given the torque applied to the shaft, these derivatives would not confine the current state of the pendulum since all other forces acting on the pendulum are functions of the angle or the speed.

An assignment to the three described variables describes the state of the pendulum at a specific timepoint t, which we will refer to as the *current state*. A model describing the relation between these three variables could be used to compute the probability of a certain state, but it would not allow any prediction about expected changes of the system because it would not contain any information about the system's dynamics. Therefore, we added a fourth variable, which encodes the angle A_{+1} at a timepoint $t + \Delta t$, which we will refer to as *future angle*. The time difference Δt is a time constant, which needs to be chosen before training. We typically used values of around $200ms$ for the pendulum setup.

The low dimensionality of the task that we chose allows to use a straight-forward approach, using a factor graph with a single factor node to capture the statistics of the four variables. This factor can connect to four variable nodes, each representing one variable of the system. With sufficient amounts of data, the single factor can be trained to approximate the joint probability distribution $P(A, A_{+1}, S, M)$ (JPD). From a trace of recorded data points consisting of A, S and M at times t, training data can be generated by setting

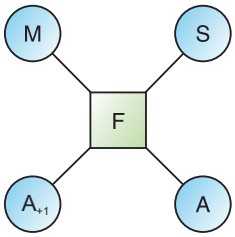

Figure 3.2: *A trivial factor graph for controlling a pendulum. All variables nodes are connected to a single factor node. The factor graph can be used to learn the relation between current angle A, future angle A_{+1}, angular speed S and the current motor command M, which defines the applied torque.*

Training in this case can be performed by gradient ascent: A four-dimensional histogram $f(A, A_{+1}, S, M)$ can be used to count the occurrences of every combination of possible values for A, A_{+1}, S and M. The estimated JPD is then equal to this histogram function, normalized to a sum of 1.

The estimated JPD can be used to compute several interesting functions. Marginal or prior probabilities for a subset of the variables can be computed by summing the entries of the JPD over all remaining variables, for example:

$$P(A, A_{+1}) = \sum_s \sum_m P(A, A_{+1}, S = s, M = m)$$

If a subset of the variables is observed, it is possible to compute conditional probabilities for the remaining variables, for example:

$$\begin{aligned} P(A_{+1} \,|\, A = \alpha, S = s) &= \frac{P(A_{+1}, A = \alpha, S = s)}{P(A = \alpha, S = s)} \\ &= \frac{\sum_m P(A = \alpha, A_{+1}, S = s, M = m)}{\sum_{\alpha_{+1}} \sum_m P(A = \alpha, A_{+1} = \alpha_{+1}, S = s, M = m)} \end{aligned}$$

In the control task that we want to solve, a motor command should be computed as a function of a target angle α^* and the current state of the system, defined by the current angle α and speed s. Given these three values, the conditional probability distribution for the motor command can be computed:

$$\begin{aligned} P(M \,|\, A = \alpha, A_{+1} = \alpha^*, S = s) &= \frac{P(A = \alpha, A_{+1} = \alpha^*, S = s, M)}{P(A = \alpha, A_{+1} = \alpha^*, S = s)} \\ &= \frac{P(A = \alpha, A_{+1} = \alpha^*, S = s, M)}{\sum_m P(A = \alpha, A_{+1} = \alpha^*, S = s, M = m)} \end{aligned}$$

Note that here, the desired angle is used as if it would be a true future angle. Since

the desired angle is the angle the pendulum should be in at a future time, this seems reasonable. However, the interpretation of the variable has been changed in comparison to the training phase. The future angle was determined as a result of an observation, whereas the desired angle represents a yet unobservable state. It is therefore possible, that the configuration is impossible, which would result in a flat conditional probability distribution for the motor command. This problem is particularly interesting in cases when the desired state is not reachable within the time Δt, because data points describing such transitions are not part of the training data.

3.2.5 Pendulum control

Using the inferred conditional probability distribution for the motor command, a simple closed-loop control algorithm can be devised:

1. Receive a desired state - a target angle α^*- as input.
2. Observe the current state of the system, comprising the current angle α and the angular velocity s.
3. Use a trained factor graph representing $P(A, A_{+1}, S, M)$ to compute the conditional probability distribution $P(M \mid \alpha, \alpha^*, s)$.
4. Select a new motor command m as a function of $P(M \mid \alpha, \alpha^*, s)$ and apply it to the pendulum, if necessary.
5. Repeat steps 2 to 4.

The precision of the described controller is of course limited by the discretization of observed parameters. But in addition, this simple controlling scheme suffers from typical control problems such as oscillations around a target state. In the described four-variable setup, the desired future state can only be defined in terms of a desired angle, the speed at that point is not considered, neither is the state of the pendulum after reaching the goal. The controller will therefore select those actions that will cause a transition to the target angle, but neglect the future effect of that action. Hence, selected motor commands might be too high and cause overshooting, which will result in pendulum oscillations around the target angle, or fast rotations of the pendulum in one direction.

These control problems could easily be solved by extending the controller to a PID-controller [Min22], but this would change the nature of the controller from a purely factor graph based control into a classical control solution which would use the factor graph output as a guiding input.

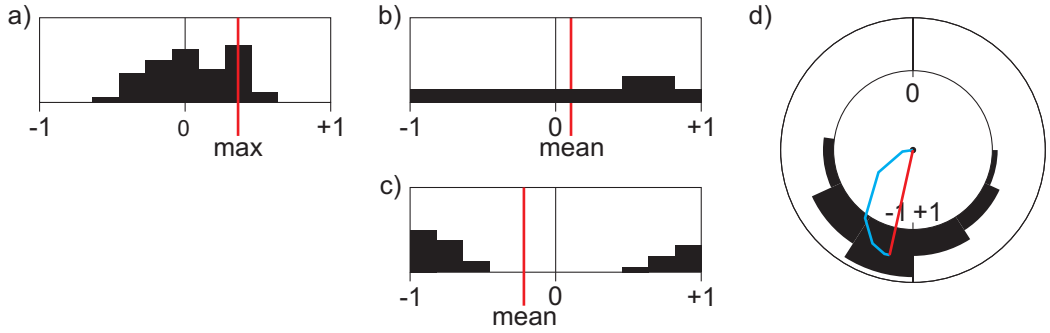

Figure 3.3: *Decoding output variables of a factor graph. Belief propagation on a factor graph yields vectors defining the marginal proabilities of unobserved variables. In order to use outputs of the factor graph for action selection in a controllable system, these probability distributions need to be decoded. a) The maximum of a distribution might be subject to noise and therefore be a bad representation of the population response. b) and c) The mean value is biased towards central values, which causes problems for example when the signal-to-noise ratio is low or when the distribution is multimodal. d) Circular mean: The distribution from c) is arranged around a circle such that each value maps to an angle, which in turn defines a direction vector. An output (red line) is determined by vector addition of all direction vectors, weighted by their respective probability (blue line, segments correspond to the individual contributions).*

A pure factor graph based solution could be to include a fifth variable, namely the future speed of the pendulum. This would allow to set the desired speed at the destination to 0, which would cause the controller to select those motor command that will make the pendulum rest at the destination.

Concluding, it is out of question that it is possible to design a much better control solution for a one-dimensional pendulum. However, we did not pursue any of the above strategies to optimize the control performance. The performance that was obtained with the described controlling algorithm was sufficient to evaluate the effects of training and the simplicity of the algorithm allowed to better trace problems caused by sparse and non-uniformly sampled training data.

3.2.6 Motor command selection

The control algorithm described in the last section relies on a function that returns a single motor command when given a probability distributions over a set of discrete states. The simplest selection mechanism would be to chose the motor command that is assigned the highest probability. However, this selection strategy bears two disadvantages: First, the maximum of the probability distribution might change its position very quickly, which would result in jittery movements of the pendulum. This problem would require low-pass filtering. Second, the maximum only takes into account a single value which might be

erroneous due to noise in the training data. Instead, it is more desirable to take into account the full population response.

This could be achieved either by sampling motor commands from the returned distribution or by computing a weighted average by multiplying each returned probability with its corresponding motor command and summing over all entries. Sampling would cause quickly changing motor commands which would result in jittery movements. Hence, the motor command would have to be low-pass filtered again, which in turn would give the same results as a weighted average.

Both approaches take into account the whole population response, but in practice, they will bias the selected motor commands towards low values. Figure 3.3 shows two examples where this effect could influence the selection: First, the mean value of a distribution will be biased towards the center of the distribution if the signal-to-noise ratio is low. Second, it can happen that two competing actions are possible in a certain scenario, as depicted in 3.3c). The mean value will select neither of the two competing actions, but instead choose a motor command that lies in the middle between the two possible ones. More generally, these effects are reflecting the fact that the expected value of the mean for random distributions on a fixed intervall is equal to the center of the intervall.

We are using an unbiased motor command selection process as depicted in figure 3.3d): All possible motor commands are arranged in order on a circle and based on its angular position, each motor command is assigned a unit vector that points to the motor command's direction. In order to select a motor command from a list of probabilities, these unit vectors are weighted by their corresponding probability and finally added to give a response vector. The direction of this response vector determines the motor command that will be selected.

At first, it might seem peculiar to arrange the highest and the lowest possible motor commands adjacent to each other on the circle. However, situations in which both of these two motor commands are possible require the selection of either one of the two actions. The described mechanism will select the one that caused the stronger population response.

It should be noted that the described method will fail in situations where two motor commands are possible and these correspond to two opposing positions in a circular arrangement. In order to correctly decode such setups, a more sophisticated method using for example winner-take-all mechanisms would be required. However, in the setups used in this project, such separated population responses were never observed, which is why the descibed decoder delivered a usefull performance.

3.2.7 Limitations

The simple approach using a single factor node bears several disadvantages that render it impracticable for most applications. The number of parameters contained in the central factor node scales proportional to the number of states of all four variables. If high precision is required in these variables, this number of parameters quickly exceeds 10^5, which would set high constraints on the number of data points that need to be observed in order to sufficiently approximate the JPD without overfitting. Since data points can only be collected at around 20 Hz, this process would be very time consuming.

Another drawback of the simple approach is its inability to generalize to unseen data. Since the future angle A_{+1} is fixed to a distinct future time during training, the factor graph will only be usefull to predict correct motor actions if at least one datapoint was seen during training that contained the precise desired combination of current angle, current speed and future angle. Otherwise, the conditional probability distribution returned for the motor command will be flat.

Population codes

Whenever the target angle is out of reach from the current configuration, it would be usefull to at least return a motor command that will bring the pendulum closer to the desired state. One way of computing such a motor command would be to iteratively move the target closer to the current state as long as the conditional probability distribution for the motor command remains flat.

However, this iterative procedure would result in a search through the input space and would be infeasible in larger systems in which the target state is high-dimensional. Instead of iteratively searching through the input space, it seems reasonable to activate several states simultaneously with a gradient towards the target state. While a single state activation defines a single state as the only desirable future state, such a gradient input signals that many target states are desirable, but to different degrees. Figure 3.4c) shows an example for such a gradient encoding scheme.

In such an encoding scheme where multiple states of the input are activated at the same time, each activation level contains some information about the desired target state. It therefore resembles a population code.Population codes in general can have arbitrary shapes, but we will use codes where the activation of each state of a variable depends on the similarity of a state to the observed value. Hence, a similarity function is used to define the activation of each state. Several functions were tried as candidates for this similarity function and the best performance was obtained using exponentially decaying functions. Such exponential tuning curves for each state allow to put emphasis on states

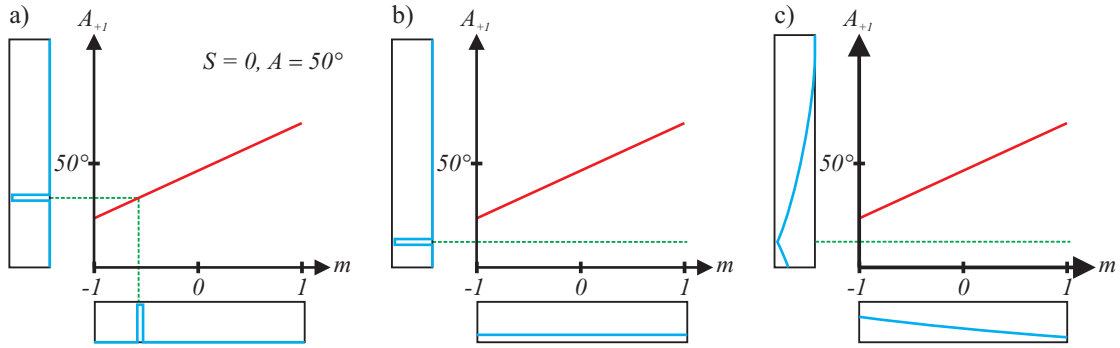

Figure 3.4: *Inferring a motor command when given a target angle. Shown is a schematic two-dimensional cut of a four dimensional factor node at $A = 50°$ and $S = 0$. The red line indicates a fictive set of training data points. a) A motor command can be inferred easily when the desired angle is close to the current angle and can be reached within a certain time frame. b) The probability distribution for the motor command is flat if the angle is further away. c) This problem can be counteracted by using population codes as input.*

representing the true observation. This increases the influence of training data points that were very similar to the currently observed state.

In a population code as described above, the vector of state activations can be interpreted as a weighting of the individual states. In case of the target state, this corresponds to a 'desirability' of each state. However, it might also be usefull to apply such gradient inputs to other input variables: If, for example, no data points have ever been observed for the current angle, but many data points have been observed for a closeby angle, it makes sense to approximate the output motor command using these data points. Again, a population code can introduce a weighting for the individual states which then defines the degree to which data points in the training set should be considered.

Summarizing, the use of gradient shaped population codes for encoding the inputs to a factor graph seems to bear two advantages: First, such codes should allow to select actions when the target state is too far away to be reached in a single timestep. Second, this approach seems to introduce generalization properties into an overfitted model.

Necessary modifications for the use of population codes

If gradient shaped population codes are used to define the activation of observed variables in a factor graph and standard belief propagation is used to infer the probability distribution of unobserved variables, the inferred distributions will be biased by the prior probabilities of the observed variables.

Consider for example figure 3.5, where a population code is used as an input message for a two-dimensional factor node. Assume the factor node was trained to represent an

identity matrix and the training data was noisy and not uniformly distributed. Due to the non-uniform distribution of the training data, the states of the input variable A have different prior probabilities. When using the sum-product algorithm to compute the output message, states of A with higher prior probability will have a larger impact on the result than others. The result is that training data points which were very similar to the currently observed state might be neglected, if they were outnumbered by closeby datapoints, as is the case in this example.

This topic will be discussed in detail in section 3.5.2. In short, standard belief propagation yields correct conditional probabilities for an unobserved variable when the observations defining the states of the remaining variables are deterministic and therefore only single states are activated. However, when multiple states are activated for observed variables, messages to unobserved variables will be influenced by the prior probabilities for the states of the observed variables.

In order to obtain the desired conditional probability distribution for the motor command given multi-state activations at the observed variables, the factor node representing the joint probability distribution $P(A, A_{+1}, S, M)$ needs to be transformed into the conditional probability distribution $P(M|A, A_{+1}, S)$, which corresponds to a normalization along the output dimension. In figure 3.5, this method is illustrated on the right.

3.2.8 Simulations

We used a low-resolution discretization of the variables in the factor graph shown in figure 3.2 to store the instantaneous dynamics of the described compliant pendulum (A, A_{+1}: 12 states, M: 13 states, S: 9 states). Contents of the single factor node were set to the relative occurrences of the respective configurations in a data set of 10^4 data points.

Figure 3.6a shows the inferred motor commands for different current angles and target angles at a speed of 0. For large distances between current and target angle, the inferred motor command remains 0. This indicates that transitions of this length were not contained in the training data set. This lack of reach can be removed by using gradient shaped population codes as inputs for the variables A, A_{+1} and S. Results for a system using such inputs are shown in figure 3.6b. Here, the factor node was normalized in output direction in order to improve inference results.

Performance evaluation

The described controller was tested during live control while the pendulum angle was monitored. For evaluating the performance of various pendulum controller, we used a fixed training protocoll: With time lags of $2.5s$, 500 random target angles were set as

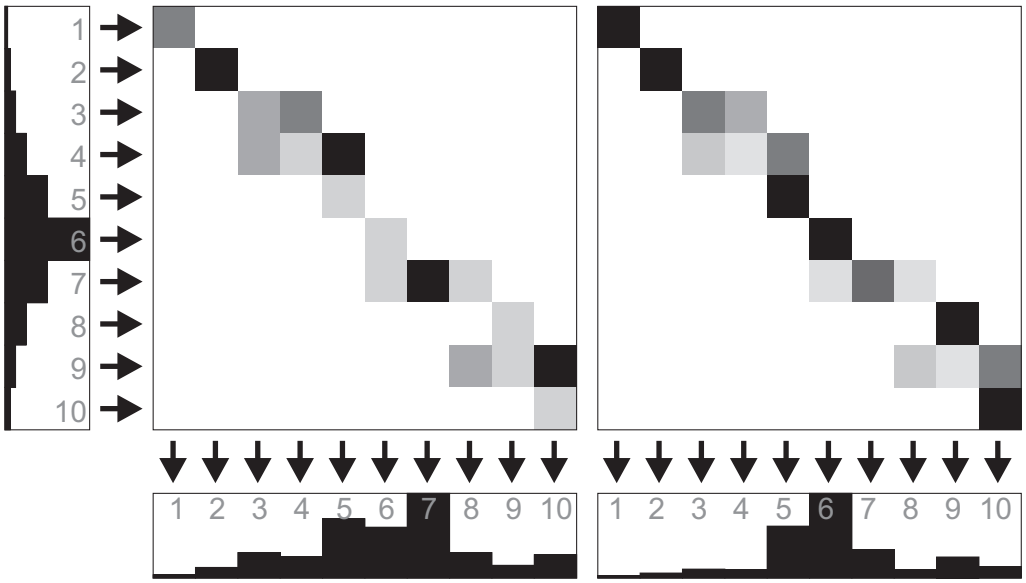

Figure 3.5: *Influence of the prior probabilities of observed variables on the message to an inferred variable. The left matrix shows a fictive joint probability function $P(A, B)$, which was estimated from noisy and non-uniformly sampled data, where $A = B$. When the state of A is defined in terms of a population code (incoming message), the prior probabilities $P(A)$ will influence the result. Here, the highest activity in the output is elicited at position 7 instead of 6, because the prior probability $P(A = 7)$ is higher than $P(A = 6)$. This problem can be solved by using the conditional probability distribution instead, as shown on the right. Here, the entries of the matrix are normalized row-wise in order to represent $P(B|A)$. The output shown on the right is closer to the expected ouput and the remaining deviations are a result of the noisy identiy matrix and perfectly match the given model. This topic is discussed in more detail in section 3.5.2.*

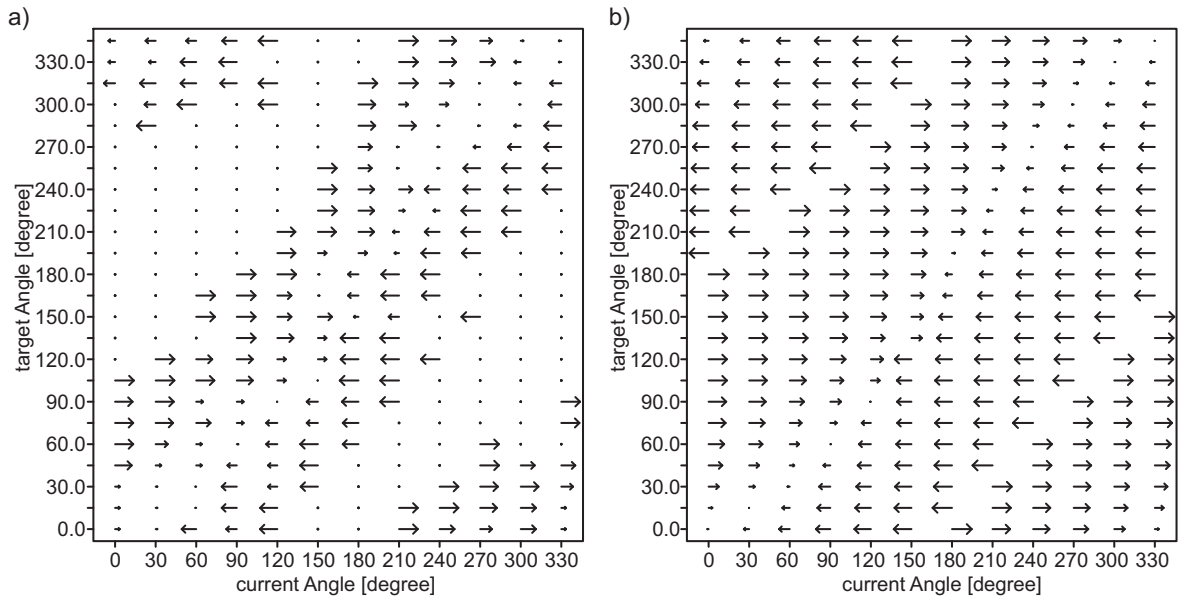

Figure 3.6: *Vector field generated by a simple factor graph. A factor graph with a single factor node was used to store the instantaneous dynamics of a compliant pendulum. After training, it was used to infer motor commands at different configurations of current and target angle for a fixed angular speed of* 0*. The lengths of the arrows indicate the inferred torques and their direction for the various configurations as indicated by the position of their origins.*

control goals. A single random sequence was reused for all tests in order to allow better comparability. A sample trace of the angle is shown in figure 3.7. The angle difference between pendulum position and target angle was recorded during the final second of each intervall and eventually, the average of these deviations was computed. We will refer to this value as the *average angle deviation*.

For the simple approach as described in the past sections, three setups were tested, each of which were constructed as described before, but different encodings were used for the observed variables. Average angular deviation for single-state activations was 49°, resulting from the fact that the model was not able to model transitions longer than the chosen time constant $\Delta t = 200\ ms$ and the default motor command was set to 0. If this default value was set to the highest angular torque, the performance of the system was significantly improved to 11.2°, which is close to the optimal expected average angular deviation of 10° for a discretization of the angle into 12 states.

By encoding the observed variables as population codes with shape of an exponential decay, performances of 15.9° and 9.5° were obtained. For the latter value, the factor node was normalized in output direction as described in figure 3.2.7 and section 3.2.7. As expected, the straightforward application of gradient shaped population codes does not result in an improvement over a discrete encoding. However, the factor graph represent-

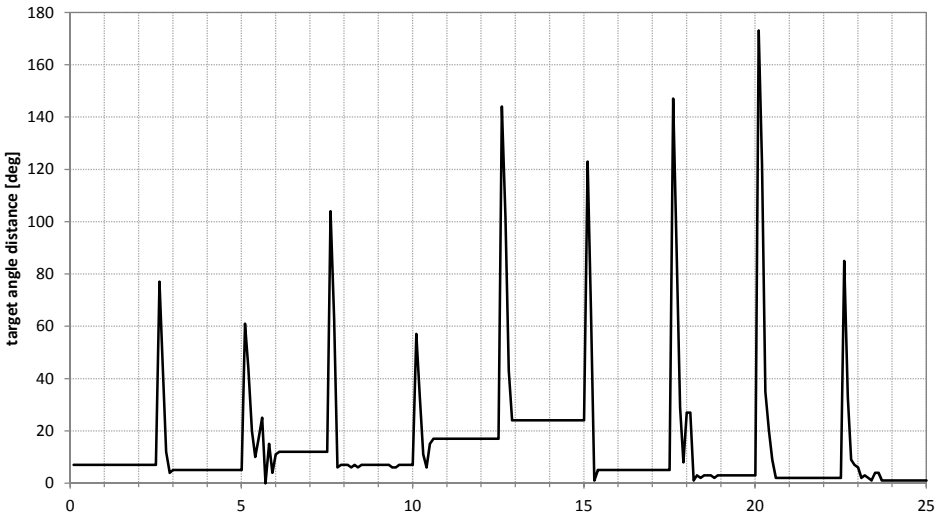

Figure 3.7: *Pendulum control using a trivial factor graph. A factor graph with a single factor node was used to infer actions during live control, which were then decoded into angular torques. Every* 2.5 *seconds, a new target angle was randomly chosen. Shown are the deviations from the desired angle in dependence of time.*

ing the conditional probability distribution for the motor command allowed to improve precision beyond the expected precision for a discretization for the angle with 12 states, which was verified in multiple test runs.

3.3 Training

In all setups described in this chapter, factor graphs are used to model the joint probability distribution over a set of variables. We assume that no prior information about the true underlying JPD is available. Instead, all described models are trained based solely on observations. Of course, these observations represent a finite sample of all possible observations and hence only allow to create an approximate model of the true JPD.

In section 3.2.4, a factor graph containing a single factor node was trained to represent the JPD over a set of four parameters. In this case, learning the parameters was particularly easy because the entries of the factor node can simply be set to the observed frequencies of the corresponding events. For factor graphs containing multiple factor nodes and that possibly contain hidden variables, learning the parameters requires more sophisticated maximization methods.

In general, the aim of model training is to adjust the parameters θ of the model in such a way that the probability distribution defined by the model approximates the observed

probability distribution and hence the true JPD. This approximation can be done by maximizing the likelihood of the observed data given the model, $p(\boldsymbol{Y}|\theta)$ or the posterior probability $p(\theta|\boldsymbol{Y})$. When the priors for the model parameters $p(\theta)$ are uniform, the posterior probability equals the likelihood. Since we have no knowledge about $p(\theta)$ and therefore have to assume uniform priors, we will focus on the likelihood as the quantity to be maximized.

We used two learning methods that were previously described by Rolfe [Rol06]: Gradient ascent and expectation maximization. Both methods aim at maximizing the log-likelihood of the observed data, which is equivalent to maximizing the likelihood because the logarithm is a strictly increasing function. In the remaining part of this section, both methods will shortly be outlined.

