NTNU
Innovation and Creativity

Dancing Robots

Axel Tidemann

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Pinar Öztürk, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The Master's thesis will investigate and implement
a multiple paired models architecture used for imitation learning in a simulated robot. The focus will be on using multiple paired models as a control architecture to
achieve imitation. A module is either consisting of paired models (i.e. an inverse and forward model).
The inverse and forward models will be implemented using neural networks.

Assignment given: 2006-01-20
Supervisor: Pinar Öztürk, IDI

Dancing Robots

Axel Tidemann
IDI, NTNU
tidemann@stud.ntnu.no

Abstract

This Master's thesis implements a multiple paired models architecture that is used to control a simulated robot. The architecture consists of several modules. Each module holds a paired forward/inverse model. The inverse model takes as input the current and desired state of the system, and outputs motor commands that will achieve the desired state. The forward model takes as input the current state and the motor commands acting on the environment, and outputs the predicted next state. The models are paired, due to the fact that the output of the inverse model is fed into the forward model. A weighting mechanism based on how well the forward model predicts determines how much a module will influence the total motor control. The architecture is a slight tweak of the HAMMER and MOSAIC architectures of Demiris and Wolpert, respectively.

The robot is to imitate dance moves that it sees. Three experiments are done; in the first two the robot imitates another robot, whereas in the third experiment the robot imitates a movement pattern gathered from human data. The pattern was obtained using a Pro Reflex tracking system. After training the multiple paired models architecture, the performance and self-organization of the different modules are analyzed. Shortcomings with the architecture are pointed out along with directions for future work.

The main results of this thesis is that the architecture does not self-organize as intended; instead the architecture finds its own way to separate the input space into different modules. This is also most likely attributed to a problem with the learning of the responsibility predictor of the modules. This problem must be solved for the architecture to work as designed, and is a good starting point for future work.

Preface

This Master's thesis documents the work done by the author from January to June 2006. The completion of the Master's thesis ends the first period of the Forskerskole at IDI, and I will continue working on my PhD after delivering the Master's thesis. I am part of the Self-organizing systems group (SOS) at the Division of Intelligent Systems, IDI, NTNU.

Acknowledgements

I would like to thank my supervisor, Pinar Öztürk, for guiding me and always making me think about what I was going to do and why. She is also a constant source of drive and motivation through the sheer interest she takes in my work, and how she actively participates in determining what I should focus on. We have collaborated closely during the entire period that has lead up to this Master's thesis. I would also like