3.3.1 Gradient ascent

From the work of Rolfe [Rol06], it follows that the gradient of the log-likelihood of the data with respect to a model parameters can be calculated as follows:

$$\frac{\partial}{\partial f_{a'}(X_{a'}=x)} \log p(\boldsymbol{Y}|\theta) \propto \frac{1}{f_{a'}(X_{a'}=x)} \cdot \left(\left\langle p(X_{a'}=x|Y(X)=Y^i) \right\rangle_i - p(X_{a'}=x) \right) \tag{3.1}$$

Here, $f_{a'}$ denotes a single function of the factor graph and $X_{a'}=x$ indicates a specific variable configuration for this function. Hence, $f_{a'}(X_{a'}=x)$ corresponds to a single parameter of one function.

In order to calculate the gradient with respect to a single parameter of the factor graph, the marginal probability for the corresponding variable configuration $p(X_{a'}=x)$ and the average probability of the configuration given the observed data $\langle p(X_{a'}=x|Y(X)=Y^i)\rangle_i$ need to be known. In a non-loopy factor graph, $p(X_{a'}=x)$ can be determined by fixing the messages from all observed variables to non-informative messages (i.e. uniform messages) and running the belief propagation algorithm until convergence. Multiplying the outer product of the incoming messages to the factor node $f_{a'}$ with the entries of $f_{a'}$ and subsequent normalization yields the vector of probabilities for all configurations of $X_{a'}$. The probabilities $p(X_{a'}=x|Y(X)=Y^i)$ can be obtained similarly, but with the observed nodes fixed to the observation Y^i.

The gradient can be used to update the parameters by following the gradient for a

distance η:

$$f_{a'}^{t+1}(X_{a'} = x) = f_{a'}^{t}(X_{a'} = x) + \eta \cdot \Delta f_{a'}^{t}(X_{a'} = x) \tag{3.2}$$

$$\Delta f_{a'}^{t}(X_{a'} = x) = \frac{\left(\langle p(X_{a'} = x|Y(X) = Y^i)\rangle_i - p(X_{a'} = x)\right) / f_{a'}(X_{a'} = x)}{|| \left(\langle p(X_{a'}|Y(X) = Y^i)\rangle_i - p(X_{a'})\right) / f_{a'}(X_{a'})||_2} \tag{3.3}$$

Small values of η will result in slow convergence. Larger values will yield faster convergence but will reduce the probability to find a local minimum of the log-likelihood for $f_{a'}$. The learning rate can also be defined as a function of time, typically by choosing a monotonically decreasing function like $\eta(t) = \eta_0 \cdot \alpha^t$ with $\alpha < 0$. However, it is difficult to determine a usefull function ηt because the time until convergence cannot be predicted.

We found that two simple techniques can improve the convergence properties: First, $\Delta f_{a'}^{t}(X_{a'})$ can be low-pass filtered. This will prevent the parameter vector from moving around the maximum and therefore allows to set η to higher values. Second, $\eta(t)$ can be defined as a function of $\eta(t-1)$ and be changed depending on the angle γ between the vectors $\Delta f_{a'}^{t}(X_{a'})$ and $\Delta f_{a'}^{t-1}(X_{a'})$: $\eta(t) = g\left(\eta(t), \gamma\right)$. The following definition resulted in fast and precise convergence:

$$\eta(t) = \begin{cases} \eta(t-1) + (a-1) \cdot \cos\gamma \cdot \eta(t-1) & \text{if } \gamma \leq 90^\circ \\ \eta(t-1) + (1 - \frac{1}{b}) \cdot \cos\gamma \cdot \eta(t-1) & \text{else} \end{cases} \tag{3.4}$$

According to this definition, the learning rate is multiplied by a if the gradients at two subsequent time steps point into the same direction and divided by b if they point into opposite directions. Note that a and b are given in similar units and can therefore directly be compared to each other. The change is modulated by $\cos\gamma$ such that η changes more slowly in non-extreme cases. It is usefull to have $a < b$ (we used $a = 1.1$ and $b = 3$). This will cause the learning rate to slowly increase when the gradient does not change and to quickly decrease when the gradient is subject to fast changes.

The described techniques cause the learning rate to automatically adapt to the shape of the curve defined by the log-likelihood in dependance of the model parameters. In parameter regimes where the log-likelihood changes slowly, the learning rate increases in order to speed up the movement through the flat parts in parameter space. Likewise, the learning rate decreases in parameter regimes where the log-likelihood changes quickly.

3.3.2 Expectation maximization

Maximizing the log-likelihood of observed data for factor graphs in which all variables are observed has been shown to be a convex optimization problem. Hence, there will be only a single global maximum of the log-likelihood. This property makes it feasable to

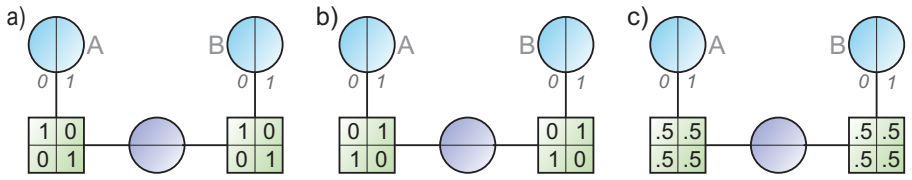

Figure 3.8: *Maximizing the likelihood for factor graphs with hidden variables is non–convex. In a) and b), two parameter configurations for a factor graph are shown, which both maximize the likelihood for two observed data points (A=0,B=0) and (A=1,B=1). If the likelihood maximization would be a convex problem, the set of optimal solutions must be a convex set. The parameter configuration used in c) is equal to the mean of the configurations in a) and b) and would therefore have to belong to the convex set of optimal solutions. It is obvious, however, that this configuration does not maximize the likelihood of the described data.*

apply simple gradient ascent methods as introduced in the last section: If the learning rate is sufficiently small, gradient ascent will converge to the maximum. The single maximum allows to start with a large learning rate and to adjust the learning rate when the maximum of the log-likelihood is missed and the gradient changes its direction.

Maximizing the log-likelihood for factor graphs containing hidden variables, however, is a non-convex problem, which can easily be proved, as shown in figure 3.8. Hence, the log-likelihood may contain many local maxima and the aim of maximization should be to converge to one of these local maxima. Again, gradient methods with sufficiently small update steps would converge to such a local maximum. But such small values for the learning rate would cause learning to become infeasibly slow. And the existence of multiple maxima makes it much more difficult to design a gradient method with a varying learning rate, because the gradients at different timesteps might point to different maxima. This makes it impossible to decide whether the learning rate should be increased or decreased based on the changes of the gradient. Consequently, adaptive methods like the technique described in the last section may not converge at all.

The expectation maximization (EM) algorithm is an iterative procedure to maximize the log-likelihood in the presence of missing or hidden data. It was shown to converge to a stationary point, most likely a local maximum of the log-likelihood [MK96]. Each iteration of the EM algorithm consists of an expectation step (E-step), followeg by a maximization step (M-Step). During the E-step, the expected log-likelihood is calculated using the current estimate for the model parameters. In the M-step, the model parameters are updated in order to maximize the expected log-likelihood found in the E-step.

By first computing the expected value of the log-likelihood and then maximizing the resulting term, Rolfe derived the following update rule for the parameters of a factor graph [Rol06]:

$$f_{a'}^{t+1}(X_{a'} = x) = f_{a'}^{t}(X_{a'} = x) \cdot \frac{\langle p(X_{a'} = x|Y^i, \theta)\rangle_i}{p(X_{a'} = x)} \tag{3.5}$$

This update rule would typically be used as follows: First, the intrinsic marginal probability $p(X_{a'} = x)$ and the marginal probability given the observed data $p(X_{a'} = x|Y^i, \theta)$ can be determined as outlined in the last section. Using the EM update rule, the model parameters can change dramatically within a single update. Therefore, it is necessary to present a large batch of data points in order to determine a good approximation of $\langle p(X_{a'} = x|Y^i, \theta)\rangle_i$ before performing an update. Normally, the complete batch of data should be presented before each update. As soon as the marginal probabilities are determined, the factor entries can be updated. The EM update rule was derived under the assumption that the partition function $Z = \sum_X \prod_a f_a(X_a)$ would only be subject to small changes during an update. Therefore, it would be desirable to change only few model parameters in each update. In practice, all parameters of a single factor node can be updated at once, but changing the entries of several factor nodes in one update would cause too much variation of the partition function.

Note that using the update rule 3.5, parameters will be set to zero when the respective marginal probability given the data is zero. During the next update, the intrinsic marginal probability would correctly be calculated as zero, hence the result of the update would be undefined. In order to avoid these special cases, it is advisable to lower-bound factor entries to insignificantly low values (i.e. 10^{-9}), such that the update rule can be executed without treatment of special cases.

In order to ensure numerical stability, it is furthermore recommendable to normalize the factor entries after each update. Scaling the factor entries will have no effect on the JPD represented by the factor graph, because the partition function will scale analogously.

The EM algorithm does not require any parameters that define the learning rate and it is thus fairly easy to apply it to unknown data. A single update can cause large jumps in parameter space, which allows to quickly approach a maximum of the log-likelihood. Despite these possibly large update steps, the algorithm is guaranteed to converge to a stationary point which most likely will be a local maximum (saddle points are unstable fixpoints) [MK96]. Gradient methods, in contrast, are only guaranteed to approach a single maximum if the learning rate is sufficiently small.

A major drawback of the EM update rule discussed here, however, is the restriction to updating a single factor node during each update. Additionally, each update should only be performed after a large batch of data has been presented to the factor graph. These two properties can cause slow learning behaviour of the algorithm which is why gradient ascent methods with high learning rate can sometimes allow to approach a maximum of

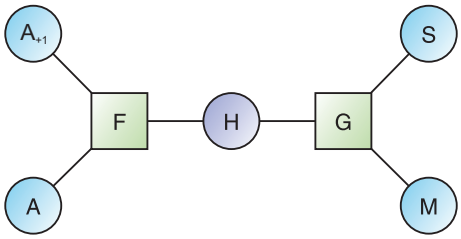

Figure 3.9: *Representation of instantaneous pendulum dynamics in a factor graph with two factor nodes. A hidden variable H was introduced into the graph from figure 3.2 in order to allow a factorization of the joint probability distribution.*

the log-likelihood faster than the expectation maximization algorithm.

3.4 Instantaneous dynamics as a factorized function

The approach described in section 3.2 used a single factor node to connect all variables of the system. Hence, the joint probability distribution $P(A, A_{+1}, S, M)$ (JPD) over these variables was stored by a single function. This stands in contrast to the main idea underlying the framework of factor graphs, which is to take advantage of the way a global function on a high dimensional input space factorizes into the product of local and lower-dimensional functions. Because of the lower dimensionality of the local functions, a factorized representation allows to significantly reduce the number of model parameters.

When the internal structure of a global function is unknown, it may be difficult to select a usefull factorization. In many cases, the global function may not be factorizable at all. The existence of a factorization requires conditional independences between the variables, which might not exist. The joint probability distribution for instantaneous pendulum dynamics for example cannot be factorized, because no conditional independences between the variables exist: In order to compute the future angle of the pendulum, the current angle, motor command and angular speed need to be known.

The introduction of non-observable, or *hidden* variables allows to resolve this problem: Instead of factorizing the global function into a single product of functions, it will effectively be reduced to a sum of products of functions.

Figure 3.9 shows an example factor graph that represents a possible factorization of the same JPD that was discussed in section 3.2.4. A single hidden variable H with $|H|$ states was introduced. The JPD is therefore represented by the following function:

$$P(A, A_{+1}, S, M) = \frac{1}{Z} \sum_i F(A, A_{+1}, H = h_i) \cdot G(H = h_i, S, M) \tag{3.6}$$

The partition function Z is used to normalize the function to a total sum of 1.

A full reconstruction of the non-factorized representation becomes possible if $|H|$ is made large enough: At some point, the number of parameters in one of the adjacent functions reaches the number of parameters in the non-factorized version. Reproducing the original function then becomes a trivial task.

Often, the amount of information that needs to be transmitted through a hidden variable is small, which allows to choose small values for $|H|$. This will cause a reduction of the total number of parameters required to store the functions F and G when compared to non-factorized represenation. Consider for example a pendulum that is not subject to gravity and in which all other external forces are constant in the movement space. In such a setup, it would be sufficient to transmit the difference between the current angle A and the future angle A_{+1} through the hidden variable, because this information would suffice to determine the correct motor command when given an angular speed. Hence, it would be sufficient if the number of hidden states $|H|$ would equal the number of states used to represent the angles. In a real pendulum that is affected by gravity and other angle-dependent forces, the information transmitted through H will be related and therefore, useful numbers of hidden states will have the same magnitude order.

The factorization obtained by introducing a low-dimensional hidden variable brings several benefits: First, the number of parameters can be reduced, which also reduces the storage requirements. Second, functions like the marginal probabilities can be computed more efficiently by belief propagation. Third, a factorized representation can force the model to generalize to unseen data. In the example above, assume a combination of current angle, future angle and speed is observed which was not contained in the training data. As long as the combination of current angle and future angle occurred in the training data, the angle difference transmitted through H would be correct, which would allow to infer the correct motor command.

However, the introduction of a hidden variable makes training more challenging: In an untrained factor graph, the states of the hidden variable have no meaning. Such a meaning must be assigned to the states during training and finding a usefull state assignment can be difficult. In the next section, we will shortly describe possible training strategies.

3.5 Population codes in a factor graph

In section 3.2.4, we introduced the idea of using population codes in a trained factor graph when inferring the state of unobserved variables. We are considering gradient shaped population codes in which the activation of each state increases with decreasing distance to the true observation. The reasoning behind using such codes as inputs to a

factor graph is to obtain linearly interpolated inference results. In an overfitted model, such interpolations could allow to smoothen outputs and thereby help to generalize to unseen data.

3.5.1 Motivation for using population codes

If the individual entities encoding a population code can be in unlimited many states, a population code clearly allows to encode continuous variables at infinite precision. For example, a code using only two units could be used to distinguish all possible settings of a parameter x with values in the intervall $[0, 1]$ by setting the activations of the two units to x and $1 - x$. In practice, encoding resolution is of course limited by each unit's resolving power, but it is obvious that this coding scheme allows to distinguish far more states than a single-unit activation code.

When performing calculations based on population codes, it is however necessary to take the population activity into account in a usefull manner in order to gain an advantage of the higher resolving power. If, for example, only the maximum activity of a population code would be taken into account when performing inference calculations on a factor graph, no benefit would be drawn out of the higher precision of the inputs. A more usefull effect of population coded activity at observed variables in a factor graph would be if the resulting messages in the graph would be weighted averages of those messages which would be obtained for single-state activations. This would allow to turn the discontinuous function represented by a factor graph with discrete states into a continuous one. A piece-wise linear function for example would be obtained if a continuous input would be encoded as a graded activity of the two most similar states, similar to the two-unit example in the last paragraph.

Three benefits can be identified that would result from essentially converting the discontinuous function represented by a factor graph with discrete state variables into a continuous function: First, inference calculations on the resulting continuous function would change smoothly as the originally continuous observations change, which would allow a better approximation of real world phenomena than a discontinuous function. The achievable higher precision, in turn, would allow to reduce the number of states of observed variables and thereby reduce the number of parameters in a factor graph.

Second, the smoothed continuous function might allow to counteract overfitting of a factor graph. The number of parameters in factor nodes that are connected to observed variables increases with their discretization resolution. A large number of parameters in factor nodes can result in overfitting if the training data set is not sufficiently large. In an overfitted factor graph, configurations of the observed variables that are not contained in the training data set are assigned zero probability and as a result, inferred probability

distributions for unobserved variables are flat. For example, a certain configuration of current angle, speed and future angle of a pendulum might not have been contained in the training data. This would be correctly modelled by an overfitted factor graph, but would make it impossible to infer a motor command for such configurations, even if very similar configurations were part of the training data. This problem is to a large degree caused by the fact that a factor graph with discrete states represents a discontinuous function, and thereby allows sudden changes of the function. Population codes can turn this function into a smooth continuous function, which are less prone to extreme overfitting. Expressed in simplified terms, function smoothening resulting from smooth inputs causes the model to consider data points in the training set that were similar but not identical to a specific configuration.

Third, function smoothening could allow a model to make predictions about impossible configurations of the system, as previously described in figure 3.4. For example, if a model is trained to represent one-timestep dynamics with a fixed timestep length, the training data set cannot contain data points describing state transitions that would take longer than this fixed time intervall. Hence, the model could not be used to infer motor commands that would introduce such a transition. Essentially, the function represented by such a factor graph assigns zero probability to all transitions that are not possible within the considered time frame. The flat surface of the function in such parameter regimes makes it impossible to infer a usefull motor command. By smoothening this function with carefully chosen population codes, these impossible transitions can be assigned a non-zero probability and function values will be different for different motor commands, which in turn allows to select a motor command.

3.5.2 Population codes vs. messages in a factor graph

Interpreting input messages as population codes or as probabilistic observations

Unfortunately, population codes cannot be used in a factor graph in a straightforward way because observations passed to a factor graph are interpreted differently from what would be expected intuitively. Consider a simple factor graph with a single two-dimensional factor node as depicted in figure 3.10a) and an input message in which two out of four states are activated by 50%. Interpreted as a population code, this message would correspond to 'the true state of A lies in the middle between states 2 and 3'. The expected inferred message for B would therefore be the average of the two output messages that would be generated for $A = 2$ and $A = 3$.

The message could also be interpreted as a probabilistic statement about the observa-

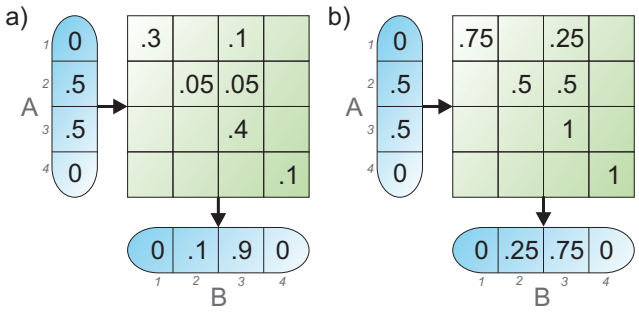

Figure 3.10: *Different interpretations of messages to a factor node. a) Given a factor node representing the joint probability distribution $P(A, B)$ over two variables and an incoming message for A, the sum-product algorithm is used to compute an output message, which is biased by the marginal proabilities $P(A)$. (all empty factor entries are assumed to be 0) b) The same factor node is normalized row-wise and hence represents the conditional probability distribution $P(B|A)$. This removes the effect of the marginal proabilities $P(A)$ in the message to B when applying the sum-product algorithm.*

tion, such that the observed probability that variable A is in state 2 is 50% and the same for state 3. Hence, if $m_{A\to F}(i)$ denotes the ith element of the message from A to the factor node, this value would be interpreted as an observed probability $P_{obs}(A = a_i) = m_{A\to F}(i)$. In this case, the message from the factor node to node B would be expected to be the conditional probability of B given a probabilistic observation about A, in the example $P(B|P_{obs}(A = 2) = 0.5 \wedge P_{obs}(A = 3) = 0.5)$. The message to node B representing this distribution would be expected to equal the expected value of the conditional probability:

$$\begin{aligned} m^{prob}_{F\to B}(j) &= E\left[P(B = b_j|P_{obs}(A = 2) = 0.5 \wedge P_{obs}(A = 3) = 0.5)\right] \\ &= P_{obs}(A = 2) \cdot P(B = b_j|A = 2) + P_{obs}(A = 3) \cdot P(B = b_j|A = 3) \\ &= \sum_i m_{A\to F}(i) \cdot P(B = b_j|A = a_i) \qquad (3.7) \\ &= \sum_i m_{A\to F}(i) \cdot \frac{P(A = a_i, B = b_j)}{P(A = a_i)} \qquad (3.8) \end{aligned}$$

Note the difference between the observed probabilities P_{obs} and the model-intrinsic probabilities P. The latter are defined by the trained factor graph whereas the former refers to a current observation that should be used to infer the state of another variable.

Interestingly, the interpretation of incoming messages as observed probabilities results in the same output message for B as if the message is interpreted as a population code.

Message interpretation in belief propagation

When using standard belief propagation on a factor graph, the described message will be interpreted differently. Here, the meaning of the example message would be 'Variable A is in state 2 or 3, but no further information is available'. Consequently, the prior probabilities for A will be taken into account when computing the message to B.

$$m_{F\to B}(j) = \frac{1}{Z}\sum_i m_{A\to F}(i)\cdot P(A=a_i, B=b_j) \tag{3.9}$$

with the partition function

$$Z = \sum_i \sum_j m_{A\to F}(i)\cdot P(A=a_i, B=b_j) \tag{3.10}$$

It can be seen that equations 3.8 and 3.9 differ solely in the normalization applied, where in the former case, the incoming message is first normalized by the marginal probabilities $P(A)$ and in the latter case, the output message is normalized. The effect of this different normalization can be observed in figure 3.10. In figure 3.10b), the factor node is normalized row-wise and represents the conditional probability distribution $P(B|A)$. Applying the sum-product rule results in the same term as in equation 3.7, which gives the expected result when interpreting the message entries as observed probabilities.

Summarizing, the sum-product algorithm does not yield the output messages that one would expect when interpreting the state of observed variables as population codes or as observed probabilities for the individual states.

3.5.3 Population codes are incompatible with factor graphs.

In the last section, it was shown that message passing on a factor graph might not yield the expected result when multiple states of observed variables are simultaneously activated. The messages to unobserved variables will not be equal to the linear interpolation of those messages that would be received with single-state activation. Instead, these messages would be biased by the prior probabilities of the observed variables. Hence, the output will likely be biased towards observations that were more frequent in the training data set.

It appears like belief propagation on a factor graph representing the JPD is not suited for inferring the probability distribution of unobserved variables when multiple states of observed variables are activated simultaneously. In this section, we will inspect whether it is possible to modify the graph and the message computation such that multi-state activations at observed variables would be interpreted as population codes or as probabilistic

observations. A more formal definition is given in the next section.

Problem definition

Let $\boldsymbol{Y} = \{Y^1, \ldots, Y^{|\boldsymbol{Y}|}\}$ be a set of $|\boldsymbol{Y}|$ observable variables, and let variable Y^i have N_i discrete states, which will be denoted by y^i_1 through $y^i_{N_i}$. We define a multi-state observation for variable Y^i as a vector $\tilde{Y}^i = \{\tilde{y}^i_1, \ldots, \tilde{y}^i_{N_i}\}$, where $\tilde{y}^i_k \in [0, 1]$ denotes the activation of state k of variable Y^i.

Given multi-state observations for all variables except Y^i, we want to be able to compute the conditional probability for Y^i as follows:

$$\begin{aligned} &P(Y^i \,|\, \tilde{Y}^1, \ldots, \tilde{Y}^{i-1}, \tilde{Y}^{i+1}, \ldots, \tilde{Y}^{|\boldsymbol{Y}|}) \\ &\quad = \sum_{\substack{k_1,\ldots,k_{i-1},\\ k_{i+1},\ldots,k_{|\boldsymbol{Y}|}\\ k_j \in \{1,\ldots N_j\}}} \left(\prod_j \tilde{y}^j_{k_j} \right) \cdot P(Y^i \,|\, y^1_{k_1}, \ldots, y^{i-1}_{k_{i-1}}, y^{i+1}_{k_{i+1}}, \ldots, y^{|\boldsymbol{Y}|}_{k_{|\boldsymbol{Y}|}}) \end{aligned} \tag{3.11}$$

We will denote this value as the conditional probability of Y^i given multi-state activations (CPDMS). The definition is equal to the linear interpolation between all conditional probability distributions for Y^i that are obtained when single-state observations are made for all remaining variables. In this interpolation, each of the conditional probability distributions is weighted by the product of the corresponding activations. If the multi-state activations are proportional to observed state probabilities, the above definition equals the expected value of the conditional probability distribution for Y^i.

We would like to be able to compute the CPDMS for every variable in $\boldsymbol{Y}$ in an efficient manner by using message passing.

General considerations

Assume a model is given that allows the computation of the JPD. In order to compute the CPDMS for a variable Y^1, we can rewrite 3.11 as follows:

$$P(Y^1 \mid \tilde{Y}^2, \ldots, \tilde{Y}^{|\boldsymbol{Y}|}) = \sum_{\substack{k_2,\ldots,k_{|\boldsymbol{Y}|} \\ k_i \in \{1,\ldots N_i\}}} \left(\prod_i \tilde{y}^i_{k_i} \right) \cdot P(Y^1 \mid y^2_{k_2}, \ldots, y^{|\boldsymbol{Y}|}_{k_{|\boldsymbol{Y}|}}) \tag{3.12}$$

$$= \sum_{\substack{k_2,\ldots,k_{|\boldsymbol{Y}|} \\ k_i \in \{1,\ldots N_i\}}} \left(\prod_i \tilde{y}^i_{k_i} \right) \cdot \frac{P(Y^1, y^2_{k_2}, \ldots, y^{|\boldsymbol{Y}|}_{k_{|\boldsymbol{Y}|}})}{P(y^2_{k_2}, \ldots, y^{|\boldsymbol{Y}|}_{k_{|\boldsymbol{Y}|}})}$$

$$= \sum_{\substack{k_2,\ldots,k_{|\boldsymbol{Y}|} \\ k_i \in \{1,\ldots N_i\}}} \left(\prod_i \tilde{y}^i_{k_i} \right) \cdot \frac{P(Y^1, y^2_{k_2}, \ldots, y^{|\boldsymbol{Y}|}_{k_{|\boldsymbol{Y}|}})}{\sum_{k_1} P(y^1_{k_1}, y^2_{k_2}, \ldots, y^{|\boldsymbol{Y}|}_{k_{|\boldsymbol{Y}|}})} \tag{3.13}$$

Equation 3.13 can be computed if the joint probability distribution is accessible, independent of its representaion, for example through a factor graph or a bayesian network. This approach was used in section 3.5.2, where a factor graph consisting of a single two-dimensional factor node was inspected: In order to compute the CPDMS for one variable given the other one, the factor nodes needed to be normalized along the output dimension.

The same approach can be extended to multi-dimensional factor nodes: For each output direction, a differently normalized factor is used in order to compute the output message using the sum-product algorithm. After normalization, these factor nodes will contain the conditional probabities for their respective output variable. Figure 3.11 depicts the effect of these transformations on message passing: A factor graph with a single factor node F is transformed into a message passing graph with three nodes F_A, F_B and F_C replacing the single factor node F, where $F_A(a,b,c) = F(a,b,c)/\sum_a F(a,b,c)$ and F_B and F_C are defined analogous. In effect, the factors represent the conditional probability distributions for their respective output variable. Computing the CPDMS for a variable in this transformed graph can be done using the sum-product rule and applying multi-state activation vectors as input messages for the remaining variables.

In this example, the three factors F_A, F_B and F_C can be directly derived from the original factor node, which would make it possible to use existing learning methods to train the factor node and only transform it whenever the CPDMS for a variable should be computed.

Intuition suggests that the same approach would extend to factor graphs composed of many factor nodes and hidden variables: Replace each factor connecting n variable nodes by n factor nodes, where each node computes the messages to a one of the variables. An example for such a transformation is depicted in figure 3.12. In this example, two undirected factor nodes f and g are each replaced by three directed factors that are each normalized with respect to their output direction, for example $f_1(Y^1, Y^2, H^1) =$

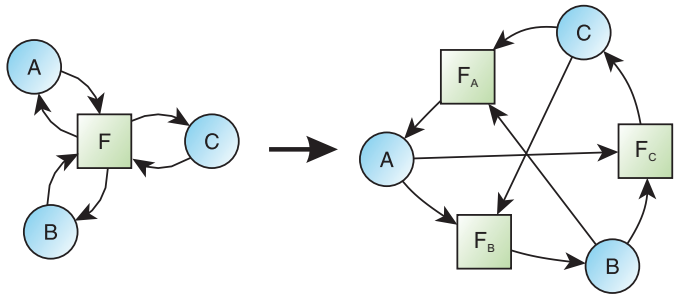

Figure 3.11: *Transforming a factor graph with a single factor node of degree 3 for computing the CPDMS of all variables: The original Factor F, representing the JPD $P(A, B, C)$ is normalized along different dimensions in order to yield the three factors $F_A = P(A \mid B, C)$, $F_B = P(B \mid A, C)$ and $F_C = P(C \mid A, B)$. Using belief propagation with the sum-product algorithm on the right graph will result in messages that are equal to the CPDMS.*

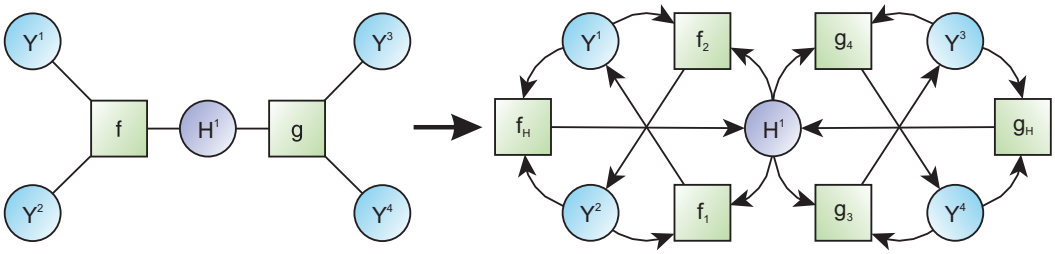

Figure 3.12: *Attempt to transform a factor graph with two factor nodes in order to enable the computation of the CPDMS for observed variables. Both factor nodes are transformed similar to the transformation in figure 3.11.*

$f(Y^1, Y^2, H^1) / \sum_i f(y_i^1, Y^2, H^1)$ and $g_H(Y^3, Y^4, H^1) = g(Y^3, Y^4, H^1) / \sum_i g(Y^3, Y^4, h_i^1)$

Assume that the original factor graph can be used to compute the JPD over Y^1, Y^2, Y^3 and Y^4:

$$P(Y^1, Y^2, Y^3, Y^4) = \frac{1}{Z} \cdot \sum_i f(Y^1, Y^2, h_i^1) \cdot g(Y^3, Y^4, h_i^1) \tag{3.14}$$

Can belief propagation on the transformed graph be used to compute the CPDMS for one of the variables, for example Y^1? When multi-state observations $\tilde{Y}^2$, $\tilde{Y}^3$ and $\tilde{Y}^4$ are used as input messages to the remaining variables, the sum product rule yields the following

message to node Y^1:

$$m_{f_1 \to Y^1} = \sum_{k_2} \sum_{k_3} \sum_{k_4} \sum_{k_H} \tilde{y}^2_{k_2} \tilde{y}^3_{k_3} \tilde{y}^4_{k_4} \cdot f_1(Y^1, y^2_{k_2}, h^1_{k_H}) \cdot g_H(y^3_{k_3}, y^4_{k_4}, h^1_{k_H}) \tag{3.15}$$

$$= \sum_{k_2} \sum_{k_3} \sum_{k_4} \tilde{y}^2_{k_2} \tilde{y}^3_{k_3} \tilde{y}^4_{k_4} \cdot \underbrace{\sum_{k_H} \frac{f(Y^1, y^2_{k_2}, h^1_{k_H})}{\sum_{k_1} f(y^1_{k_1}, y^2_{k_2}, h^1_{k_H})} \cdot \frac{g(y^3_{k_3}, y^4_{k_4}, h^1_{k_H})}{\sum_i g(y^3_{k_3}, y^4_{k_4}, h^1_i)}}_{x} \tag{3.16}$$

While the result looks similar to the CPDMS as defined in 3.13, the term x does not equal the conditional probability for Y_1, which can be verified by calculating the expected term:

$$P(Y^1 \,|\, y^2_{k_2}, y^3_{k_3}, y^4_{k_4}) = \frac{P(Y^1, y^2_{k_2}, y^3_{k_3}, y^4_{k_4})}{\sum_{k_1} P(y^1_{k_1}, y^2_{k_2}, y^3_{k_3}, y^4_{k_4})} \tag{3.17}$$

$$= \sum_{k_H} \frac{f(Y^1, y^2_{k_2}, h^1_{k_H}) \cdot g(y^3_{k_3}, y^4_{k_4}, h^1_{k_H})}{\sum_{k_1} \sum_i f(y^1_{k_1}, y^2_{k_2}, h^1_i) \cdot g(y^3_{k_3}, y^4_{k_4}, h^1_i)} \tag{3.18}$$

The term x from 3.15 and the term in 3.18 contain slightly different denominators, which cannot easily be transformed into each other, because the latter cannot be factorized.

This example illustrates a problem with the definition of the CPDMS: The quotient in 3.13 prevents the application of the distributive law, which is the basic principle underlying message passing schemes such as belief propagation. The calculation of the CPDMS using 3.13 would therefore force the separate calculation of all $\prod_{i \neq 1} N_i$ terms (which arise as a result of the sums over $\{k_2, \ldots, k_{|\boldsymbol{Y}|}\}$).

Application of the distributive law might become possible if the conditional probability distribution $P(Y^1 \,|\, Y^2, \ldots, Y^{|\boldsymbol{Y}|})$ can be computed as a sum of products of local functions of low dimensionality. In other words: A factor graph can be found that represents a factorization of $P(Y^1 \,|\, Y^2, \ldots, Y^{|\boldsymbol{Y}|})$. In some cases, this factor graph could be a subgraph of a factor graph representing the JPD. In the following, an example for such a distribution will be given.

A factorized conditional probability distribution

Consider a bayesian network as depicted in figure 3.13a) and its factor graph representation in b). Here, the joint probability distribution is given as follows:

$$P(Y^1, Y^2, Y^3, Y^4) = \sum_{k_H} P(Y^1) \cdot P(Y^2) \cdot P(h^1_{k_H} \,|\, Y^1, Y^2) \cdot P(Y^3) \cdot P(Y^4 \,|\, h^1_{k_H}, Y^3) \tag{3.19}$$

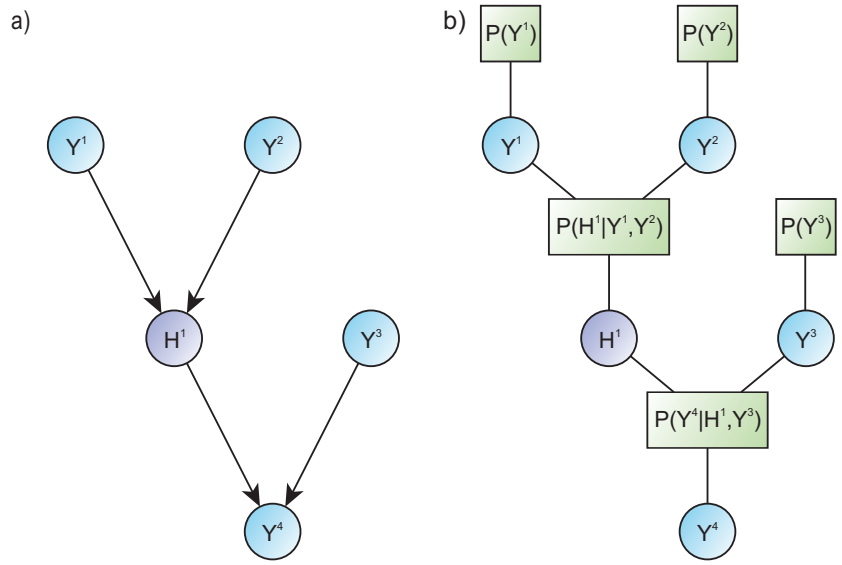

Figure 3.13: *a) A simple bayesian network with 4 observed variables and 1 hidden. b) The factor graph representation of a).*

Note that the variables Y^1, Y^2, and Y^3 are independent of each other. The conditional probability distribution for Y^4 can be written as a sum of products of functions that are contained in the JPD:

$$P(Y^4 \,|\, Y^1, Y^2, Y^3) = \sum_{k_H} P(h^1_{k_H} \,|\, Y^1, Y^2) \cdot P(Y^4 \,|\, h^1_{k_H}, Y^3) \tag{3.20}$$

This form of the conditional probability distribution allows the application of the distributive law when computing the CPDMS for Y^4:

$$P(Y^4 \,|\, \tilde{Y}^1, \tilde{Y}^2, \tilde{Y}^3) = \sum_{k_1,k_2,k_3} \tilde{y}^1_{k_1} \tilde{y}^2_{k_2} \tilde{y}^3_{k_3} \sum_{k_H} P(h^1_{k_H} \,|\, y^1_{k_1}, y^2_{k_2}) \cdot P(Y^4 \,|\, h^1_{k_H}, y^3_{k_3}) \tag{3.21}$$

$$= \sum_{k_H,k_3} \tilde{y}^3_{k_3} \cdot P(Y^4 \,|\, h^1_{k_H}, y^3_{k_3}) \underbrace{\sum_{k_1,k_2} \tilde{y}^1_{k_1} \tilde{y}^2_{k_2} \cdot P(h^1_{k_H} \,|\, y^1_{k_1}, y^2_{k_2})}_{m_{P(H^1 \,|\, Y^1, Y^2) \to H^1}} \tag{3.22}$$

The right hand side of the equation illustrates the advantage of using the distributive law: By separating the sums over k_1 and k_2 from the sums over k_H and k_3, the effective number of calculations that need to be performed is reduced. The result equals the message to node Y^4 when belief propagation is applied to the subgraph of 3.13b) where the factor nodes containing the prior probabilities $P(Y^1)$, $P(Y^2)$ and $P(Y^3)$ have been replaced by the multi-state activation vectors of the respective variables. The brace indicates the downstream message that would be sent to node H^1

Computing the CPDMS of several variables using the same graph

While the CPDMS for Y^4 in the last example can be calculated using belief propagation on the described subgraph, the message that would be sent to node Y^3 does not equal its CPDMS:

$$m_{P(Y^4 \mid H^1, Y^3) \to Y^3} = \sum_{k_H, k_4} \tilde{y}^4_{k_4} \cdot P(y^4_{k_4} \mid h^1_{k_H}, Y^3) \sum_{k_1, k_2} \tilde{y}^1_{k_1} \tilde{y}^2_{k_2} \cdot P(h^1_{k_H} \mid y^1_{k_1}, y^2_{k_2}) \tag{3.23}$$

$$= \sum_{k_1, k_2, k_4} \tilde{y}^1_{k_1} \tilde{y}^2_{k_2} \tilde{y}^4_{k_4} \underbrace{\sum_{k_H} P(h^1_{k_H} \mid y^1_{k_1}, y^2_{k_2}) \cdot P(y^4_{k_4} \mid h^1_{k_H}, Y^3)}_{\neq P(Y^3 \mid y^1_{k_1}, y^2_{k_2}, y^4_{k_4})} \tag{3.24}$$

As indicated, the last term does not equal the conditional probability distribution $P(Y^3 \mid y^1_{k_1}, y^2_{k_2}, y^4_{k_4})$, which would be the case if the message were equal to the CPDMS for Y^3. Apparently, the graph needs to be changed to enable the correct computation of this distribution. In the given example, the required change is simple: By replacing the factor $P(Y^4 \mid H^1, Y^3)$ with the conditional probability distribution $P(Y^3 \mid Y^4)$, the message to Y^3 that was computed above would become equal to the desired CPDMS. In order to store both $P(Y^4 \mid H^1, Y^3)$ and $P(Y^3 \mid Y^4)$, the factor node could simply store the JPD $P(Y^3, Y^4, H^1)$, which allows to derive both conditional probability distributions by normalizing along different directions.

However, the example graph discussed above was derived from a simple bayesian network and assumed that three out of four variables were independent of each other. This property was required in order to compute the CPDMS of Y^4 and Y^3 through message passing on a factor graph that was only slightly modified compared to the graph containing the JPD.

Figure 3.14 shows a more general graph that assumes no independences between the observed variables. The figure shows only two factor nodes f and g, where the latter is a placehoder for an arbitrary graph structure. We will inspect how the CPDMS for Y^1 and Y^2 can be computed based on messages in such a general graph.

Preferably, the desired distributions should be computable from the same message arriving at f from node H^1. Based on the last example graph, we showed already that depending on the output direction, different functions might need to be applied at the node f in order to compute the correct CPDMS for different variables. Let $\alpha(Y^1, Y^2, H^1)$ be the function used to compute the CPDMS for Y^1 and $\beta(Y^1, Y^2, H^1)$ the corresponding function for Y^2. After calculating the CPDMS for both variables as in equation 3.21, we

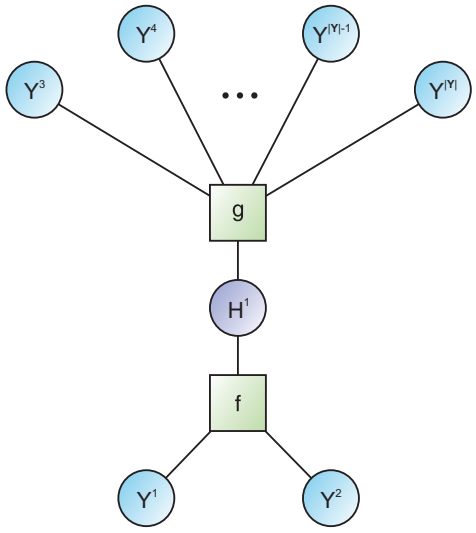

Figure 3.14: *A very general factor graph. The factor g stands representative for an arbitrary graph structure that might contain many more hidden variables and factor nodes.*

get the following conditions:

$$P(Y^1 \,|\, Y^2, Y^3, \ldots, Y^{|\boldsymbol{Y}|}) = \sum_{k_H} \alpha(Y^1, Y^2, h^1_{k_H}) \cdot g(Y^3, \ldots, Y^{|\boldsymbol{Y}|}, h^1_{k_H}) \tag{3.25}$$

$$P(Y^2 \,|\, Y^1, Y^3, \ldots, Y^{|\boldsymbol{Y}|}) = \sum_{k_H} \beta(Y^1, Y^2, h^1_{k_H}) \cdot g(Y^3, \ldots, Y^{|\boldsymbol{Y}|}, h^1_{k_H}) \tag{3.26}$$

A solution can be found by making the following assumption:

$$P(Y^1, H^1 \,|\, Y^2, Y^3, \ldots, Y^{|\boldsymbol{Y}|}) = \alpha(Y^1, Y^2, H^1) \cdot g(Y^3, \ldots, Y^{|\boldsymbol{Y}|}, H^1) \tag{3.27}$$

$$P(Y^2, H^1 \,|\, Y^1, Y^3, \ldots, Y^{|\boldsymbol{Y}|}) = \beta(Y^1, Y^2, H^1) \cdot g(Y^3, \ldots, Y^{|\boldsymbol{Y}|}, H^1) \tag{3.28}$$

Which implies:

$$\begin{aligned} \frac{\alpha(Y^1, Y^2, H^1)}{\beta(Y^1, Y^2, H^1)} &= \frac{P(Y^1, H^1 \,|\, Y^2, Y^3, \ldots, Y^{|\boldsymbol{Y}|})}{P(Y^2, H^1 \,|\, Y^1, Y^3, \ldots, Y^{|\boldsymbol{Y}|})} = \frac{P(Y^1, Y^3, \ldots, Y^{|\boldsymbol{Y}|})}{P(Y^2, Y^3, \ldots, Y^{|\boldsymbol{Y}|})} \\ &= \frac{\sum_{k_2} P(Y^1, y^2_{k_2}, Y^3, \ldots, Y^{|\boldsymbol{Y}|})}{\sum_{k_1} P(y^1_{k_1}, Y^2, Y^3, \ldots, Y^{|\boldsymbol{Y}|})} \end{aligned} \tag{3.29}$$

Here, the ratio between α and β is a function of all observed variables. Hence, at least one of the two functions cannot be a local and low-dimensional function as desired (in fact, the input space of these function would need to be changed). It follows that the node f would have to have connections to all observed variable nodes.

Without the assumptions made in 3.27 and 3.28, it might be possible to find local

functions for α and β that depend only on Y^1, Y^2 and H^1. This, however, would require that the message sent from H^1 to f contains enough information about the current state of all observed variables connected to g for computing both $P(Y^1 \,|\, Y^2, \ldots, Y^{|\boldsymbol{Y}|})$ and $P(Y^2 \,|\, Y^1, Y^3, \ldots, Y^{|\boldsymbol{Y}|})$.

The conditions given in equations 3.25 and 3.26 require the separation of each of the two conditional probability distributions into two functions after introducing one hidden variable. With a very high number of states assigned to the hidden variable, this factorization allows to represent any function. However, the minimal number of states for H^1 required to sufficiently approximate a specific function depends on the function and the chosen separation of the parameter set. While the chosen factorization in the example might allow to encode one of the conditional probability distributions with a very low number of hidden states, this does not imply any boundary on the number of hidden states required for encoding the second distribution. This results from the complex ratio between the two distributions, which is equal to the ratio calculated in 3.29.

This insight remains valid even when removing the assumption that the function g remains constant independent of whether it is used to compute the CPDMS for Y^1 or for Y^2: A specific separation of the parameter set into two sets might be efficienct for encoding one conditional probability distribution, but could require a very large amount of hidden states in order to encode a second distribution. The problem gets worse if the number of variables that are connected to the node f in figure 3.14 is increased: The more variables are connected to f, the less likely it will become that a small number of hidden states will be sufficient for factorizing all conditional probability distributions of variables connected to f based on the same variable set separation.

Of course, there exist joint probability distributions that allow an efficient factorization of all possible conditional probability distributions with the same separation of the variable set. One trivial example are distributions where all variables are independent of each other. It remains an open question though whether there exist JPD's which do not contain many independences but still allow to express all CPD's using sets of functions defined on the same subsets of the parameter set.

For arbitrary distributions however, it seems like the number of states of hidden variables would have to be set to large values in order to allow the transmission of all information necessary to compute all conditional probability distributions. The increase of the overall number of adjustable parameters would render such an approach infeasible. Instead, it would probably be more efficient to create a separate factorization for each conditional probability distribution. A single undirected factor graph representing the JPD would have to be replaced by $|\boldsymbol{Y}|$ directed graphs, which would have to be trained separately. This, in turn, would be inappropriate for large numbers of variables, because

the number of local functions would scale quadratically with the number of variables.

3.6 Function smoothing without population codes

In the last section, we showed that it is infeasible to use message passing schemes for the computation of conditional probability distributions when given multi-state activations for observed variables. Messages to unobserved variables in a factor graph will be biased by the prior probabilities of observed variables, and these effects are difficult to remove unless a directed graph is used, which allows a message-based computation of the conditional probability distribution of only a single variable.

The original motivation to introduce population codes was described in section 3.5.1: If the population coded activity could be properly taken into account in calculations on a factor graph, the discontinuous function represented by a factor graph with discrete states would effectively be transformed into a interpolated continuous function. This, in turn, promised to yield three advantages: First, the interpolated function could have provided a more precise model of real world phenomena than a discontinuous function, which would have solve. Second, function smoothening could have helped to counteract overfitting. Third, function smoothening could have allowed to select motor commands even when the desired transition cannot be obtained instantaneously.

As discussed, the use of population codes in belief propagation on a factor graph introduces severe problems and as a consequence, results from inference calculations are biased towards observations which were more frequent in the training data set. Experiments showed that this disadvantage outweighs the possible advantages described above. As such, three problems remain unsolved: First, factor graph based control has a low precision if continuous observed parameters of a robotic system are discretized into variables with few states. Second, factor graphs tend to overfit to training data if observed variables have too many states. Third, a factor graph representing instantaneous dynamics does not allow to select motor commands if a desired transition cannot be performed instantaneously.

The third point relates to a fundamental problem of the architecture we chose to solve a planning task and its usage: A single iteration on a model representing the instantaneous dynamics of a robotic system does not suffice to make predictions about future changes of the system and therefore cannot be used to plan sequences of movements. In chapter 4, we will introduce a novel kind of graphical modesl which overcome the limitations of models representing instantaneous dynamics and can be used to execute such sequences of actions in order to reach a goal.

The remaining two problems relate to the granularity of discretization, which will be

discussed further below.

3.6.1 Choosing appropriate variable dimensions

The most obvious solution to avoid overfitting in factor graphs with discrete state variables is to reduce the dimensionality of observed variables. In a setup as depicted in figure 3.9, two three-dimensional factors are used. In this setup, the hidden variable H is used to transmit all important information about the combination of current angle and future angle to another node which will determine a motor command based on the current angular speed. As described in section 3.4, this information will be related to the angle difference between the current and the future state. While it would be possible to encode this information in a message with very few states, experiments showed that a much better performance is achieved if the resolution of the hidden variables is in the same order as the resolution of the observed variables. The number of parameters in the factor node F should therefore scale proportional to the third power of the chosen angle resolution.

With only approximately 10^4 available training datapoints, the angular resolution should be chosen around 10-40 states in order to avoid overfitting of this single factor. Similar considerations apply to the second factor G. Extensive experiments have confirmed that these settings yield the best control performance. Best results were obtained if the current and the target angle were discretized at a resolution of 36 states, and motor command and speed at resolutions of 21 and 11 states, respectively. The resolution of the hidden variable had little influence on performance, as long as its state space comprised between 10 and 30 states. Interestingly, none of the tested factorized representations was able to outperform the simple approach using population codes on a factor graph representing the conditional probability distribution for the motor command. Peak performance was at a hidden variable resolution of 11 states, which resulted in an average angle deviation of 12.5°.

It should be noted that the computational time required for running belief propagation on a factor graph scales linearly with the number of parameters in the factor nodes. When using the expectation maximization algorithm, a single training run requires as many belief propagation runs as the number of data points in the training set (see section 3.3.2). If both the number of data points and the number of parameters are in the range of 10^4, each training run will require a multiple of 10^8 calculations. (The required number of clock cycles will be about 50 times higher than this value because the sum product rule requires additional overhead and messages for several directions need to be calculated.) Execution of several hundreds of training runs therefore quickly exceed several hours if the resolution of variables is large. The maximum resolution for variables in a factor graph is therefore also limited by practical considerations.

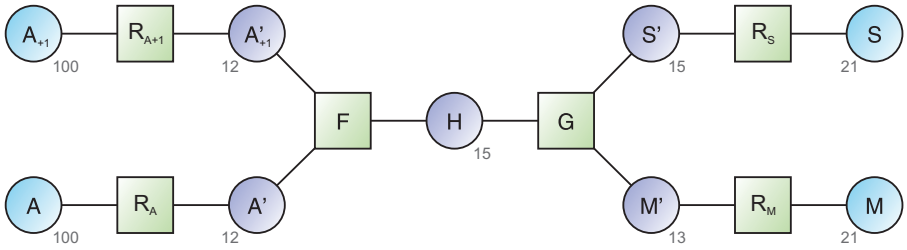

Figure 3.15: *An factor graph similar to the example from figure 3.9, but with additional mapping nodes R_x, which reduce high-dimensional observed variables to low-dimensional hidden variables. (Small numbers indicate the number of states of each variable.) The mapping nodes allow to translate a code using single-state activations for the observed variable into a multi-state activation code for the hidden variables.*

3.6.2 Further factorization

In order to allow observed variables with high resolution, three-dimensional factors connecting at least two of the observed variables must be avoided, because these would contain too many parameters. Several factor graphs were therefore trained which contained only two-dimensional factor nodes. These setups, however, require the introduction of hidden variables with degree bigger than 2 in order to create a factor graph connecting all observed variables. Training of factors that are attached to such hidden variables is often slow because several factor nodes must agree to associate a specific meaning to the individual states of the hidden variable. In addition, all graphs with hidden variables of degree 3 or higher that we tested on control tasks performed poorly. Therefore, we focused our analysis on graphs with multiple factor nodes of degree 3 and hidden variable nodes with degree 2.

Using mapping factor nodes to reduce variable resolution

High dimensional observed variables can also be mapped to lower dimensional hidden variables by using factor nodes of degree 2. Figure 3.15 shows an example factor graph that uses this idea in order to map the four observed variables to hidden variables with fewer states. This approach was motivated by the impossibility of using population codes in a factor graph: The mapping nodes (or reduction nodes) R_x can be used to compute a low-dimensional population code from a single-state activation of a high-resolution input variable and could therefore act like encoders for population codes. Unlike discussed in previous sections, gradient shaped population codes would not be used as inputs to a factor graph, but could instead arise internally inside the graph. Since the mapping nodes are also subject to training algorithms and will therefore become a part of the modelled joint probability distribution, problems discussed in previous sections cannot arise in this

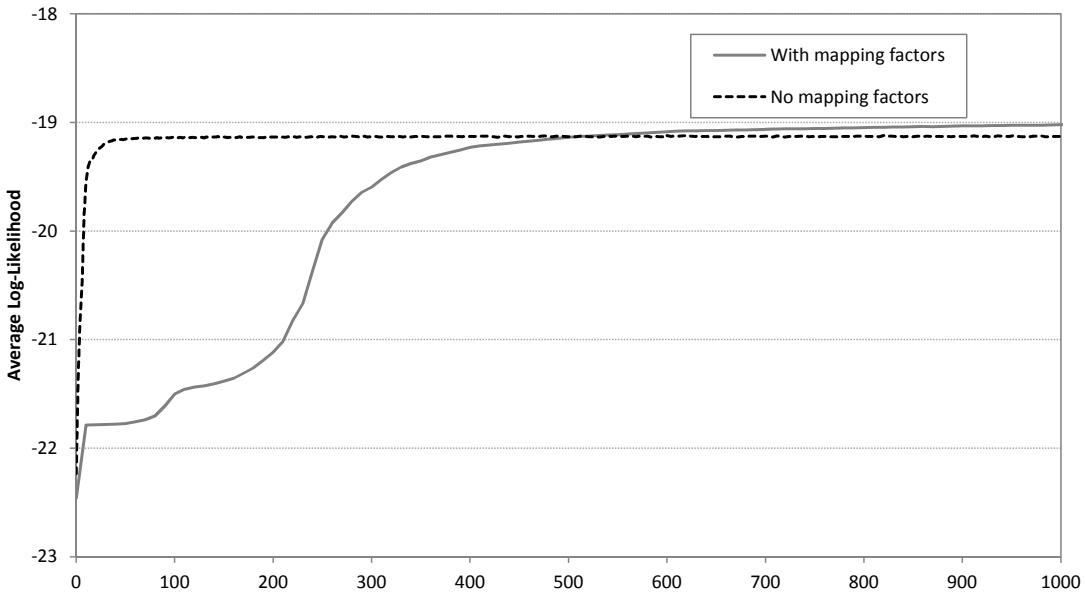

Figure 3.16: *Evolution of log-likelihood with training progress. The log-likelihood of the training data monotonously increases in dependence of training runs. Factor graphs with a single hidden variable (black dashed line) converge significantly faster than after the addition of mapping nodes (grey line) together with 4 additional hidden variable nodes. The addition of mapping nodes allows a slightly better representation of the training data.*

approach.

Figure 3.16 shows the training success for such a network, compared to training of a graph without mapping nodes. Due to the increased number of hidden variables, the average log-likelihood of the training data converges much slower, but finally to a slightly higher log-likelihood. However, this improved representation of the training data was not reflected in terms of system performance.

A stronger effect on the log-likelihood of the data could be seen for data from a simulated pendulum with increased gravity constant: Data obtained from this system was biased towards angles in the range between $-90°$ and $90°$ (lower half of the pendulum's reach). The mapping nodes allowed to reflect this bias by adapting the mapping resolution depending on the density of the data. This effect is illustrated in figure 3.17. It resulted in a significantly higher final log-likelihood of the data than without mapping nodes as shown in figure 3.18.

When the control performance of this model was compared to a model without mapping factors in a test where target angles were evently distributed, it performed worse on average. This decrease in performance can be attributed to higher deviations when target angles were set to be in the upper half of the pendulum's reach. This effect would be

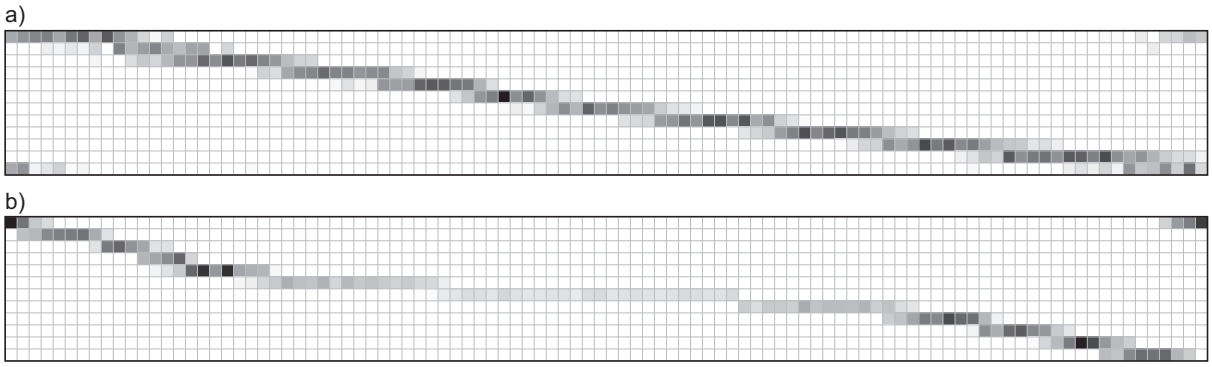

Figure 3.17: *Contents of a mapping node (node R_A in figure 3.15 connecting a high--dimensional representation of the angle and a low-dimensional hidden variable, after training with data from a real pendulum (a) or a simulated pendulum (b). Darkness of each square indicates the relative connection strength between a high-precision angle (horizonatal) and a low-precision hidden representation of the angle (vertical). In the simulated pendulum, more states*

expected since this regions is represented less precisely by the model containing mapping factors. Disappointingly, however, the performance for target angles in the lower half of the pendulum's reach was also not significantly better when using mapping nodes. Yet, the performance of the simpler model for such target angles was already very good and the performance limiting factor in this region was most likely the resolution for the motor command. Since motor commands were evenly distributed in the training data, the mapping factor for the motor command assigned nearly the same resolution to all possible motor commands.

3.7 Factorized representation of robot kinematics

In this section, we will consider a more complex robotic system with four degrees of freedom. We describe an approach to solving a simple control task on this system, which uses a factor graph representing the robot's kinematics for computing its inverse kinematics. Unlike in methods described in the previous sections, the dynamics of the robotic system are not modelled in this approach. Instead, a target configuration of the robotic system is computed as a function of cartesian coordinate and the robot's joint angles are then linearly moved towards this target configuration.

3.7.1 An industrial robotic arm

Figure 3.19 shows a Katana robotic arm, manufactured by Neuronics [Neu11] with 6 degrees of freedoms. Here, we will not consider the state or angle of the gripper, which

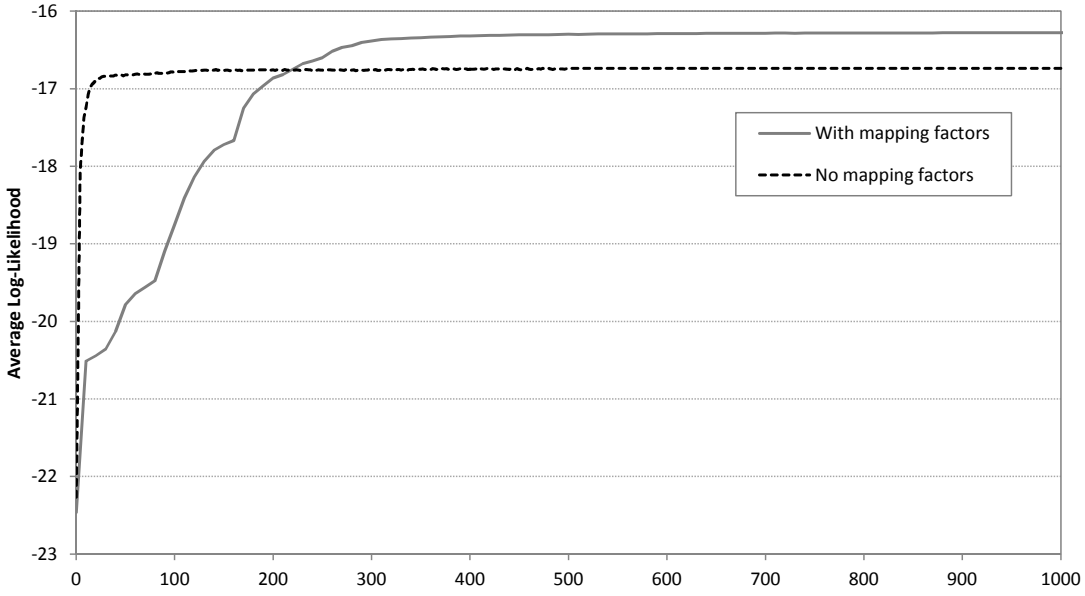

Figure 3.18: *Evolution of log-likelihood with training progress for a simulated pendulum. Shown is the evolution of the log-likelihood of the training data as a function of training rounds. This plot is similar to figure 3.16, but for a simulated pendulum with high gravity constant. Here, a factor graph with mapping nodes (grey line) allows the creation of a significantly better model than without mapping nodes (black dashed line).*

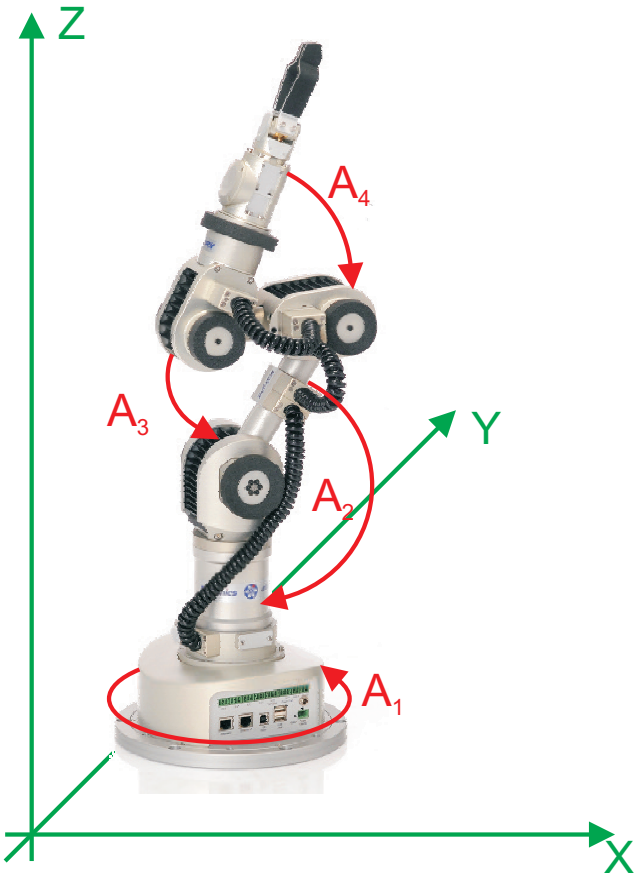

Figure 3.19: *A robotic arm with six degrees of freedom. In this project, only the four indicated degrees of freedom were taken into account, the gripper status (opened/closed) as well as the gripper rotation were ignored. Hence, the robotic arm can be regarded as a system with 4 degrees of freedom, namely the angles A_1, A_2, A_3 and A_4. The robot software provided functions to compute the cartesian coordinate (x, y, z) of the tip of the center of the gripper.*

effectively reduces the robot to a system with 4 degrees of freedom. Each of the axes is equipped with a high precision servo motor which is controlled by a microcontroller. The microcontrollers allow to perform linear movements to a desired configuration, specified as a set of desired angles.

3.7.2 Control task

A control task was chosen which is similar to the task we described for the compliant pendulum arm in section 3.2.2: Given a position in cartesian coordinates, move the center of the robot's gripper to this position by modifying the joint angles. For simplicity, the pointing direction of the gripper can be ignored.

A crucial difference distinguishes this task from the pendulum-task: While the motor

of the pendulum allowed to modify the torque acting on the pendulum shaft and thereby allowed compliant control, the microcontrollers for the motors of the robotic arm implement a servomechanisms and will automatically compensate all external forces, which results in non-compliant control. As a consequence, a configuration change to a set of target angles is largely simplified, because the servo controllers will generate a linear movement towards this set of angles. To a large degree, this simplified control renders it unnecessary to consider the dynamics of the robotic system, because normally, the output that needs to be computed for a given target position in cartesian position is independent of the current state of the system.

However, care needs to be taken in order to avoid collisions: A linear adjustment of target angles towards a target position might cause the robot to collide with itself or the ground floor. We use a simple filter function, which will stop any movement that would cause such collisions.

3.7.3 An iterative control algorithm based on a factor graph

Our strategy to solve the described control task is based on a factor graph representing the joint probability distribution $P(X, Y, Z, A_1, A_2, A_3, A_4)$, which describes the relation between discretized cartesian coordinates and discretized joint angles of the robotic system, thus its kinematics. Figure 3.20 shows an example factor graph composed of four factor nodes that are connected to each other with two additional hidden variable nodess, which was successfully used for representing the JPD. Factor nodes were trained using the expectation maximization algorithm and based on data, which was generated under human control. For each data point, angles were read out directly from the robot's sensors and a cartesian coordinate was calculated using a software library provided by the robot's manufacturer.

Given a trained factor graph representing the JPD $P(X, Y, Z, A_1, A_2, A_3, A_4)$, our control algorithm can be outlined as follows:

1. Accept a target position defined in cartesian coordinates as input.
2. Read out the current position of angles.
3. Clamp the variable nodes of the factor graph to the observed angles and the input coordinates.
4. Run belief propagation and read out the messages arriving at each node representing an angle and decode these messages into target angles.
5. Start a movement towards the configuration defined by the set of decoded target angles by linearly changing the angles at the axes.

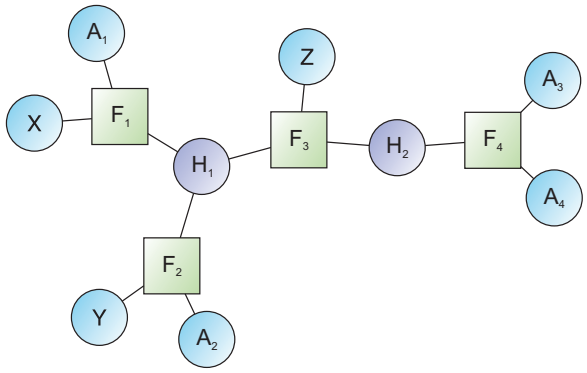

Figure 3.20: *A simple factor graph for representing kinematics of a robot arm. Two hidden variables H_1 and H_2 are introduced in order to limit the maximum node degree to 3.*

6. Interrupt movement after 300 ms and repeat control procedure by returning at step 2.

Computing target angles

In steps 3 and 4, the factor graph representing the JPD $P(X, Y, Z, A_1, A_2, A_3, A_4)$ is used to compute a set of target angles. This computation is less straightforward than it might seem at first sight, which is why its details need further inspection.

First, assume that the target position and the target angles would all be encoded as messages in which a single element is different from zero. The state of the respective variables would thereby be fixed to the observed value. After belief propagation, the message arriving at a node representing the angle A_x would then encode the conditional probability distribution of A_x given the current configuration of the remaining 3 angles and the desired cartesian coordinate. In many cases however, the current configuration of the other three angles and the target position are conflicting and simply changing the angle A_x is not sufficient to reach the desired position. Hence, the conditional probability distribution for A_x would be flat, because none of the possible angles will cause the joint probability to be greater than zero. Consequently, it would be impossible to choose a target angle for A_x.

This problem reflects the trivial insight that the robotic arm will be unable to reach every possible possible position by adjusting a single angle. Obviously, it is wrong to assume that all angles except the angle A_x were fixed to the current value. Instead, it could be assumed that no prior information exists about these angles, which could be encoded as flat input messages to the respective variable nodes. However, the consequence would be that each angle would be adjusted independent of the current configuration. The messages arriving at a node A_x would equal the marginal distribution for that angle

given only the cartesian coordinate. A single target value could then be drawn from this distribution but it would be independent of the current configuration. Such a method could cause the robot arm to make inefficiently long movements.

Therefore, we use a gradient shaped population code to define the input message for each observed angle. All elements of the input message are activated to some degree, but the activation will be stronger the more similar the corresponding variable state is to the currently observed angle. In this way, output messages of the factor graph will be biased towards configurations that are similar to the current configuration of the angles.

As described in section 3.5.3, the resulting output messages will not equal the expected value of the conditional probability distribution, but will be biased to configurations which were observed more frequently in the training data. As a result, inferred angles will be less precise. However, we found this problem to be less troublesome in the setup considered here: Since the cartesian target position is always defined as a messages with single-state activation, there is less bias towards robot configurations which do not match the desired position. Performance diminishes if the target is also defined in terms of a gradient shaped population code.

Figure 3.21 shows a screenshot of the software FGControl, which contains visualizations of the inputs to and the output messages from the observed variable nodes in a factor graph modelling the JPD.

Obtaining smoother motions

In an early version of the algorithm, a new goal was for the robot arm was selected every 50 ms. However, stopping an ongoing motion and initiating a new motion causes the robot arm's motions to become stuttery, independent of whether the new motion is almost identical to the last one. Therefore, the control algorithm was slightly adapted. Simply increasing the time intervall to 300 ms resulted in unesthetic sudden changes of motion direction and poor convergence towards the goal, because the angles of the system changed too much between measurements and likewise, the computed target angles changed radically.

A simple idea allowed to improve the algorithm, which resulted in much smoother movements of the robotic arm: After step 4 in the algorithm described above, a target set of angles is known, which will then lead to a short movement towards the corresponding configuration for a time of 300 ms short time. In the considered system, the angle change within this time is deterministic. Therefore, the set of current angles that will be measured at the next iteration of the algorithm can be predicted. Moreover, the outcome of the factor graph calculation at the next iteration can also be computed, because the target coordinate will remain the same.

In order to achieve smoother movements, we take advantage of this insight and iterate 5 times over steps 3 and 4, before finally selecting a target angle. In the first iteration, the currently observed set of angles is used as an input to the factor graph. A target set of angles is then computed as described in the last section. In the next iteration, it is assumed that a total angle change of 17° would have been performed, starting at the current configuration of the robot. The result of this iteration will be a refined output compared to the first iteration and will then be used as an input for the third iteration and so on. After the fifth iteration, a movement is then initiated towards the target angle set computed in the last run.

3.7.4 Simulations

The control algorithm for the Katana robot arm was tested in various setups. Approximately 20000 data points which were recorded under human control were used as training data. Different factor graph topologies were tested, with most of them yielding similar performace. In all these factor graphs, observed variables were discretized into 10 states. Figures 3.22 and 3.23 show two typical traces from live control tasks, which were obtained with a factor graph topology as depicted in figure 3.20. Control precision was about as low as expected given the low-resolution discretization of the observed variables. In about $10 - 20\%$ of all tests, the arm's movement stopped long before the target position was nearly reached. We attribute these cases to the filter algorithm, which stopped all motions which would have caused collisions.

While the hidden variables H_1 and H_2 needed to have a minimum resolution of about $10 - 20$ states, a further increase of the resolution did not yield significant improvements. We also tested higher setups with higher-resolution observed variables, but this had also no beneficial effect on control performance, probably because the number of available data points was not sufficient for more precise training.

3.8 Discussion

3.8.1 Appeal of our approach

In this chapter, we described several factor-graph based approaches to various motor control problems. Besides their distributed computational properties inherited from the framework of factor graphs, the presented algorithms have several appealing properties: First, they are self-contained systems that learn an internal model of a robotic system solely through experience. Second, the undirectedness of the models allows an application of these in any direction. A factor graph that is trained to represent the relation between

cartesian coordinates and angles in a robotic system can be used to compute both the direct kinematics as well as the inverse kinematics. Third, all these computations are a result of simple operations. Both during learning and during live control, the same operations are executed, which are based on the simple sum-product rule. Finally, the control algorithms do not depend on the meaning of the input variables. Instead of transmitting the cartesian coordinate of a robot as input, one could equally well transmit other properties of a current configuration, such as the coordinates of the robot's gripper in two 2D images.

3.8.2 Variable encoding

Low precision of the control algorithms we designed can partly be explained with the chosen parameterization of the functions in the factor graphs: These depend on discrete variables and use a naive scheme to store the function values. Under this kind of parameterization, factor-graphs based models for robotic systems either have a low precision due to low-resolution discretizations, or tend to overfit, because data is too sparse for training the large amount of parameters.

Our initial goal was to overcome the limitations of this parameterization through the use of gradient shaped population codes. However, we were able to show in this thesis that such codes cannot yield the desired effects in factor graphs, unless these graphs are transformed into directed graphs, which can then be used to infer one of the variables. However, we are not aware of any learning algorithm for training graphs.

Since gradient shaped inputs cannot be used to properly smoothen a functions represented by a factor graph, different parameterizations would be desirable. Another motivation is the slow learning convergence, which could be largely improved by using lower-dimensional parameterizations. For most real-world phenomena, it seems like continuous variables would be a wiser choice over variables with discrete states. However, the only continuous functions we know of that allow precise inference calculations are unimodal Gaussian functions, as used for example by Toussaint et al. [Tou09, TG10] or in the Kalman filter [Kal60]. A factor graph with functions representing unimodal Gaussian functions, however, is equal to a single unimodal and high-dimensional Gaussian.

3.8.3 Modelling dynamics of a robotic system with a factor graph

Besides low precision, another apparent problem of the control algorithms we presented is their inability to model the dynamics of a robot. The algorithm for control of a robotic arm ignores the dynamics of the robot but instead drives the motors of the robot linearly

towards a goal. Modelling instantaneous dynamics of a robot in a factor graph proved to be usable in a simple compliant system. In the regarded system, though, goals can in most cases be reached within a single timestep. As soon as a simple obstacle is introduced at a specific angle, a model for the instantaneous dynamics is no longer sufficient because many possible transitions of the system will require several timesteps. By increasing the time constant Δt, such transitions could be modelled, but the model will become less precise for short motions.

Modelling temporal processes apparently requires more sophisticated methods. In chapter 4, we will introduce a new kind of graphical model topologies that are designed to represent long action sequences in terms of a hierarchical organization.

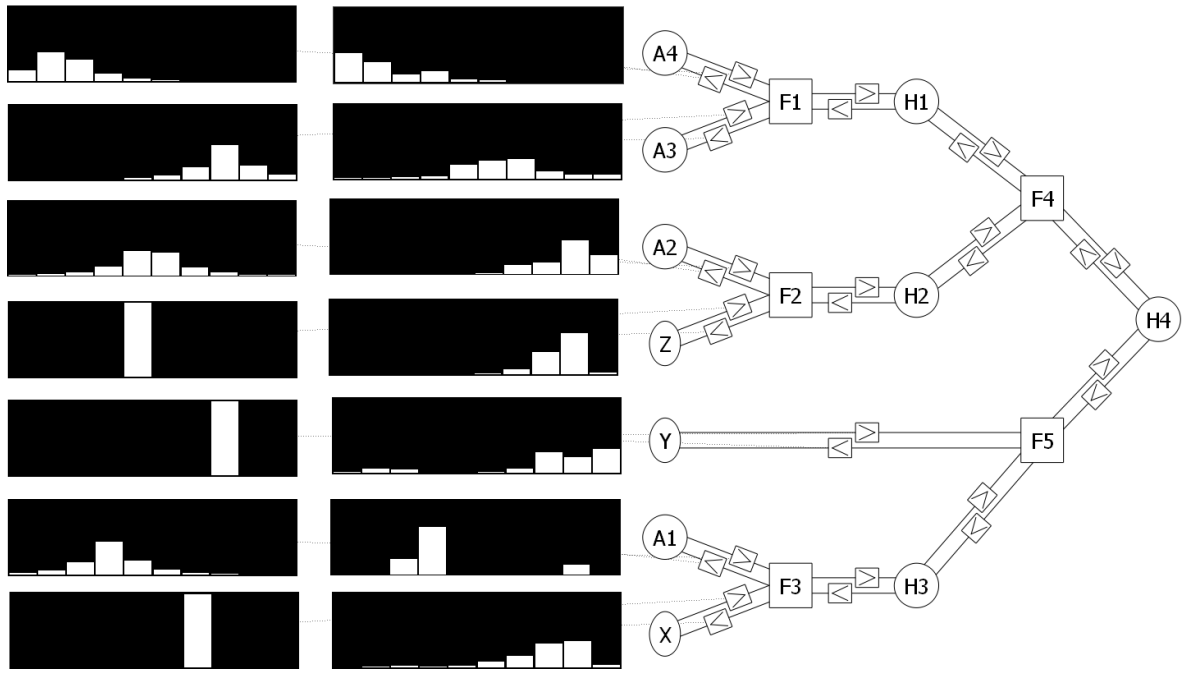

Figure 3.21: *Typical input and output messages in a factor graph used for controlling a robotic arm. The figure shows a screenshot from FGControl. The factor graph on the right is composed of 5 factor nodes ($F1$ to $F5$), 4 hidden variables ($H1$ to $H3$) and the 7 observed variables representing the angles of the robotic arm ($A1$ to $A4$) and the cartesian coordinate (X, Y and Z). Shown on the left are the inputs to the individual observed variables (left) and the output messages arriving at these nodes (right). Each message is composed of 10 elements and the activation of the individual elements is indicated by white bars (full height corresponding to an activation level of one). The inputs to all angle nodes are encoded as population codes with exponential shape, whereas a specific cartesian coordinate is defined by applying sharp messages to the node X, Y and Z. The output messages indicate the factor graphs belief about each variable, given the inputs to all the other nodes. The output messages at the angle nodes are used to compute a set of target angles, towards which a linear motion will be executed. Here, particularly the output messages at $A2$ and $A3$ differ from the input messages, indicating that these angles need to be changed the most.*

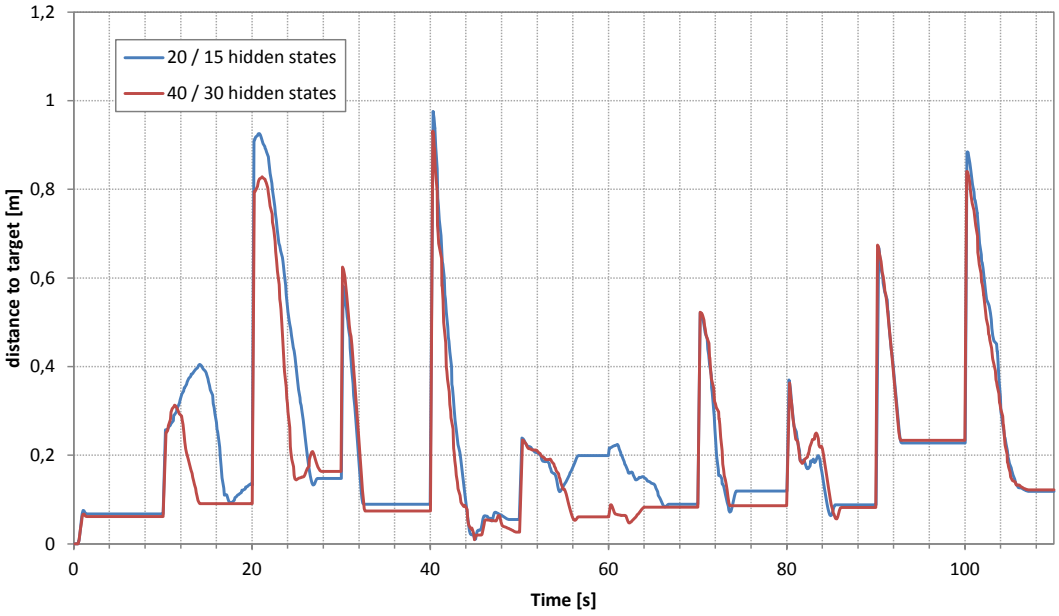

Figure 3.22: *Movements of a robotic arm under factor graph control. A factor graph as depicted in figure 3.20 was trained to represent the kinematics of a katana robotic arm and was then used in live control. Every* 10*s, a new random target position was chosen and the algorithm outlined in section 3.7.3 was used to steer the robot arm based on target angles as obtained from inference calculations on the factor graph. Shown is the temporal evolution of the distance to the target (length of the arm* $\approx 1\,m$*). Two setups with different numbers of states were tested (blue line:* H_1*: 20 states,* H_2*: 15 states; red line:* H_1*: 40 states,* H_2*: 30 states). In both cases, all observed variables were discretized into 10 states.*

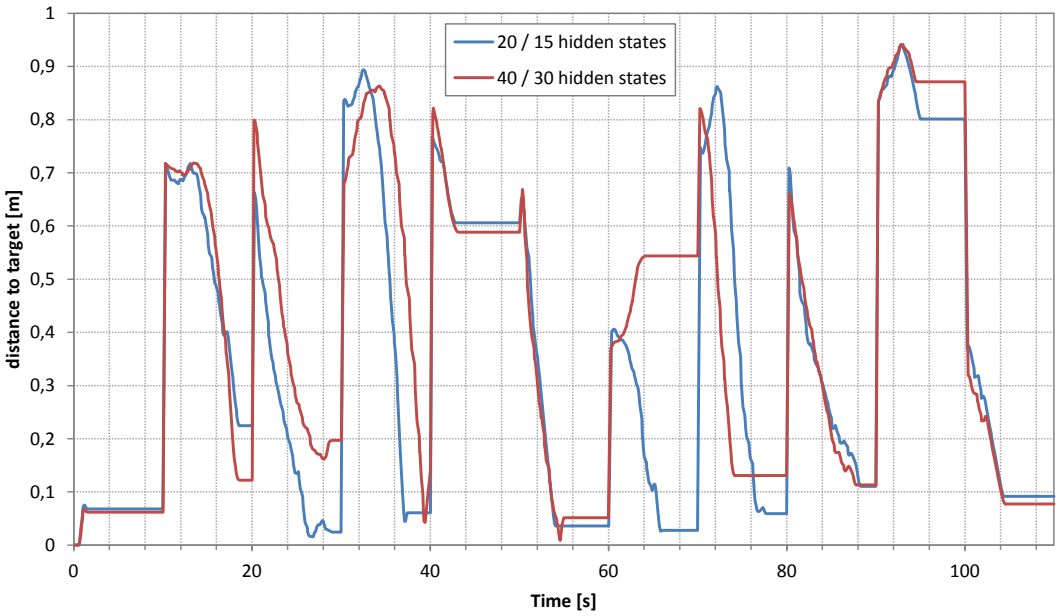

Figure 3.23: *Similar recording as in figure 3.22, but for a different sequence of targets.*

Chapter 4

Hierarchical navigation networks

4.1 Motivation

In order to solve complex tasks, we often need to exert long sequences of coordinated short actions. As an example, imagine a person that is leaving from home and heading to a local airport in order to catch a plane. Solving this task requires the person to execute thousands of short muscle actions. While the complete sequence of actions is necessary in order to reach the defined goal, it is impractical to plan the complete trajectory of all body limbs in advance.

Interestingly, we are able to outline a path to the goal within a few seconds, which allows us to almost immediately initiate a movement. Instead of planning a detailed trajectory for each of our joints and predicting the precise position of our body at all timesteps in the future, we rather follow more abstract goals and subgoals. In the example, the abstract goal the person is pursueing could be the state of being at the airport. Even if the final goal would be defined in terms of a precise position and body pose, these details are irrelevant for reaching it as long as the distance to the goal is high.

An abstract formulation of the goal will in most cases not on its own be sufficient to determine suitable muscle actions, but we can easily formulate less abstract subgoals that need to be reached in order to get to the final goal. These, in turn, can guide the formulation of even less abstract subgoals and so on. In the example, a possible series of such subgoals might be 'reach the bus stop', 'leave the house', 'leave the room', 'stand up' and 'shift weight forwards'. This subgoal is the least abstract and the only one that will have an effect on the precise muscle actions the person needs to execute immediately. Future muscle actions can be computed as soon as the state of the body has changed. In this way, the complete sequence of actions unfolds only during the process of solving the task.

In this chapter, we present a hierarchical approach to navigation that is inspired by

this way of computing a path between a source and a destination. Similar to the scheme desribed above, our approach uses an abstraction hierarchy, which arises as a result from learning through experience. The relation between the current state of the agent and a given goal is evaluated at different abstraction levels. As a result of this evaluation, subgoals at lower abstraction levels are chosen, which ultimately guide the immediate behaviour of the agent.

We show how to implement the described scheme of computation in terms of a *hierarchical navigation network* (HNN), which is related to graphical models and as such is composed of a network of many small computational nodes. Each of these nodes belongs to an abstraction level and is connected to a small number of other nodes at the same or an adjacent abstraction level. Nodes can communicate through these connections by message passing, where each message consists of vectors of real numbers. The messages sent to a node compose the only information that is accessible to that node. While we assume that the network topology is predefined by the programmer, each node can learn its own message-producing behaviour from its experience of the messages it receives. The nodes are organized in a hierarchical structure, which reflect the layering of abstraction levels mentioned above. We describe the topology of HNNs and show how these can be used to solve navigation tasks by applying them to maze navigation tasks. The nodes of the network use learning rules that rely solely on local messages available to the nodes.

Our goal was the development of a biologically-inspired hierarchical system for navigation in arbitrary environments that would be able to learn to represent the environment, learn the causal effects of actions inside it, and be able to guide action selection for navigation. We chose message passing on a graphical model as a computational paradigm because it shares several features with neural computational systems: Computation is distributed among multiple computational nodes, information is stored locally at the computational units, and every unit only has access to this local information and to messages it receives from connected units. We wanted the model to quickly converge when given a source and a destination, efficiently guiding the selection of basic actions.

HNNs have several interesting properties: First, they demonstrate how a simple on-line learning rule can lead to the emergence of a hierarchical decomposition of space. Second, the self-emerging hierarchical decomposition of space allows the network to quickly converge when given a source and a destination and to select an appropriate basic action at every point in the journey. While the generated paths are not necessarily optimal in terms of length, the resulting path lengths get close to optimal as the system learns. Third and most importantly, HNNs represent an alternative way to model temporal processes with graphical models. Processing across different time scales arises naturally as a consequence of the hierarchical spatial decomposition.

4.2 Relationship to existing approaches

Existing solutions to navigational tasks can be classified in terms of several criteria: First, information about the environment can be acquired and stored by the agent in different ways. Second, certain assumptions can be made about the available computational architecture. Third, each approach is designed to work in navigational environments with specific properties. In the following, we will mainly focus on approaches that resemble our approach in the third criterion by being designed to work in graph-based environments. When working with other types of environments, such as continuous geometric spaces, other approaches can be applied including solutions from control theory, which require that the behaviour of the navigational agent can be expressed as a dynamical system. This, however, cannot easily be done in graph-based environments.

Regarding possible schemes of information acquisition and storage, the easiest situation is found when the full structure of the world is known and a navigational agent is allowed to directly access the connectivity of a graph, for example in the form of adjacency lists. This assumption is made in standard graph navigation algorithms that go back to Dijkstra's algorithm [Dij59]. More advanced extensions like the D* algorithm [Ste95] also work when parts of the graph are invisible at the time of computation, but these approaches all have in common that all acquired information can be directly accessed and that a sequence of actions is computed by iteratively accessing this data. In contrast, information in our approach is stored locally at the computational nodes. Also, each of the nodes computes its output messages in a non-procedural way as a result of the incoming messages, without having to store any intermediate results.

Incomplete information about the environment is also assumed in solutions from reinforcement learning. In these approaches, an agent learns behaviour through trial-and-error interactions with an environment [SB98] by evaluating a reward signal it receives. When applied to navigation problems, a RL-based agent would typically receive a positive reward signal whenever reaching a specified goal, and it would be trained to estimate expected future reward values for all possible pairs of state and action. Actions would then be selected based on their expected future reward. The information about expected future reward distinguishes RL approaches from standard graph navigation algorithms. It resembles a direction pointer which guides the behaviour of an agent toward a goal and thereby allows action selection without the need to plan longer sequences of actions. However, if the agent is supposed to be able to navigate to any destination, it needs to be trained for each possible destination and for each triple of source, action and destination, a separate value of expected reward needs to be learned and stored. This results in an increase of learning time and storage requirements when the corresponding data is stored in a naive way. In comparison, our approach does not rely on a reward signal. Instead

of modelling a three-dimensional function $f_{RL} : \{goal, state, action\} \rightarrow reward$, a HNN rather represents the two-dimensional function $f_{HNN} : \{goal, state\} \rightarrow action$.

When there is a regionalized structure to the environment, then there are significant redundancies within this stored information. Hierarchies can be used to express the regional structure of an environment and thereby reduce the amount of information stored. This has been exploited by hierarchical approaches to RL, such as option-related approaches [BM03]. Here, the hierarchy is on paths of different lengths, called options. Options refer to policies that allow transitions between defined distant points in space [SPS99]. A skill that allows a transition between a given pair of source and destination can be reused in other contexts as part of a longer path, or as part of another skill, which yields a hierarchical organization of skills. In approaches using intrinsic motivation, skills can automatically be learned by an agent [SBC05].

Another way of exploiting regional structure in the environment by using hierarchies was demonstrated by Dayan and Hinton, who looked at transitions between regions in a hierarchy of different spatial scales [DH93]. In terms of information acquisition and storage, their approach is closely related to the HNNs we present: In both cases, information about the environment is acquired in the form of sequential exploration data and both approaches use a hierarchy of spatial scales to represent the acquired knowledge. The existence of a reward signal as well as the inability to learn higher level representations from experience distinguish this approach from ours.

The existing approaches to navigation described above resemble each other in terms of the typical computational architecture that is required for algorithm execution. Approaches from graph theory and RL are traditionally designed to work on classical von-Neumann architectures and as such are based on procedural algorithms that have global access to all stored information. However, there exist more recent approaches that are based on distributed schemes of computation. Conradt employed an approach to graph navigation that is remarkably similar to Dijkstra's algorithm but relies on a network of agents that communicate with each other [Con08]. Here, the nodes of a graph are represented by individual agents and a path search is performed through message passing between the agents.

Similarly, solutions have emerged from reinforcement learning that rely on distributed computational schemes. These are often biologically motivated and work with neural networks. Examples can be found both in value function approaches [FMD00, SCSG05, LWW10] and in approaches using a direct policy search [Seu03, LWW10].

The HNNs we introduce here are each composed of a single graphical model and as such inherits the distributed computational character of these models: Computation is shared amongst several small computational units that share information through message

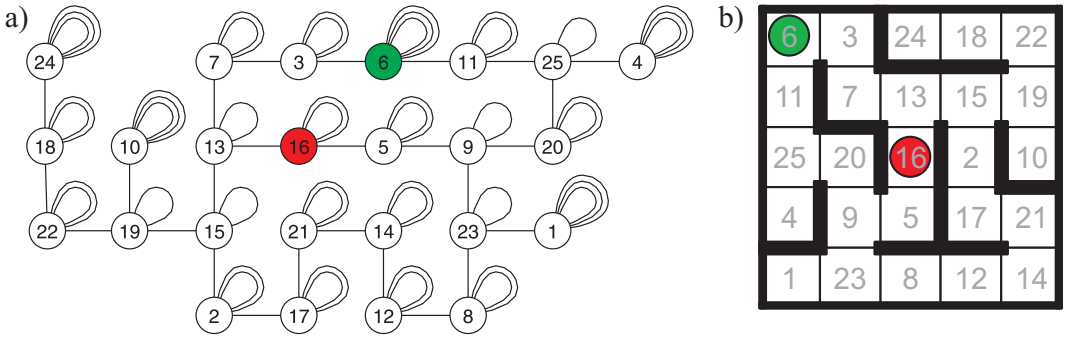

Figure 4.1: *a) Graph navigation: The objective is to move the agent (green node) along edges in the graph to its destination (red circle). Numbers indicate an order in which the states could be sorted and should emphasize the absence of a prior knowledge regarding the graph structure. b) Mazes with discrete state space can be formulated as graph navigation tasks. The maze shown here is equivalent to the graph shown in a).*

passing.

4.3 Factor graphs for navigation

4.3.1 The task

In the following, we are considering a simple navigation task: An agent is placed at a node in a graph containing N nodes and is given the label of a destination node inside the same graph (see figure 4.1). At each timestep, the agent chooses one edge along which it will travel to an adjacent node. Thus, edges represent single-timestep-actions that the agent can execute. Self-edges represent actions that cause no change to the agent's position. For simplicity, we will only consider graphs with small constant node degree, typically 4. This allows us to use the same labels for the possible actions at each node. Given the labels of the current position and the destination, the task is to execute a sequence of actions that moves the agent to its destination.

The setup described above can be described as a Markov decision process (MDP) with a deterministic relation between actions and state transitions. The problem can be extended by making the outcome of actions probabilistic. For simplicity, we will assume that the environment behaves deterministic for now, but we will also test our approach in probabilistic environments in section 4.6. Maze navigationally tasks can be formulated as graph navigation problems as depicted in figure 4.1.

4.3.2 Standard solutions to graph navigation

The standard approach to solve a task as described above is to store the adjacencies between positions in the maze and then to perform a breadth-first search starting at the current position. This approach yields the optimal path. However, the breadth-first search algorithm has two main disadvantages: First, its time complexity is $O(d^k)$, where d denotes the distance between the two positions and k the number of dimensions of the maze. While this tolerable in low dimensions, it becomes infeasibly slow for navigation in high-dimensional mazes. Second, this type of iterative algorithm is not well suited to implementation in neural architectures: While determining the optimal path, the algorithm needs to cache sequences of positions and actions or at least the preceding action for each position that was reached. While multiple positions could be remembered by having dedicated neurons or groups of neurons to each store one single position, caching of actions would require multiple locations to be able to store the same actions.

Another approach would be to pre-compute all paths between each possible combination of current position and destination and to store the first action for each such combination.

4.3.3 A factor graph approach

The adjacencies between different positions in the maze could also be stored by a small factor graph as depicted in figure 4.2a: Here, an action A and the position before (P) and after that action (D) are represented by variable nodes. The factor node connecting all three variable nodes contains a three-dimensional table containing the probabilities for each event: $P(P, A, D)$. In a deterministic environment as in our example, this table would only contain $N \cdot |A|$ non-zero entries, where $|A|$ denotes the number of possible actions.

This factor graph could be used to infer the outcome of a single action or to infer the action that gives the maximum likelihood to move the agent to a desired position.

However, it would be more difficult to infer a correct action if the goal state cannot be reached with a single action, because $P(P, A, D|D \notin V_P) = 0$, where V_P specifies the set of states adjacent to P. One possibility to deal with the temporal aspect of sequences of actions in this setup is to 'unroll' time as depicted in 4.2b. In this graph, each action at each future timestep is represented as a distinct variable node. However, since the path to the destination is unknown, the intermediate states would need to be inferred together with the appropriate actions. Hence, finding the optimal sequence of actions would require a more elaborate scheme than simply sending the beliefs along the graph. Additionally, this approach requires the knowledge of the length of the path, which is

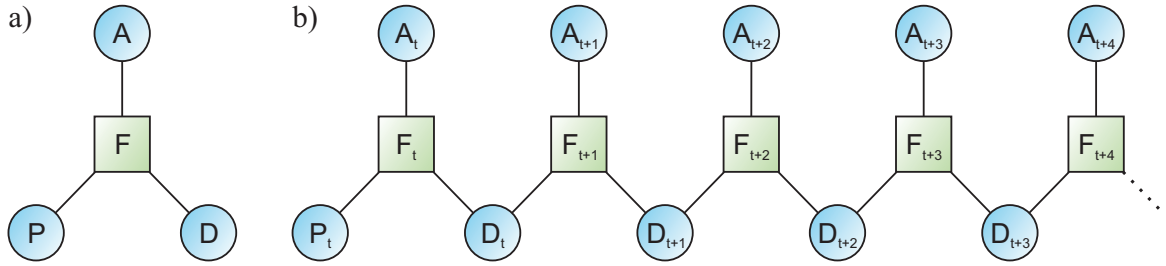

Figure 4.2: *a) A factor graph to infer single actions. P denotes a position before executing an action A and D the position after that action. The factor F defines the transition probabilities between states. b) Standard approach for inferring sequences of actions using factor graphs by unrolling time. Starting at a position P_t, the execution of a sequence of actions $(A_t, A_{t+1}, \dots)$ causes the sequential transition to positions $(D_t, D_{t+1}, \dots)$. The transition probabilities at time step t are defined by F_t.*

usually not accessible. Determining this parameter further complicates the algorithm.

The repetitive structure of this kind of network also renders it neurally unplausible: Neurophysiological evidence makes it seem unlikely that our brains store many copies of the same information, i. e. the adjacencies between different positions in space. Of course, it would be possible to perform the same computations by using a network as in figure 4.2a and repetitively use a data structure storing the parameters of one single factor node. This iterative scheme, however, would require the storage of many intermediate results, which again seems difficult with a neurally plausible setup.

Finally, both approaches suffer from another problem when it comes to computing long action sequences: All actions have to be pre-computed before a single action can be executed. Hence, the computational time required for planning cannot be less than linear in the distance between the current position and a goal. This would limit the length of sequences that could be planned by a neural information processing system within a few milliseconds.

In dynamic environments, computing the complete sequence of actions before executing a single action can also be futile: Some data about the environment might not be available at the time the path is computed or the environment could change and thereby render the pre-computed path invalid.

4.4 Hierarchical navigation networks

4.4.1 Hierarchy of simple factor graphs

Unlike the factor graph-based approaches described in the last section, we want to pursue another approach to deal with the temporal aspect of sequence planning. As we have seen

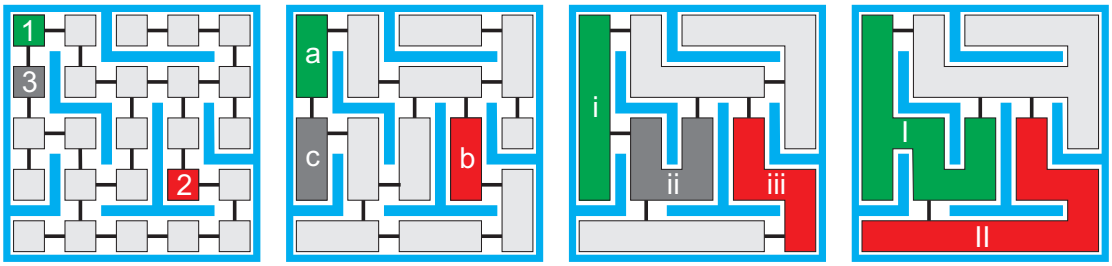

Figure 4.3: *A maze at different resolutions. Possible transitions in the original maze can be represented by a graph as indicated in the left frame by the black connections. Lower resolution transition graphs can be obtained by clustering positions. At some resolution, the current position (green) and a target (red) will be adjacent to each other, which allows to determine a high-level action, such as 'move from region I to region II'. The knowledge of such an abstract action allows to determine subgoals (dark grey) for representations with higher resolution. For example, the transition from region I to region II in the rightmost representation can be initiated by first executing a transition from region i to region ii.*

above, it is easy to infer one-step-sequences: Using belief propagation on the factor graph in 4.2a with a given current position and a destination yields the marginal probabilities for the actions. Selecting the action with highest probability then gives the best chance to find the goal.

Assuming there existed a representation of the same maze with a coarser resolution along with a set of higher-level actions, we could create a second factor graph, storing the same relation as before, but for higher level states $P^{(1)}$, $D^{(1)}$ and higher-level actions $A^{(1)}$. After mapping the current position P and the destination D to the corresponding higher-level states $P^{(1)}$, $D^{(1)}$, we could infer a higher level action, given that the current position and the destination are adjacent in the coarser representation.

Continuing along these lines, we could use coarser and coarser representations together with more and more abstract actions. This way, a hierarchy of representations would be obtained. At some resolution of the maze, the current position and the destination will be adjacent to each other, which would then allow us to infer some high-level action.

Coarser representations of the maze can be generated by clustering adjacent positions as shown in figure 4.3. This can also be done by analyzing recorded position sequences of movements in the maze as we will do in section 4.5.1. However, the execution of high-level actions seems to require the solution of the original planning task, because it should be possible to generate a sequence of elementary actions from one high level action.

4.4.2 Combined hierarchical model

In our approach, we define actions at one level of the hierarchy in terms of positions at the next lower hierarchy level. This way, an action at hierarchy level $h+1$ would correspond to the instruction to go to a specific position which is defined in the space of hierarchy level h. In the following, we will refer to this position as *subgoal* $S^{(h)}$ in order to distinguish it from the final destination of the movement. Instead of targeting the destination directly, the agent performs movements to achieve subgoals.

Figure 4.4 illustrates how different hierarchy levels can be combined to one message-passing model, which we call an hierarchical navigation network (HNN): Similar to figure 4.2a, the current position $P^{(0)}$, the final destination $D^{(0)}$ and the elementary action A are represented as variable nodes. Coarser representations $P^{(1)}$ and $P^{(2)}$ of the current position are extracted on the left side of the network. Similar coarse representations of the destination are extracted on the right side. At each level h, a variable node indicates the current subgoal $S^{(h)}$. This subgoal can be induced by the destination $D^{(h)}$ or computed as a function of the current position $P^{(h)}$ and the subgoal $S^{(h+1)}$ which is selected at the next hierarchy level. This function is stored in the factor node $F^{(h)}$. Since this node supervises the selection of a suitable subgoal at the level, we will refer to it as *instructor node* below.

Similar to messages in factor graphs with discretet-state variables, messages in an HNN are vectors of real numbers. The length of the messages that are communicated with variable nodes is defined by the number of states of the corresponding variable and each element in the message vector is linked to the probability that the variable is in a certain state.

4.4.3 Navigating using an HNN

Figure 4.5 illustrates how a single elementary action is determined using an HNN. Input to an HNN is passed to the variable nodes $P^{(0)}$ and $D^{(0)}$ in form of messages encoding the observed current position and the destination of the agent. In a deterministic environment, a single entry in these messages would be set to one, in order to encode that the observation is certain.

The input messages to the nodes $P^{(0)}$ and $D^{(0)}$ are used to compute messages to nodes $P^{(1)}$ and $D^{(1)}$, which encode the current position of the agent and the destination at coarser representations of the maze. Messages to higher level nodes $P^{(h)}$ and $D^{(h)}$ are computed analogously. At all levels, the destination is also set as a possible subgoal by forwarding the incoming message to the destination-node $D^{(h)}$ to the neighboring subgoal-node $S^{(h)}$. At some level h, the current position and the subgoal will be adjacent to each other. The

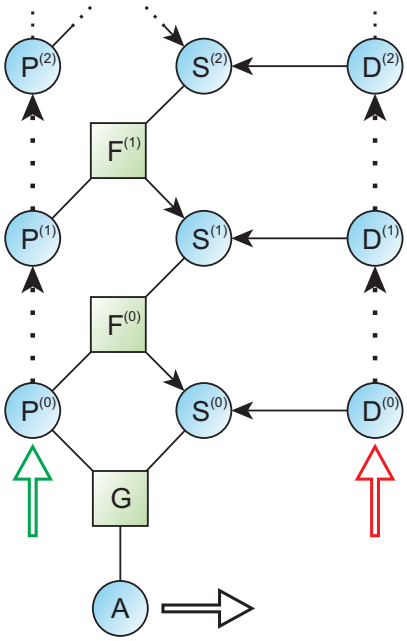

Figure 4.4: *The graph structure of an HNN. The hierarchical graph can be interpreted as being a composition of several small graphs as depicted in figure 4.2a). Each such small graph operates at a different spatial scale of the environment and can only determine actions if the current position and the destination are adjacent to each other. Actions at higher levels are defined in terms of positions at the next lower resolution, which allows to glue the simple graphs together. Subgoal nodes $S^{(h)}$ are introduced in order to combine the destination at each level with subgoals induced from higher levels. The current state (green arrow) and a destination (red arrow) are transmitted to the nodes $P^{(0)}$ and $D^{(0)}$, respectively. From there, coarser representations are extracted, which are sent upwards in the hierarchy to the nodes $P^{(1)}$, $P^{(2)}$ etc. and similarly for the destination. Subgoals $S^{(h)}$ at each level h are either set to the destination at the same level or are induced from a higher level subgoal through an instructor node $F^{(h)}$. By comparing the subgoal $S^{(0)}$ at the lowest level with the current state, the node G selects an action A (black arrow). For details, see figure 4.5.*

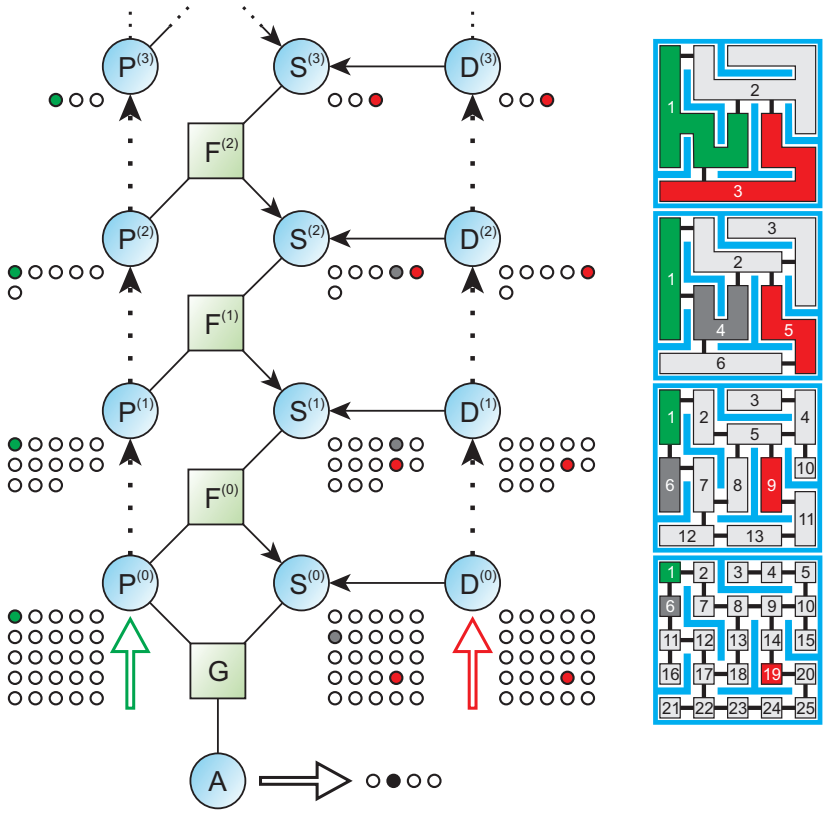

Figure 4.5: *Determining an immediate action using an HNN. In the example, a 5x5 maze is represented by an HNN. An observed current position and a destination are encoded as 25-dimensional vectors and transmitted to the nodes $P^{(0)}$ and $D^{(0)}$. In the example, these messages are represented by matrices of 5x5 circles. Only one entry corresponds to the observed position and is set to one, as indicated by a filled circle. Information about the current position and the destination is passed upwards through the network. At each level, the resolution at which the maze is represented is reduced and along with that, the length of the transmitted vector is reduced. Each node encoding the destination forwards the incoming message (red circle) to a subgoal-node at the same level, where it is combined with a message from the next higher level of the hierarchy (grey circle). The latter message is computed by the instructor node $F^{(h)}$ by comparing the subgoals at the next higher level with the current position at its own hierarchy level. The instructor nodes have limited knowledge and can only determine subgoals for the lower level if the subgoal at the higher level and the current position at that level are adjacent. Since the resolution of the maze is reduced with each level, it is ensured that at some level, the destination (which is also a subgoal) and the current position are adjacent to each other, which allows to send a suitable subgoal down the hierarchy. This procedure ensures that at each hierarchy level, at least one subgoal is known which can be used to compute a subgoal for the next lower level. Finally, a comparison of the possible subgoals and the current position at the lowest hierarchy level yields an output message encoding a possible action, which is again encoded as a vector, where each possible action is represented by one entry (here: 4 possible actions, 'move downwards' is selected).*

instructor node $F^{(h-1)}$ will then be used to determine an alternative subgoal $S^{(h-1)}$ for the next lower level, which will be adjacent to the current position $P^{(h-1)}$ at that level. This process continues until a suitable subgoal has been found for the lowest level, which will allow the derivation of an elementary action. The output is read at the action-node A and equals the message from G to A: This messages assigns each possible action one value, with each value rating whether the corresponding action should be selected in order to move the agent closer to the destination. The output message can then be used by a controller to select a single action.

When executing an elementary action, the current position changes. This will in turn cause an update of some messages sent from the $P^{(h)}$-nodes, finally resulting in a newly computed elementary action. Due to the coarser representation at higher levels, subgoals at these will change more slowly than low-level subgoals. Note that the precise sequence of elementary actions is computed while the agent is moving, it is not pre-computed. This bears the advantage of being able to react to unpredictable events along the way. This is especially useful if the movement of the agent or the behaviour of the world are stochastic.

4.4.4 Smooth higher-level regions

Up to now, we have assumed that each high-level position corresponds to a precisely defined group of lower-level positions. In this approach, a high-level state can only be in one of two states at a time: activated or not activated. If we allow states to be partly activated, it becomes possible to create high-level states that correspond to regions of lower-level states with soft boundaries. Given a precise position at maximum resolution, several higher level states can be partly activated at the same time, together composing a population code for the current position. Figure 4.6 shows an example of how the 12 second-level states can be tuned to regions with soft boundaries.

Theoretically, the described population code would allow the identification of every elementary position based on the activation levels of all states at one level of the hierarchy. In practice, this will be limited by the noisy resolution of partial activation. However, it does allow a more precise definition of the target and the current position at a given level of the hierarchy. Navigation can benefit from this increased precision: Assuming a goal lies within a certain region, but there are several possible ways to get from the current position to the goal, partial activation of several regions that are close to the goal can help to choose the shortest path.

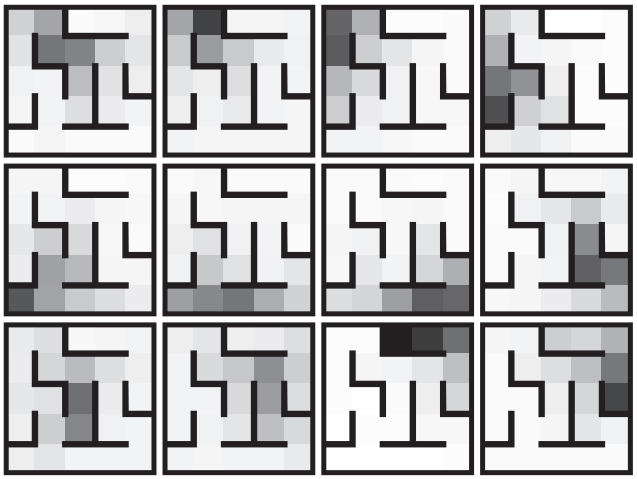

Figure 4.6: *Assignment of regions with soft boundaries to 12 states in level 2. Each depicted maze shows the tuning of one of 12 states. Grey levels indicate strength of activation.*

4.4.5 Message passing between nodes

In our model, we assume a message-passing communication style between nodes, which is very similar to belief propagation in factor graphs [KFL01]. Unlike in a factor graph, though, information flow in a HNN is directed.

A more detailed cutout of the graph delineated in figure 4.4 is shown in figure 4.7, which illustrates the messages that are sent between nodes at hierarchy level h. Green square nodes in the graph behave like factor nodes in a factor graph and use the sum-product rule to compute outgoing messages. Blue circle nodes are related to variable nodes in factor graphs, with the exception that the nodes $P^{(h)}$ and $D^{(h)}$ forward the unmodified incoming message to two nodes.

The message $p^{(h)}$ arriving at the node $P^{(h)}$ defines the current state at the resolution of level h. It is forwarded to the nodes $V^{(h)}$ and $F^{(h)}$. At $V^{(h)}$, this message is translated into a coarser representation as follows:

$$p_i^{(h+1)} = \sum_j v_{ij}^{(h)} \cdot p_j^{(h)} \tag{4.1}$$

Here, the parameters $v_{ij}^{(h)}$ define the regions to which the states at level $h+1$ are tuned. Basically, the activation $P^{(h+1)}$ of states at level $h+1$ is computed by multiplying the activation vector $P^{(h)}$ with a matrix $V^{(h)}$: $P^{(h+1)} = V^{(h)} \cdot P^{(h)}$. The activation of the individual states of $D^{(h+1)}$ are computed analogously, by multiplying the activation at

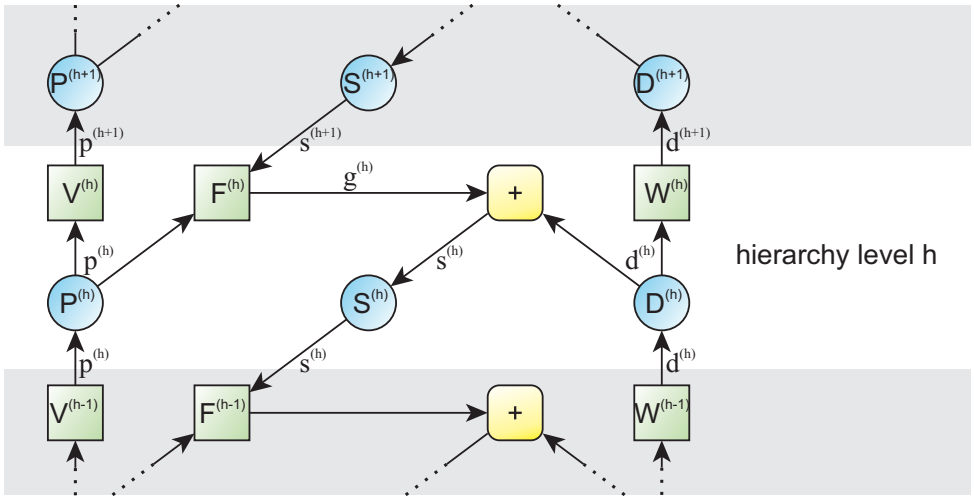

Figure 4.7: *Details of message passing in an HNN. Shown are the messages (see edge labels) that are sent between nodes on a hierarchy level h. In comparison to figure 4.4, three additional nodes per level are shown: $V^{(h)}$, $W^{(h)}$ as well as a +-node, which performs a weighted addition of incoming messages. Blue circles represent variable nodes, which simply forward messages arriving at them. Green squares represent factor nodes which compute an outgoing message by applying the sum-product rule to the incoming messages.*

level h with a weight matrix $W^{(h)}$:

$$d_i^{(h+1)} = \sum_j w_{ij}^{(h)} \cdot d_j^{(h)} \tag{4.2}$$

Again, $w_{ij}^{(h+1)}$ are the parameters defining the shape of higher-level states.

In the context of the maze task we are considering here, current state and destination have the same input encodings, which are fed into the nodes $P^{(0)}$ and $D^{(0)}$. It would therefore be possible to set $V^{(h)} = W^{(h)}$ for all hierarchy levels h, such that the representations of current state and destination would be equal accross all levels. However, we use distinct matrices for each side, which are trained separately from each other. This way, the representations at the individual levels will typically differ from each other, thereby confirming that our approach would also work smoothly in setups where the inputs for current state and destination have different input encodings.

The instructor node $F^{(h)}$ receives the message $p^{(h)}$, which defines the current state at the respective hierarchy level, and the message $s^{(h+1)}$, containing information about a desired higher-level subgoal. It combines these two messages using the sum-product rule in order to infer subgoals at its own hierarchy level, which are compatible with the

higher-level subgoal, encoded in the message $g^{(h)}$:

$$g_k^{(h)} = \sum_{i,j} f_{ijk}^{(h)} \cdot p_i^{(h)} \cdot s_j^{(h+1)} \tag{4.3}$$

$$\tag{4.4}$$

The destination, encoded in the message $d^{(h)}$ determines one possible subgoal that could be desirable to reach. Whenever this goal is out of reach, a subgoal should be selected which is compatible with higher-level goals, as contained in the message $g^{(h)}$. These two messages are combined by weighted addition at the +-node into a message $s^{(h)}$, containing all possible subgoals:

$$s_j^{(h)} = \alpha \cdot \overline{d}_j^{(h)} + (1-\alpha) \cdot \overline{g}_j^{(h)} \tag{4.5}$$

Here, α defines the ratio by which the destination and the message coming from the next higher level are weighted. We used $\alpha = 0.9$. Note that the message $s^{(h)}$ is the only message which is not computed according to the sum-product rule. The incoming messages are first normalized in order to ensure proper weighting:

$$\overline{d}_j^{(h)} = d_j^{(h)} / \sum_i d_i^{(h)} \tag{4.6}$$

$$\overline{g}_j^{(h)} = g_j^{(h)} / \sum_i g_i^{(h)} \tag{4.7}$$

By making $\alpha > 0.5$, the current destination is assigned a higher priority than subgoals that are compatible with higher-level subgoals. That is, if in the representation at a certain level the current position and the destination are close enough to each other to know what action to take, then this takes precedence over the less-precise instructions from the higher level.

4.4.6 Scaling properties

In the following, we will analyze the scaling properties of our hierarchical approach with respect to computational cost and required storage. For the purpose of this analysis, we make two simplifications: First, we assume that a state at hierarchy level h responds to exactly two states at hierarchy level $h-1$ whose response regions are connected. This way, the size of response regions doubles with each level. Second, the representations for the current state and the goal should be identical at all levels. Our third restriction concerns the data stored in instructor nodes: Assume the current position causes activity of states $c^{(h)}$ at hierarchy level h and $c^{(h+1)}$ at hierarchy level $h+1$. An instructor node $F^{(h)}$ should

be able to determine a subgoal $s^{(h)}$ for level h if and only if it is given a subgoal $s^{(h+1)}$ at level $h+1$ that is adjacent to $c^{(h+1)}$. The output $s^{(h)}$ of this instructor node needs to be a region inside $s^{(h+1)}$ or $c^{(h+1)}$.

The last requirement makes sure that our algorithm will allow navigation in all cases possible with the described setup: Any destination will be adjacent to any current position at some hierarchy level h. The instructor node at this level will either provide a lower-level subgoal that is closer to or a subregion of the higher-level goal. In the first case, the agent will later be able to provide a subgoal similar to the second case. In the second case, the agent will move to a position that will be adjacent to the destination at a hierarchy level smaller than h. By induction, it follows that the agent is guaranteed to find the destination.

Network size

Since we require that the region sizes double with every hierarchy level, a single region at level h will cover a region of 2^h positions. Hence, in a maze of size 2^h, every pair of positions will be adjacent to each other at some hierarchy level smaller than h. We conclude that the maximum number of hierarchy levels that need to be engaged to compute a path in a maze of size N is $\log_2 N$.

Node sizes

If we assume that regions at hierarchy level h have an average $m^{(h)}$ neighboring regions of the same size, an instructor node $F^{(h)}$ needs to store $m^{(h+1)}$ values for each possible current position $c^{(h)}$. Since the maze is split into $N/2^h$ regions at level h, the instructor node needs to store a total of $m^{(h+1)}N/2^h$ values. The total number of values stored in all instructor nodes is therefore:

$$\sum_{h=0}^{\log_2 N-2} m^{(h+1)}N/2^h \tag{4.8}$$

Additionally, at the lowest level, the action instructor G needs to store $Nm^{(0)}$ values. If $m^{(h)}$ is approximately constant and equal to m across hierarchy levels, the total sums up to less than $3mN$ values. This approximation is reasonable for many real-world applications such as navigation in two-dimensional environments, as long as the shape of regions is similar at all hierarchy levels.

In order to obtain higher level representations of a current position or a destination, we also need to store mappings between the levels. In our simplified case, every region at one hierarchy level projects to exactly one region at the next hierarchy level. Hence, we need

to store $2 * N/2^h$ values, where the additional factor of 2 reflects the fact that we need to extract higher level representations both for current position and destination. Summing up over all levels, this corresponds to $4N$ additional values. Overall, we conclude that our algorithm requires storage of $O(Nm)$ values. Interestingly, this is result is similar to the number of parameters needed to store a state-action-reward function, which is used in classical reinforcement learning [SB98] in order to learn movements to a single fixed goal.

Computational cost

In order to compute one single action given a current position and a destination in a maze of size N, $O(log_2 N)$ messages need to be sent, which follows directly from the network size. With the assumptions given above, every message will contain only one non-zero entry. It is therefore possible to also restrict the computational costs to $O(log_2 N)$.

Path length

In the worst case, the restricted version of our algorithm used in this section might generate paths with length $O(N)$, independent of the minimal distance of starting position and destination. This worst case, however, will only happen in specifically designed malicious setups, where higher level regions are laid out disadvantageously. Examples are situations where the current state and the destination are only adjacent to each other at an upper hierarchy level and the common boundary of the corresponding high-level regions is far away from the current state and the destination. The algorithm will always choose a path through this boundary, which will be inefficient if a shortcut between current position and goal exists. If high-level regions are chosen wisely and form a compact region at maximum resolution, these worst case scenarios can be avoided.

Using smooth boundaries for higher level regions and relaxing the assumptions made above will render this problem irrelevant in real-world use. On average, path lengths resulting by application of our algorithm are close to optimal, as shown in section 4.6

4.5 Implementing an HNN

4.5.1 Training parameters in an HNN

Up to now, we implicitly assumed that we would be given hierarchical networks with appropriate parameters. For the system to be able to learn to navigate in new environments, it must be possible to train these parameters based on experience. For the sake of neural

plausibility, learning should happen locally. That is: the parameters of each node in the network should be trained using only the messages received by the node.

In the maze task described in section 4.3, training data can be acquired by using an explorative controller to generate actions. A simple such controller could execute a random walk. In order to generate more long-distance information about the maze, more sophisticated controlling strategies need to be applied which take advantage of previously aquired knowledge about the maze. A partially trained hierarchical network can also serve to generate more directed walks.

Data acquired using an exploratory controller consists of a list of actions together with a stream of position data that was generated while successively executing the actions. For each action that was executed in a training run, the positions before and after executing this action and the action itself are presented to the nodes $P^{(0)}$, $D^{(0)}$ and A, respectively. After all messages in the network have been updated, the parameters of the nodes are adjusted. This process of presenting one sample and adjusting the parameters is repeated for all actions of a training run in the order they were executed.

In the following, we present learning algorithms that demonstrate that all parameters in an HNN can be learned based on local messages only and with limited storage for buffering. However, they were derived phenomenologically and no assertions can be made neither about convergence time nor about optimality of the learned parameters. It remains an open question whether learning algorithms exist that can be shown to be optimal in terms of a global cost function or in terms of convergence time while preserving the desirable features of our algorithms.

Generating coarse representations

States in higher-level representations of a maze should correspond to groups of lower-level positions which are spatially clustered. In a stream of position data, generated from walks in the maze, spatial clustering will be expressed as temporal clustering. Neighboring positions will also appear close to each other in the stream. Therefore, we could apply any temporal clustering algorithm to the provided data stream in order to generate higher-level representations. In order to make our training algorithm more neurally plausible, we chose to use a clustering method that has only access to a short history of positions instead of being able to consider all data in a long data trace.

We will now describe our clustering algorithm, which is based on a winner-take-all method. As described in section 4.4.5, the activation of states at a level h is computed by matrix-multiplying the activities from level $h-1$. Both the current position and the destination are transformed into the higher-level representation, by multiplying with $V^{(h)}$ or $W^{(h)}$, respectively. Because both matrices are trained using the same method, we will

only describe the training of $V^{(h)}$ in the following.

Initially, all entries $v_{ij}^{(h)}$ of $V^{(h)}$ are initialized to constant values. After each presentation of a new datapoint, a winninng row $i^*(t)$ and a winning column $j^*(t)$ of the matrix $V^{(h)}$ are selected in the following way: First, the activities $p_j^{(h+1)}$ are computed according to equation 4.1. These activities define the outgoing message. Second, a winning row $i^*(t)$ and a winning column $j^*(t)$ are selected based on the activities in the outgoing and the incoming message, respectively:

$$i^*(t) = \arg\max_i \frac{p_i^{(h+1)} - \mu_i^{(h+1)}}{\sigma_i^{(h+1)}} \qquad j^*(t) = \arg\max_j \frac{p_j^{(h)} - \mu_j^{(h)}}{\sigma_j^{(h)}} \tag{4.9}$$

Here, $\mu_i^{(h)}$ and $\sigma_i^{(h)}$ denote the average activity and its standard deviation for state i at level h. These values are continually updated according to the following equations:

$$\mu_j(t) = (1-\alpha) \cdot \mu_j(t-1) + \alpha \cdot p_j^{(h)} \tag{4.10}$$

$$\sigma_j^2(t) = (1-\beta) \cdot \sigma_j^2(t-1) + \beta \cdot (\mu_j(t) - p_j^{(h)})^2 \tag{4.11}$$

The parameters α and β need to be adjusted to the respective hierarchy level, we used values of 0.002 at $V^{(0)}$ and 0.001 at $V^{(1)}$, both for α and β.

The winning row $i^*(t)$ corresponds to an element of the outgoing message and therefore to a state of $P^{(h+1)}$ at the next hierarchy level. Similarly, the winning column $j^*(t)$ corresponds to a state of $P^{(h)}$. During training, our aim is to increase the sensitivity of the winning higher level state for the current winning lower level state, but also for winning lower-level states at recent timesteps. In this way, lower level states that were observed in successive timesteps in the training data are mapped to the same higher level state. Thereby, a temporal clustering of data points is interpreted as a spatial clustering.

We use a row vector x to assign to each state of $P^{(h)}$ a value to memorize its recent winning activity. At each timestep, the vector is multiplied by a γ and the element $j^*(t)$ is increased by ε:

$$x_j(t) = \begin{cases} \gamma \cdot x_j(t-1) + \varepsilon, & \text{if } j = j^*(t) \\ \gamma \cdot x_j(t-1) & \text{else} \end{cases} \tag{4.12}$$

This vector is then added to the winning row $j^*(t)$ in the matrix V. ε denotes a learning constant and γ a decay-constant, determining the weighting for winning states at past timesteps, with values between 0 and 1. Since the effect on a winning row $j^*(t)$ persists over time, several high-level states might become tuned to the corresponding lower-level state, which introduces a competition between high-level states for a given low-level state.

With larger γ-values, the decay becomes slower and therefore, more future winning states of $P^{(h+1)}$ will be tuned to state $j^*(t)$ of $P^{(h)}$. This has two effects: First, the response regions of the higher level states will have softer boundaries and there will be more overlap between the regions. Second, competition between high level-states is increased. This second effect results in more compact high-level regions because such regions are more stable under the effects of competition. This helps to avoid the creation of high-level response regions that are disconnected.

After incrementing the winning entries, the matrix $V^{(h)}$ is normalized to one, which will cause a decay of all entries that are never increased.

Training inter-level instructors

As shown in equation 4.3, the message $\vec{g}^{(h)}$ is computed using parameters $f_{ijk}^{(h)}$. These need to be trained based on messages that arrive at the $F^{(h)}$-nodes. In section 4.4.5, we described how these nodes receive messages from the nodes $P^{(h)}$ and $S^{(h+1)}$. The output message $g^{(h)}$ of the $F^{(h)}$-node should have the same encoding as the output message of the $D^{(h)}$-node to allow a simple weighted addition of both messages at the +-node. Therefore, the $F^{(h)}$-node needs to also have access to the output message of $D^{(h)}$ in order to learn the parameters $f_{ijk}^{(h)}$.

During training, changes of activities in the messages coming from $S^{(h+1)}$ can be observed. An increase in activity of one state between timesteps t and $t + \tau^{(h+1)}$ is apparently caused by the sequence of actions that was executed within the past $\tau^{(h+1)}$ timesteps. Our goal is to reproduce this sequence whenever the destination causes a high activity in the same state. Similarly, sequences that lead to a decrease of activity should be avoided in such situations.

Our training strategy can be outlined as followed: First, recent messages coming from the nodes $P^{(h)}$ and $D^{(h)}$ are recorded. Second, deviations $\Delta_{(t-\tau^{(h+1)},t)} s_j^{(h+1)}$ between the two messages coming from $S^{(h+1)}$ at timesteps $t-\tau$ and t are computed. These deviations define how much the recent history served to get closer or farther away from states in $S^{(h+1)}$. Third, the parameters $f_{ijk}^{(h)}$ are changed proportionally to the product of the recorded recent messages and the deviations $\Delta_{(t-\tau^{(h+1)},t)} s_j^{(h+1)}$.

The following equations define the change in $f_{ijk}^{(h)}$ at each timestep:

$$trace_{ik}^{(h)}(t) = \sum_{u=t-\tau^{(h)}}^{t} p_i^{(h)}(u - \tau^{(h)} + 1) \cdot d_k^{(h)}(u) \tag{4.13}$$

$$\Delta f_{ijk}^{(h)} = \eta^{(h)} \cdot \Delta_{(t-\tau^{(h+1)},t)} s_j^{(h+1)} \cdot trace_{ik}^{(h)}(t) \tag{4.14}$$

Here, $\eta^{(h)}$ is a learning rate that needs to be set to sufficiently small values to allow a

slow convergence. We used $\eta^{(0)} = 1.5 \cdot 10^{-4}$ and $\eta^{(1)} = 3 \cdot 10^{-3}$. $\tau^{(h)}$ defines the timeframe that is considered at a certain level.

The average number of timesteps that need to pass until a certain change in activity can be observed in messages sent from $S^{(h)}$ depends on the level h. In general, this time constant could be adjusted by observing the speed of change. For simplicity, we set $\tau^{(h)}$ to 2^h. This setting reflects the exponential increase of region sizes with increasing hierarchy level and yields usable results in our configurations.

During training, the messages coming from $D^{(h)}$ encode a position that is reached one timestep after the state encoded in the message from $P^{(h)}$. When using the network to generate action sequences, however, $F^{(h)}$ should generate a message $g^{(h)}$ which encodes a subgoal that can be reached within $\tau^{(h)}$ timesteps. Therefore, an additional delay of $+1$ timesteps is assigned to messages coming from node $P^{(h)}$ during training as can be seen in equation 4.13.

4.5.2 Generating useful streams of data

The performance of a trained hierarchical network depends largely on the quality of the provided training data. Since we generate coarse representations by analyzing the stream of training data, action sequences executed while recording training data influence the topology of these representations. In general, neighboring regions will be grouped if transitions between them occured often. Similarly, inter-level instructor nodes will only know about transitions between two regions at their respective hierarchy level if transitions between them occured.

Taken together, it is desirable to provide training data that contains many transitions between different regions at all levels of the hierarchy. While it might be suitable to use data from a simple random walk to train the lowest hierarchy level, such data is too erratic to train higher levels of the hierarchy. With increasing hierarchy level, training data needs to contain longer paths.

We use a simple online training method, where a partially trained HNN network is used to generate data which is suitable to further train the same network. An agent is placed at a random position in an unknown maze and controlled by a controller that is based on an initially untrained HNN. This controller first executes a certain number of random actions before it selects a goal from the set of positions that have been visited already, favoring positions that have been visited fewer times in the past. The HNN is then employed to reach this goal: The current position and the destination are presented to the network. An action is then selected with a probability proportional to the entries in the resulting message arriving at node A. Actions are iteratively selected in this way and executed until the agent reaches the selected goal or a predefined maximum of actions

is executed. After that, the controller will again execute a few random action before selecting a new goal. These additional random action allows to discover new positions in the map. This process is repeated and the collected data is in parallel used to train the network.

By employing the partially trained HNN in the described way, it becomes possible to generate data that contains longer and straighter paths through the maze. As soon as one level of the hierarchy is trained, actions or action sequences that cause no transition on this level of the hierarchy will automatically be avoided because the HNN will select those actions that cause transitions towards the goal. This allows to generate better data for training the next hierarchy level.

4.5.3 Using a trained HNN for navigation

Given a current position and a destination, the output of a trained HNN consists of a message that assigns one value to each possible action. Each of these values represents a belief about the optimality of the corresponding action. A controller can use this vector to determine an action for a given configuration. In the last section, we described a controller that chooses an action with a probability proportional to the corresponding response value. This strategy bears the advantage of easily incorporating an explorative element.

In order to quickly move from a position to a destination, it is desirable to reduce the amount of exploration. Therefore, we chose a different controlling strategy when evaluating a trained HNN: our controller picks actions with probability proportional to the square of the corresponding entry in the response vector. This procedure can be interpreted as a soft-max on the response vector. It causes a stronger focus on those actions that receive the highest response values.

4.6 Simulations

We tested our algorithms by applying them to navigation tasks on random undirected graphs. All graphs contained 36 nodes of degree 4. Edges were randomly added while enforcing 40% self-edges and finally selecting only graphs with diameter 16.

For each navigation environment, we created a HNN with 3 hierarchy levels. Variable nodes of the first, second and third level were fixed to 36, 18 and 6 states, respectively. Information about the current and future state was transmitted to the nodes $P^{(0)}$ and $D^{(0)}$ through additional multiplicative nodes $V^{(-1)}$ and $W^{(-1)}$ similar to the $V^{(h)}$- and $W^{(h)}$-nodes in figure 4.7, which were also trained as described in section 4.5.1, but with small time constants ($V^{(-1)}$: $\gamma = 0.01$, $W^{(-1)}$: $\gamma = 0.1$).

All parameters in these networks were initialized to constant values, ensuring that the network contained no information about the topology of the environment. Parameters were then trained using a stream of position data that was generated by a controller as described in section 4.5.2.

Training was executed in 2 phases. During the first phase, learning rates for creating coarse representations (ε in forumula 4.12) were set to large values which ensured fast adaption of parameters ($\varepsilon = 0.006$ for $V^{(-1)}$ and $W^{(-1)}$, otherwise $\varepsilon = 0.0015$). Time constants γ were also set to large values ($V^{(0)}$ and $W^{(0)}$: $\gamma = 0.88$, $V^{(1)}$ and $W^{(1)}$: $\gamma = 0.95$), in order to cause strong competition between high-level states, which causes high-level regions to become more compact and significantly reduces the occurrence of high-level regions that respond to disconnected regions in the maze.

All these learning parameters were decreased during the second phase, which helped to achieve better network performance after learning: Decreasing ε helped to avoid significant shifts of the response regions of higher-level states ($V^{(-1)}$ and $W^{(-1)}$: $\varepsilon = 0.0002$, otherwise $\varepsilon = 0.0001$). Such shifts could make it necessary to re-learn the parameters of the instructor nodes, since these are tied to a certain representation. Lower time constants resulted in high-level regions with sharper boundaries ($V^{(0)}$ and $W^{(0)}$: $\gamma = 0.4$, $V^{(1)}$ and $W^{(1)}$: $\gamma = 0.6$). In environments with such regions, messages at upper hierarchy levels have a more precise meaning, which improves the overall network performance after training.

Introducing two phases during learning might at first glance seem like conflicting with the dogma of neural plausibility, since it requires some external mechanism that evaluates the overall states of training and sends a signal to all nodes. However, the parameter changes could happen gradually: Each node extracting a higher level state could continually adapt its learning constant, based on the amount of data it has observed.

4.6.1 Evaluation

Computing average path lengths

In order to evaluate the performance of our method, we compare the average path lengths an HNN-based controlling strategy yields with the optimal path lenghts. Interestingly, it is possible to analytically compute the average path lengths in this setup.

When given a current position and a destination, the controlling strategy described in section 4.5.3 uses the output of a trained HNN to assign a probability to each possible action. During analysis, the result of these actions is known, which makes it possible to calculate a vector containing transition probabilities for a given position and destination. This can be done for all starting positions and a fixed goal, resulting in a transition

probability matrix Γ for a fixed goal. We are interested in the average pathlengths ξ_{ij} between each position i and a given destination j. If $i = j$, no action needs to executed, and therefore $\xi_{jj} = 0$. For all other positions i, the path is one step longer than the average pathlength from those positions that can be reached from i in one step, weighted by the associated transition probability.

In matrix notation, we get:

$$\vec{\xi_j} = \Gamma \vec{\xi_j} + \vec{u} \tag{4.15}$$

$\vec{u}$ contains the observations described above: All its entries are equal to 1, except for the jth entry, which is 0. The transition probabilities from position j should all be set to 0 in order to make sure that $\xi_{jj} = 0$. By solving this system of linear equations for $\vec{\xi_j}$, we obtain a vector containing the average pathlength from all positions to position j. This process can be repeated for all possible destinations in order to obtain the average path lengths for all possible pairs of start and destination.

Tests on random graphs

We define the *path penalty* as the ratio between a computed average path length and the minimal distance as obtained from graph analysis. A path penalty of 1.10 corresponds to an average detour of 10% compared to the optimal path.

We generated 1000 random undirected graphs and for each one trained an HNN for 70000 timesteps. Every 2000 timesteps, we measured all path penalties. These were grouped into six groups for goals at a distance of 1, 2, 3 to 4, 5 to 8, 9 to 12 and 13 to 16 steps. In four test runs, the performance of the network degraded exponentially for certain paths, which was caused by higher level regions that responded to disconnected parts in the graph. We believe this problem could be avoided by making the representations at higher levels overcomplete, with several states responding to every position. We excluded those four test runs from the analysis in figure 4.8, which shows how the path penalties improve during learning, averaged over all paths in all test runs. The same analysis was performed for a noisy environment, in which selected actions were ignored 20% of the time and replaced by a random action. Figure 4.9 shows the results, averaged over 999 of 1000 runs.

The evolution of the path penalties expresses an interesting behaviour: While path penalties for longer paths converge later during training, they converge approximately at the same speed as the path penalties for shorter paths (except for those for goals at distance 1). In addition, the relation between convergence time and path length seems to be non-linear: Penalties for goals at distances of 13 to 16 steps converge about 2 times slower than path penalties for goals at distances of 3 to 4 steps. Noise up to the tested degree does not seem to have a huge impact on training success.

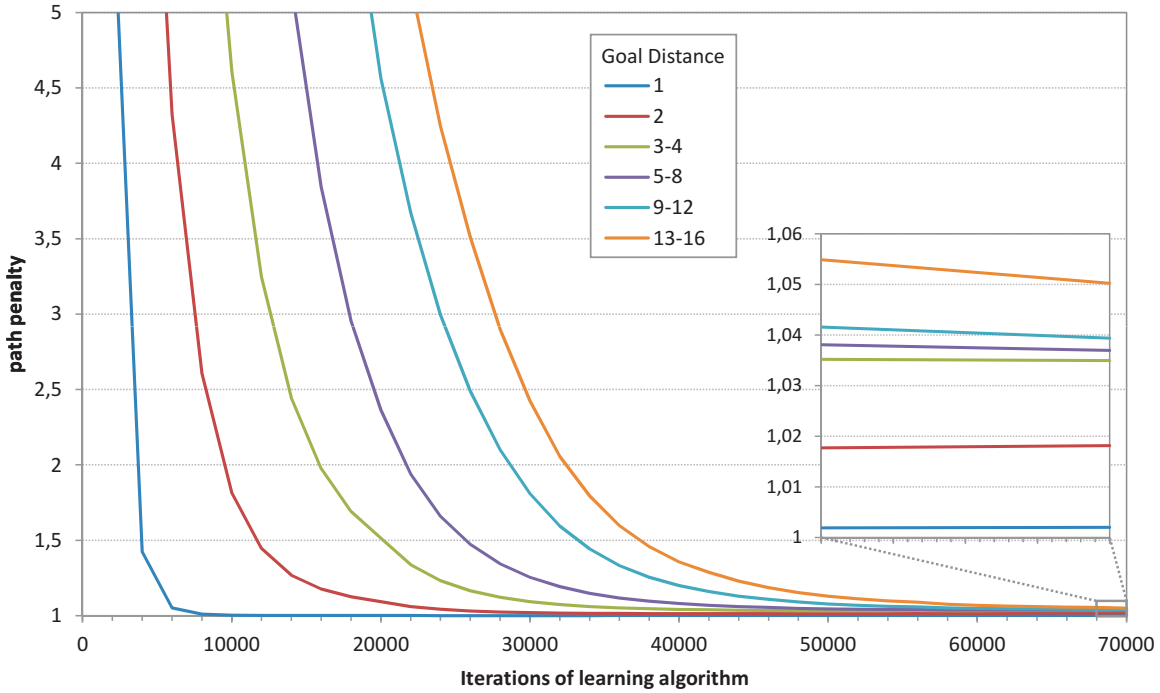

Figure 4.8: *Performance of HNNs as a function of training rounds. The figure shows the evolution of measured path penalties (see text for explanation) as a function of training iterations, where each iteration corresponds to one observed data point. Each of the lines shows the evolution of path penalties for a specific goal distance range (goal distances were grouped in goal distances of* 1, 2, 3-4, 5-8, 9-12 *or* 13-16 *steps). Curves show averages over 996 out of 1000 runs. In the remaining 4, path penalties did not converge because one of the high-level regions was disconnected. Final path penalties in the plot are* ≤ 1.05.

This behaviour matches our expectations: Longer paths cannot be learned by a HNN as long as the shorter paths are not properly understood. As soon as these are properly modelled, longer paths can be modelled quickly.

4.7 Discussion

In this chapter, we describe how planning and execution of action sequences can be achieved using a fixed graphical model in the context of graph-based navigation tasks.

The hierarchical navigation networks (HNN) that we describe in this chapter rely on a hierarchical decomposition of space to which the granularity of processing is matched. Connections at each hierarchy level express possible spatial transitions which are local at the respective level scale. Higher level connections constrain possible transitions at the lower levels. The basic organization of these networks is given by the programmer, whereas the connections at each level are learned from experience.

Fine-grained information about the current position of an agent and a destination are

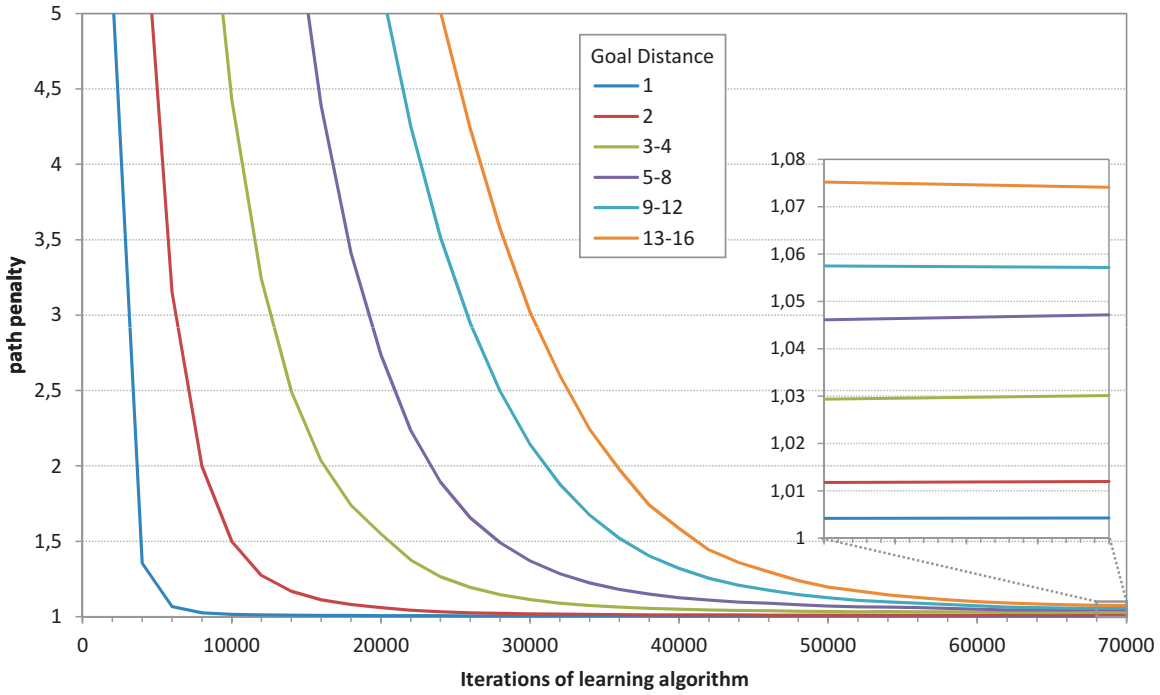

Figure 4.9: *Performance of HNNs as a function of training rounds for a noisy system. The figure shows the same plot as figure 4.8, but for a system with noisy observations during training: At each step, a random action was selected with probability* 0.2*. Curves show averages over 999 out of 1000 runs. Final path penalties in the plot are* ≤ 1.07*.*

fed into nodes at the lowest level of the network. This information propagates through the levels of the spatial hierarchy and simultaneously, each level resolves possible paths between source and destination at that scale, constrained by solutions at the adjacent level. Overall, the network solves a feasible route between a source and a destination and the process imposes the next elementary action at the lowest level of the hierarchy.

Our approach demonstrates how to implement navigation using a fixed graphical model. In contrast to most approaches from reinforcement learning or classical graph-based navigation, it does not require a procedural processing style but instead relies entirely on a message passing scheme between nodes of a network. All messages that are sent between nodes inside the network only require simple computation at the nodes and most messages are created following the belief propagation algorithm [Pea88] except for messages that combine information from two different levels of the hierarchy, which in turn are computed by weighted addition.

Graphical models such as factor graphs have been shown to be computationally efficient for the implementation of many algorithms from various fields such as artificial intelligence, lattice physics, and digital communcations [KFL01]. These implementations take advantage of the factorized representation of a global function with many parameters.

Factor graphs have also been applied to temporal processes. For example, the Kalman filter can be implemented as a factor graph [Kal60, KFL01]. Factor graphs can also be used in motor control, where time series need to be predicted [TG10]. All approaches we know of that apply graphical models to temporal processes employ a time-slicing scheme to model temporal dynamics. That is, the same graph structure is repeated for every timestep that should be modeled. When dealing with long timescales, this time slicing can lead to very large graph structures and unfeasibly long computation times.

In an HNN , each layer of the hierarchy works on a different abstraction level of input space and time. The hierarchical approach thereby leads to a hierarchical factorization of the navigation problem. We therefore think that HNNs introduce a novel approach of how to model time in graphical models.In the current version, the current and the future state of the agent are encoded in single variables. Hence, the input variables are not encoded in a factorized representation, which makes it difficult to apply HNNs to problems with high dimensional state space. We are optimistic that a factorized representation of space can be incorporated in future extensions of our approach.

In section 4.5.1, we described simple online methods of how the parameters of an HNN can be trained. The algorithms we presented are designed to work locally on single nodes of the network and only require information about recent messages that were visible to the respective node.

We showed that useful decompositions of space can be generated using such a local

learning rule. However, the presented algorithms do not guarantee any optimality of the generated decompositions. Better decompositions of space could be obtained using clustering methods that take into account global information. In a continuous environment, the spatial representation at higher hierarchy levels could also be generated using slow feature analysis [WS02]. As Wiskott et al. showed, it is possible to use this method in a hierarchical fashion which allows one to extract features that change at very different timescales. Such a hierarchical arrangement was also employed by Legenstein et al. in the context of reinforcement learning, where the behaviour of an agent was trained based on the slow features of a high-dimensional input [LWW10].

In a recent paper, Hopfield proposes a neurally plausible model for navigation [Hop10]. In this model, each position in space causes a firing of a specifically tuned spiking neuron, called a *place cell*. During maze exploration, each place cell is trained to connect to those place cells which are tuned to regions adjacent to the cell's receptive field. Navigation tasks are then solved by executing a 'mental exploration' phase, in which firing activity propagates in random directions starting from a place cell representing the current position until the place cell representing the destination becomes active. The neural activity path that led to this cell can then guide an animal to the destination.

We believe that the hierarchical approach we describe could guide to the development of an extension to Hopfield's detailed neurally plausible model. This would allow one to replace the random mental exploration phase with a much more effective hierarchical search.

The HNNs as described in this chapter form a proof-of-concept of how a single message-passing network can be trained to represent an environment and then be used for navigation. These models are composed of many small computational units that each work with locally available information, both during training and during application to navigation tasks. Importantly, our models can be extended to larger state spaces and more hierarchy levels in a straight-forward way. Concluding, we introduced a new concept of how different spatial and temporal scales can be incorporated in a graphical model.

Chapter 5

Discussion

5.1 The big picture

In this thesis, we have pursued a novel approach to motor control and motion planning. We demonstrated how graphical models can be used to model the joint probability distribution of instantaneous dynamics of a robotic system or to model the kinematics of a robot. In both cases, we described how to apply the probabilistic model in the context of control tasks. Finally, we introduced a new hierarchical method for representing time in a graphical model and applied this method to maze tasks that require the execution of sequences of actions. Alle models we presented form self-contained systems for control. They are trained solely from experience and require no prior information about the structure of the robotic system.

Robot kinematics in a factor graph

Kinematics of a robot describe the static dependences between angles and the spatial positions of the robot joints. This can be contrasted to the dynamics of a robot, which describe the temporal dependences between configurations of the robot. We used a factor graph to approximate the joint probability distribution of joint angles in a robotic arm and its end-effector position in section 3.7. This probability distribution captures a part of the kinematics of the robot. Consequently, this factor graph cannot predict the temporal dynamics of the robot arm. However, it can be used to infer the end-effector position when given a set of angles, but also vice versa for computing the inverse kinematics. It is appealing how the undirectedness of a factor graph that is trained solely from experience allows to compute both these functions. We described an algorithm that uses this factor graph to steer the robotic arm towards a desired position in space.

The drawback of this approach is that the motion of the robot arm is not explicitly computed by our model. Instead, a set of target angles is given and the joint angles

are simply adjusted linearly in order to approach this target. As a result, there is no intrinsic possibility to avoid collisions. In our setup, configurations that would cause collisions are avoided by applying a complex filter function that will rule out harmfull intermediate positions. In addition, this approach requires that the motor commands that need to be applied in order to reach the target configuration do not depend on the starting configuration.

Instantaneous dynamics in a factor graph

In a robotic system where selection of an action depends on the desired target and the current position, a factor graph representing the kinematics cannot provide sufficient control. This is for example the case in the simple control task on a compliant pendulum that was considered in this thesis project: While an applied angular torque of 0 is the correct command in order to hold the pendulum in an upright position, this torque is not helpfull when attempting to reach this state (see sections 3.2 and 3.4).

In order to overcome this problem, we used a factor graph to represent the instantaneous dynamics of the robotic system. A factor graph was trained to relate actions with the state of the system before and after executing these actions. This factor graph was used in a control task to infer suitable motor commands in order to cause a transition to a desired state.

The major drawback of this approach of using a factor graph to model instantaneous dynamics of a robotic system is its limited reach: The graph can only be used to infer actions in situations where the goal can be reached with a single action.

Graphical models for representing temporal sequences

The common approach to overcome this issue is to view the dynamics over N timesteps as a repetition of single-timestep dynamics. In a factor graph representation, this would correspond to a graph with N variable nodes representing the actions at each timestep and $N + 1$ variable nodes representing the starting and target state as well as all intermediate states, as for example sketched in figure 4.2b. Such a time-slicing approach was successfully used for time-series analysis like the Kalman filter [Kal60], but also in the context of motor control [TG10, Tou09]. These approaches have also been extended to systems in which the dynamics of a system is non-markovian and depends on the recent two timesteps [ML09]. All described time-slicing approaches require an iterative procedure on a function representing instantaneous dynamics when analyzing sequences.

We pursued a different approach, which uses a hierarchical decomposition of space and time (see chapter 4). The layers in our hierarchial graphical models each act similar to a factor graph representing instantaneous dynamics. However, each layer operates on

a different temporal and spatial scale. Higher levels that consider larger time frames can impose subgoals onto lower levels of the hierarchy. Subgoals that are imposed from a higher level of the hierarchy to a lower level can be reached within a single timestep at that level. Hence, the instantaneous dynamics stored at that level are sufficient in order to select an action, which imposes a subgoal onto an even lower level. At the lowest level, the selected action corresponds to a real action, which will then be executed.

5.2 Neural plausibility

In several studies within the past decade, it has been suggested that human behaviour is optimal in a bayesian sense. This seems to be the case in visual perception [KMY04], but also when combining visual and haptic cues [EB02] and in sensorimotor learning [KW04]. Consequently, it is not foolish to describe human behaviour using probabilistic models, which approximate the joint probability distribution over observed data.

Factor graphs represent an elegant way to model probability distriubtions. They can be used to approximate functions with many parameters as products of smaller functions. Inference calculations can be performed using belief propagation, which relies on message passing between small computational units. Interestingly, factor graphs share several qualitative properties with nervous systems: First, computations are distributed to many small computational units: In a factor graph, these units are formed by nodes, whereas neurons or ensembles of neurons are commonly regarded as the computational units of brains.

Second, each computational unit operates solely on local data, consisting of messages that are sent to this unit and a set of parameters, which in turn can also be trained based on messages. Nodes in a factor graph can only access information that is transmitted to them through population-code like messages. Similarly, all information a single neuron can access must be locally available. Third, factor graphs express a uniform structure, with similar operations being executed in all parts of the graph. Likewise, the mammalian cortex expresses a remarkably uniformity [HW74, RHP80] and the same computational microstructure can be identified in many parts of the mammalian cortex [DM04, DMW89].

The hierarchical navigation networks that we presented introduce a new concept of how to model temporal processes in a graphical model. We believe that this approach is neurally more plausible than the common approach to model time by time-slicing. First, our method does not require any iterative procedure as used in time-slicing approachs. This iterative procedure would either require a highly repetitive and redundant structure or a iterate many times over the same structure, but require the storage of temporary results.

In an HNN , actions at higher hierarchy levels are described in terms of subgoals at the lower levels. These higher level actions could be interpreted as being motor primitives, which can be executed in various contexts. The model therefore matches the common thinking that the brain stores such motor primitives [FH05].

If an HNN would be used to model a complex robotic system, single states of variables at lower levels in the hierarchy would most likely respond to specific body configurations. At higher levels though, these states would most likely be tuned to regions in space, independent of a specific body configuration. In neuroscience, units that respond to regions in space are known as *place cells*, which have been discovered in rat hippocampus [OD71]. In repetitive environments, higher-level states in an HNN could also become tuned to grid-like structures, similar to response regions of grid cells [HFM$^+$05, DBB10]. Grid cells can have grid-like response regions of different different scales [BST$^+$10]. A similar organization would be expected at high levels in an HNN : Response region sizes increase with increasing hierarchy level.

5.3 Conclusion

The three approaches to various motor control and planning tasks we have described in this thesis are by no means competitive with the best algorithms available today. However, our aim was not to improve existing techniques, but to make a step into a new direction in order to develop a different perspective on motor control problems.

We made certain assumptions about the available computational architecture, which were chosen based on the computational properties expressed by brains. The resulting algorithms have interesting properties, such as their ability to learn to control a system without any prior knowledge. Information is processed in a non-procedural way and motor actions are selected based on the result of a sequence of simple multiplications and summations. In addition, all models we presented rely on distributed computation, where the individual computational units can only access local information.

Two main aspects of motor control problems were modelled in this thesis and the associated complex functions were broken apart into smaller functions: First, we demonstrated that the topological properties of a complex robot can be factorized into a product of low-dimensional functions. Second, a novel kind of graphical models was presented, which allow to partition the temporal dependencies of a controllable system into smaller functions. We believe that these two approaches could be combined in the future and therefore form important steps towards graphical models for robotic control.

5.4 Outlook

On the one hand, we showed how to model the kinematics of a robot can be factorized and represented as a factor graph. On the other, we introduced a new concept of how to model temporal processes in graphical models. In the former approach spatial aspects of a robot were modeled, while temporal dynamices were completely ignored. In contrast, the latter approach provided a factorization of temporal dynamics, but up to now includes no spatial factorization, which limits its usability in large state spaces. Future work will hopefully allow to combine both approaches: If it would be possible to include a fatorization of the state in an HNN , these models could prove to be usefull in much higher-dimensional tasks.

All factor graphs in this thesis were hand-crafted. While experiments showed that the precise topology had very little influence on system performance in our setups, this is not necessarily the case for higher-dimensional systems. In such systems, defining a usefull graph topology might be much more intricate. Several studies suggest methods of how to learn structure in graphical models, but none of these considers networks with hidden variables [PPL97, Mcc03, LGK07]. It is promising to extend this research to factor graphs with hidden states, which would allow to construct fully self-contained control algorithms that can learn to control a robotic system without any user input.

In our approaches to robot control, the precision of control was coupled to the overall number of parameters. In order to increase the precision of factor-graph-based control algorithms, it would be desirable to choose a different parameterization for the contents of factor nodes. Representing factor nodes with continuous functions such as gaussian mixture models could significantly reduce the overall number of parameters, which could result in better generalization properties and therefore require much less training data. Continuous high-level regions in an HNN could be extracted from continuous lower-level representations using slow feature analysis [WS02].

Appendix A: Factor graphs

A.1 History of the belief propagation algorithm

The belief propagation algorithm was developed separately in the fields of artificial intelligence, coding theory as well as in statistical physics. The success of the belief propagation algorithm and the fact that it was independently developed in various fields underline its flexibility.

In the artificial intelligence literature, the algorithm was originally developed for inference calculations on Bayesian networks and was used in expert systems, for example for medical diagnosis [Pea88, Jen96]. Several successful algorithms from the field of coding theory can be expressed as instances of the belief propagation algorithm on factor graphs, such as the Viterbi algorithm [Vit67] and the forward-backward algorithm [BP66]. The very successfull turbo codes, which are near-shannon limit error correcting codes can be described as belief propagation on a loopy factor graph [BGT93]. The similarities between these algorithms were discovered by Aji an McEliece [AM00].

Shortly after that, Yedida, Freeman and Weiss discovered a close connection between the belief propagation algorithm and the Bethe approximation of statistical physics [YFW03].

A.2 Factor graphs

Factor graphs are visual representations of functions over sets of variables which can be described as products of lower-dimensional functions. Consider for example a high-dimensional function G, which is defined over variables Y^1 to Y^6, H^1 and H^2 and assume that this function can be expressed as a product of smaller functions as follows:

$$\begin{aligned} &G(Y^1, Y^2, Y^3, Y^4, Y^5, Y^6, H^1, H^2) \\ &\qquad = f_1(Y^1, Y^2, H^1) \cdot f_2(Y^3, H^1, H^2) \cdot f_3(Y^4, Y^5, H^2) \cdot f_4(Y^6, H^2) \end{aligned} \tag{A.1}$$

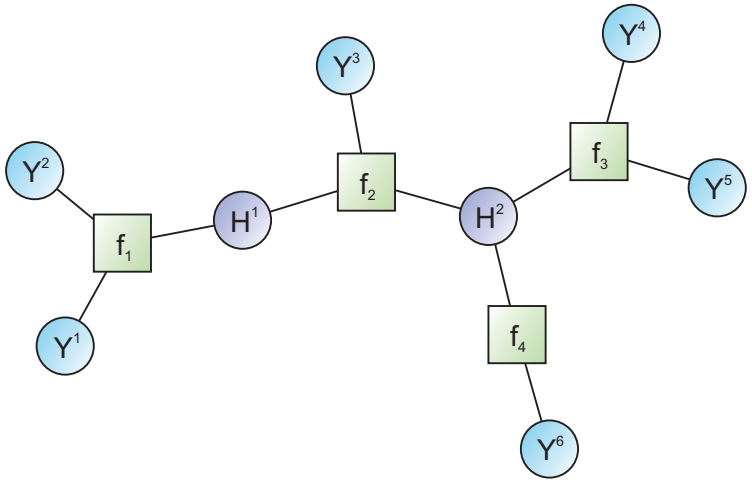

Figure A.1: *Visualization of a function that can be factorized as a factor graph. Here, the function* $f_1(Y^1, Y^2, H^1) \cdot f_2(Y^3, H^1, H^2) \cdot f_3(Y^4, Y^5, H^2) \cdot f_3(Y^6, H^2)$ *is displayed: Circles indicate variable nodes that can be either be associated to real-world observations (observed Variables, blue) or hidden variables (purple). Squares indicate relations between the variables, which are associated with a function* f_a*, which is defined over the variables that are connected to its node.*

In order to visualize a such a function as a factor graph, a variable node is drawn for each variable (typically drawn as circles) and each is function f_a is assigned a factor node (drawn as rectangles). Each factor node then gets connected to all variable nodes that it is defined on. In this way, a bipartite graph is created, with variable nodes in the first and factor nodes in the second set of nodes. Figure A.1 shows a visualization of the function chosen above.

In general, the variables in a factor graph can take on one of possibly infinitely many values, which allows the use of continuous variables. In this thesis, however, we consider functions over variables that can take on values from a limited alphabet, which we will denote the *discrete states* of these variables. Variables can be distinguished into a set $\boldsymbol{Y} = \{Y^1, \ldots, Y^{|\boldsymbol{Y}|}\}$ of observed and a set $\boldsymbol{H} = \{H^1, \ldots, H^{|\boldsymbol{H}|}\}$ of hidden variables. Observed variables are directly associated with some observable quantity, whereas hidden variables refer to an internal state. The cartesian product of the two sets will be denoted by $\boldsymbol{X} = \boldsymbol{Y} \times \boldsymbol{H} = \{Y^1, \ldots, Y^{|\boldsymbol{Y}|}, H^1, \ldots, H^{|\boldsymbol{H}|}\}$. We will use the notation X^i to refer to the ith variable in this set.

In analogy to the example above, any function G that depends on the variables in $\boldsymbol{X}$ and factorizes into a product of functions can be written as follows:

$$G(\boldsymbol{X}) = \prod_a f_a(X_a) \tag{A.2}$$

Here, a is an index to identify the individual functions f_a and X_a denotes the subset of $\boldsymbol{X}$ that a function f_a depends on.

A.3 Belief propagation

By factorizing a high-dimensional function into a product of lower-dimensional functions, the number of parameters required to define the function can be reduced. More importantly, certain properties of the function can be calculated more efficiently by using a factorized representation than by using the a naive representation. In particular, this includes the calculation of certain marginal function values, which are obtained by summing the function values for all possible configurations of a selected set of variables. In the example from equation A.1, one might for example want to compute the following marginal probability:

$$G(Y^1) = \sum_{Y^2,Y^3,Y^4,Y^5,Y^6,H^1,H^2} G(Y^1,Y^2,Y^3,Y^4,Y^5,Y^6,H^1,H^2) \tag{A.3}$$

$$= \sum_{Y^2,Y^3,Y^4,Y^5,Y^6,H^1,H^2} f_1(Y^1,Y^2,H^1)\cdot f_2(Y^3,H^1,H^2)\cdot f_3(Y^4,Y^5,H^2)\cdot f_4(Y^6,H^2) \tag{A.4}$$

Here, the global function is summed over all variables except Y^1. The number of terms that need to be summed therefore increases exponentially with increasing number of variables. However, the distributive law allows to rewrite this function as follows:

$$G(Y^1) = \sum_{Y^2,Y^3,Y^4,Y^5,Y^6,H^1,H^2} G(Y^1,Y^2,Y^3,Y^4,Y^5,Y^6,H^1,H^2) \tag{A.5}$$

$$= \sum_{Y^2,H^1} f_1(Y^1,Y^2,H^1) \sum_{Y^3,H^2} f_2(Y^3,H^1,H^2)\cdot\left(\sum_{Y^4,Y^5} f_3(Y^4,Y^5,H^2)\right)\cdot\left(\sum_{Y^6} f_4(Y^6,H^2)\right) \tag{A.6}$$

Interestingly, the summation over certain variables can be separated from the other summations. In effect, it is for example sufficient to compute the sum $\sum_{Y^6} f_4(Y^6,H^2)$ once for each possible assignment to H^2. These values can then be reused for each assignment to the other variables. Similarly, the sum over Y^4 and Y^5 can be precomputed to yield another vector of values. Both can then be used to compute a vector of values for the sum over Y^3 and H^2 and so on. In a factor graph, the calculated temporary results can be interpreted as messages that are sent between the nodes: the nodes f_4 and f_3 each send a vector of values to the node H^2, where the vectors are multiplied and forwarded to the node f_2 and so on.

The belief propagation algorithm generalizes this observation to arbitrary tree-shaped factor graphs. Two rules define how outgoing messages are calculated at variable nodes and at factor nodes.

The message $m_{X^i \to f_a}$ that is sent from a variable node X^i to a factor node f_a is equal to the product of all messages sent to X^i, except the message coming from f_a:

$$m_{X^i \to f_a}(k) = \prod_{f_{a'} \in Adj(X^i) \setminus \{f_a\}} m_{f_{a'} \to X^i}(k) \tag{A.7}$$

Here, $Adj(X^i)$ is the set of factor nodes adjacent to X^i.

At a factor node f_a, a message $m_{f_a \to X^i}$ to a variable node X^i is calculated by multiplying the function values $f_a(X_a)$ with the incoming messages from all nodes except X^i:

$$m_{f_a \to X^i}(k_i) = \sum_{\{k_1, k_2, \dots\} \setminus \{k_i\}} \left(f_a(k_1, k_2, \dots) \prod_{X^m \in Adj(f_a) \setminus \{X^i\}} m_{X^m \to f_{a'}}(k_m) \right) \tag{A.8}$$

Here, the k_i refer to a specific discrete state of the ith variable connected to f_a. The function values $f_a(k_1, k_2, \dots)$ essentially get multiplied with the values in the incoming messages that correspond to the respective coordinate. For fixed k_i, the sum is then computed over all these products to yield the $k_i$the element of the outgoing message.

If all messages in a non-loopy factor graph are calculated according to these rules, it can be shown that marginal function values for selected variables can be calculated by multiplying all incoming messages at that node [KFL01].

A.4 Joint probability distribution as a factor graph

In general, factor graphs can represent any factorizable function. Oftentimes, when probabilistic relations are modelled, the function of interest is the joint probability distribution (JPD) over a set of variables: $P(\boldsymbol{X})$.

Since a probability distribution must sum up to 1, a factorized representation of the function usually needs to be the factorized function is usually needs to be normalized:

A.4.1 Joint probability function as a factor graph

$$P(\boldsymbol{X}) = \frac{1}{Z} \prod_a f_a(X_a) \tag{A.9}$$

The normalization factor Z is called the *partition function* and is defined as follows:

$$Z = \sum_{\boldsymbol{X}} \prod_{a} f_a(X_a) \tag{A.10}$$

Importantly, the marginal function values of the JPD have interesting relevance. For example, the marginal distribution over a variable X^i is equal to the prior probability distribution of that variable. Moreover, it is possible to compute conditional probability distributions of a single variable X^i given obervations about observed variables. In that case, the messages from the observed variables to their adjacent factor nodes simply need to be fixed to a single state. This will effectively avoid the summation over this variable. By normalizing the product of the message to the node X^i, the conditional probability distribution is obtained.

List of Figures

List of Tables

Bibliography

[AM00] Srinivas M. Aji and Robert J. McEliece, *The generalized distributive law*, IEEE Transactions on Information Theory **46** (2000), no. 2, 325–343.

[BGT93] C. Berrou, A. Glavieux, and P. Thitimajshima, *Near Shannon limit error-correcting coding and decoding: Turbo-codes (1)*, IEEE International Conference on Communications (ICC) **2** (1993).

[BM03] Andrew G. Barto and Sridhar Mahadevan, *Recent Advances in Hierarchical Reinforcement Learning*, Discrete Event Dynamic Systems **13** (2003), no. 4, 341–379.

[BP66] Leonard E. Baum and Ted Petrie, *Statistical Inference for Probabilistic Functions of Finite State Markov Chains*, The Annals of Mathematical Statistics **37** (1966), no. 6, 1554–1563.

[BST+10] Charlotte N. Boccara, Francesca Sargolini, Veslemøy Hult H. Thoresen, Trygve Solstad, Menno P. Witter, Edvard I. Moser, and May-Britt B. Moser, *Grid cells in pre- and parasubiculum.*, Nature neuroscience **13** (2010), no. 8, 987–994.

[Con08] Jörg Conradt, *A distributed cognitive map for spatial navigation based on graphically organized place agents.*, Ph.D. thesis, ETH Zurich, 2008.

[DBB10] Christian F. Doeller, Caswell Barry, and Neil Burgess, *Evidence for grid cells in a human memory network.*, Nature **463** (2010), no. 7281, 657–661.

[DH93] Peter Dayan and Geoffrey E. Hinton, *Feudal reinforcement learning*, Advances in Neural Information Processing Systems 5, Morgan Kaufmann, 1993, pp. 271–278.

[Dij59] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), no. 1, 269–271.

[DIPR07] K. Doya, S. Ishii, A. Pouget, and R.P.N. Rao, *Bayesian brain - probabilistic approaches to neural coding*, MIT Press, Cambridge Massachusetts, 2007.

[DM04] Rodney J. Douglas and Kevan A. Martin, *Neuronal circuits of the neocortex.*, Annual Reviews in Neuroscience **27** (2004), 419–451.

[DMW89] Rodney J. Douglas, Kevan A. C. Martin, and David Whitteridge, *A canonical microcircuit for neocortex*, Neural Comput. **1** (1989), 480–488.

[EB02] Marc O. Ernst and Martin S. Banks, *Humans integrate visual and haptic information in a statistically optimal fashion.*, Nature **415** (2002), no. 6870, 429–433.

[FH05] T. Flash and B. Hochner, *Motor primitives in vertebrates and invertebrates.*, Curr Opin Neurobiol **15** (2005), no. 6, 660–666.

[FMD00] David Foster, Richard Morris, and Peter Dayan, *A model of hippocampally dependent navigation, using the temporal difference learning rule*, Hippocampus **10** (2000), no. 1, 1–16.

[HFM+05] Torkel Hafting, Marianne Fyhn, Sturla Molden, May-Britt Moser, and Edvard I. Moser, *Microstructure of a spatial map in the entorhinal cortex*, Nature **436** (2005), no. 7052, 801–806.

[HKK+07] Kensuke Harada, Shuuji Kajita, Fumio Kanehiro, Kiyoshi Fujiwara, Kenji Kaneko, Kazuhito Yokoi, and Hirohisa Hirukawa, *Real-time planning of humanoid robot's gait for force-controlled manipulation*, Mechatronics, IEEE/ASME Transactions on **12** (2007), no. 1, 53 –62.

[Hop10] John J. Hopfield, *Neurodynamics of mental exploration*, Proceedings of the National Academy of Sciences **107** (2010), no. 4, 1648–1653.

[HW74] David H. Hubel and Torsten N. Wiesel, *Uniformity of monkey striate cortex: A parallel relationship between field size, scatter, and magnification factor*, The Journal of Comparative Neurology **158** (1974), no. 3, 295–305.

[Jen96] F.V. Jensen, *An introduction to bayesian networks*, no. Bd. 1, Springer, 1996.

[Kal60] R. E. Kalman, *A New Approach to Linear Filtering and Prediction Problems*, Transactions of the ASME – Journal of Basic Engineering (1960), no. 82 (Series D), 35–45.

[Kel08] Andreas Keller, *Pendulum design*, Semester work, Institute of Neuroinformatics, University of Zurich and ETH Zurich, Zurich, Switzerland, 2008.

[KFL01] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger, *Factor graphs and the sum-product algorithm*, IEEE Transactions on Information Theory **47** (2001), no. 2, 498–519.

[KKK+03] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kiyoshi Fujiwara, Kensuke Harada, Kazuhito Yokoi, and Hirohisa Hirukawa, *Resolved momentum control: Humanoid motion planning based on the linear and angular momentum*, Proceedings of the IEEERSJ International Conference on Intelligent Robots and Systems, vol. 2, 2003, pp. 1644–1650.

[KLP+08] M.M. Khan, D.R. Lester, L.A. Plana, A. Rast, X. Jin, E. Painkras, and S.B. Furber, *Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor*, Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on, june 2008, pp. 2849 –2856.

[KMY04] Daniel Kersten, Pascal Mamassian, and Alan Yuille, *Object perception as bayesian inference*, Annual Reviews in Psychology **55** (2004), 271–304.

[KP09] Jens Kober and Jan Peters, *Learning motor primitives for robotics*, pp. 2112–2118, IEEE, May 2009.

[KW04] Konrad P. Kording and Daniel M. Wolpert, *Bayesian integration in sensorimotor learning*, Nature **427** (2004), no. 6971, 244–247.

[LGK07] Su I. Lee, Varun Ganapathi, and Daphne Koller, *Efficient Structure Learning of Markov Networks using L1-Regularization*, Advances in Neural Information Processing Systems 19 (B. Schölkopf, J. Platt, and T. Hoffman, eds.), MIT Press, Cambridge, MA, 2007, pp. 817–824.

[LU09] S. Litvak and S. Ullman, *Cortical circuitry implementing graphical models*, Neural Computation **21** (2009), no. 11, 3010–3056.

[LWW10] Robert Legenstein, Niko Wilbert, and Laurenz Wiskott, *Reinforcement learning on slow features of high-dimensional input streams.*, PLoS computational biology **6** (2010), no. 8.

[MBLP06] Wei J. Ma, Jeffrey M. Beck, Peter E. Latham, and Alexandre Pouget, *Bayesian inference with probabilistic population codes*, Nature Neuroscience **9** (2006), no. 11, 1432–1438.

[Mcc03] Andrew Mccallum, *Efficiently inducing features of conditional random fields*, Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI03), 2003.

[Min22] Nicolas Minorsky, *Directional stability of automatically steered bodies*, Journal of the American Society for Naval Engineers **34** (1922), no. 2, 280–309.

[MK96] Geoffrey J. Mclachlan and Thriyambakam Krishnan, *The EM Algorithm and Extensions*, 1 ed., Wiley-Interscience, Hoboken, USA, November 1996.

[ML09] Piotr Mirowski and Yann LeCun, *Dynamic factor graphs for time series modeling*, Machine Learning and Knowledge Discovery in Databases (Wray Buntine, Marko Grobelnik, Dunja Mladenic, and John Shawe-Taylor, eds.), Lecture Notes in Computer Science, vol. 5782, Springer Berlin / Heidelberg, 2009, pp. 128–143.

[Neu11] Neuronics, *Katana robot arm*, `http://www.neuronics.ch/cms_en/web/index.php?id=244&s=katana`, 2011.

[Nok11] Nokia Corporation, *Qt - cross-platform application and UI framework*, `http://qt.nokia.com/`, 2011.

[OD71] J. O'Keefe and J. Dostrovsky, *The hippocampus as a spatial map. preliminary evidence from unit activity in the freely-moving rat*, Brain Research **34** (1971), no. 1, 171–175.

[Pea88] Judea Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, 1 ed., Morgan Kaufmann, September 1988.

[PPL97] Stephen Della Pietra, Vincent Della Pietra, and John Lafferty, *Inducing features of random fields*, IEEE Transactions on Pattern Analysis and Machine Intelligence **19** (1997), 380–393.

[PVS03] Jan Peters, Sethu Vijayakumar, and Stefan Schaal, *Reinforcement Learning for Humanoid Robotics*, Conference on Humanoid Robots, September 2003.

[Rao04] Rajesh P.N. Rao, *Bayesian Computation in Recurrent Neural Circuits*, Neural Computation **16** (2004), no. 1, 1–38.

[Rao05] Rajesh P. N. Rao, *Hierarchical bayesian inference in networks of spiking neurons*, Advances in Neural Information Processing Systems (Cambridge MA) (Lawrence K. Saul, Yair Weiss, and Léon Bottou, eds.), vol. 17, MIT Press, 2005.

[RHP80] A. J. Rockel, R. W. Hiorns, and T. P. Powell, *The basic uniformity in structure of the neocortex.*, Brain **103** (1980), no. 2, 221–244.

[Rol06] Jason Rolfe, *The cortex as a graphical model*, Master's thesis, California Institute of Technology, 2006.

[SB98] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*, The MIT Press, March 1998.

[SBC05] Satinder Singh, Andrew Barto, and Nuttapong Chentanez, *Intrinsically motivated reinforcement learning*, Proc. of the 18th Annual Conf. on Neural Information Processing Systems (NIPS'04), 2005.

[SBTS10] Freek Stulp, Jonas Buchli, Evangelos Theodorou, and Stefan Schaal, *Reinforcement learning of full-body humanoid motor skills*, 10th IEEE-RAS International Conference on Humanoid Robots, 2010, pp. 405–410.

[SCSG05] D. Sheynikhovich, R. Chavarriaga, T. Strösslin, and W. Gerstner, *Spatial representation and navigation in a bio-inspired robot*, Biomimetic Neural Learning for Intelligent Robots (2005), 98–98.

[Seu03] Seung, *Learning in Spiking Neural Networks by Reinforcement of Stochastic Synaptic Transmission*, Neuron **40** (2003), no. 6, 1063–1073.

[SMD09] A. Steimer, W. Maass, and R.J. Douglas, *Belief propagation in networks of spiking neurons*, Neural Computation **21** (2009), no. 9, 2502–2523.

[SPS99] Richard S. Sutton, Doina Precup, and Satinder P. Singh, *Between mdps and semi-MDPs: A framework for temporal abstraction in reinforcement learning*, Artificial Intelligence **112** (1999), no. 1-2, 181–211.

[Ste95] Anthony Stentz, *The Focussed D* Algorithm for Real-Time Replanning*, IJCAI, 1995, pp. 1652–1659.

[TBS10] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal, *Reinforcement learning of motor skills in high dimensions: A path integral approach*, ICRA, 2010, pp. 2397–2403.

[TG10] Marc Toussaint and Christian Goerick, *A bayesian view on motor control and planning*, From Motor Learning to Interaction Learning in Robots (2010), 227–252.

[Tou09] Marc Toussaint, *Robot trajectory optimization using approximate inference*, Proceedings of the 26th Annual International Conference on Machine Learning (New York, NY, USA), ICML '09, ACM, 2009, pp. 1049–1056.

[Vit67] A. Viterbi, *Error bounds for convolutional codes and an asymptotically optimum decoding algorithm*, Information Theory, IEEE Transactions on **13** (1967), no. 2, 260–269.

[WS02] Laurenz Wiskott and Terrence J. Sejnowski, *Slow feature analysis: unsupervised learning of invariances.*, Neural computation **14** (2002), no. 4, 715–770.

[YFW03] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss, *Understanding belief propagation and its generalizations*, pp. 239–269, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

Curriculum vitae

Personal Details

Name	Dennis Göhlsdorf
Date of birth	April 18th, 1978
Place of birth	Münster, Germany
Nationality	German

Professional Experience

11/2007 - **Scientific Assistant, Ph.D student**
Institute of Neuroinformatics, ETH Zürich and University of Zürich
Thesis title: Motor Control with Graphical Models

07/2007 - 11/2007 **Scientific Assistant**
Institute of Neuroinformatics, ETH Zürich and University of Zürich
Development of a Java library for three-dimensional Delaunay triangulations that allows dynamic changes and copes with points in arbitrary position (other libraries are non-dynamic or require points to be in general position)

04/2007 - 05/2007 **Software Development**
yWorks GmbH, Tübingen
Implementation of an $O(N)$ planarity test for a commercial library of graph algorithms

2000, 2006 **Teaching Assistant**
University of Tübingen
Lectures in computer science

Education

04/2001 - 02/2007 **University of Tübingen**

Major course: Bioinformatics (Diploma)
Thesis: Planarity testing and planarizations
Diploma (final mark): 1.0, with distinction

08/2001 - 07/2002 **California State University**,*Long Beach*
Two semesters abroad, studying biochemistry

10/1998 - 06/2004 **University of Tübingen**
Major course: Biochemistry (Diploma)
Thesis: Studies on the proteolytic cleavage of MHC class I related molecule MICA
Diploma (final mark): 1.0

08/1988 - 06/1997 **Pascalgymnasium Münster**
enrolled in a bilingual german-french program
final average mark: 1.1)

Alternative Civilian Service

08/1997 - 08/1998 **University Hospital**,*Münster*
Civilian service in a nursing ward

Miscellaneous Activities

01/2003 - 06/2007 **Aikikai Tübingen e. V.**
Honorary trainer for Aikido

1997 **International Chemistry Olympiad**
Participation at the third round in Germany (40 participants)

1996, 1998 **Federal Competition for Computer Science**,*Germany*
Two participations at the final round (30 participants

Languages

German **native**

English **fluent**

French **basic**

Publications

[GGW+04] D. Goehlsdorf, F. Gieseke, T. Weinschenk, B. Dengjel, S. Stevanovic, H.-G. Rammensee, and A. Steinle, *Proteolytic MICA shedding: Molecular characterization of a novel immune escape strategy of tumor cells*, Abstracts of the Joint Annual Meeting of the German and Dutch Society for Immunology, Maastricht, Oct. 20th-23th **4–6** (2004), 292–293.

[GKS07] D. Goehlsdorf, M. Kaufmann, and M. Siebenhaller, *Placing connected components of disconnected graphs*, Asia-Pacific Symposium on Visualisation (APVIS '07), IEEE, 2007, pp. 101–108.

[SGS06] H. Salih, D. Goehlsdorf, and A. Steinle, *Release of MICB molecules by tumor cells: mechanism and soluble MICB in sera of cancer patients*, Human immunology **67** (2006), 188–195.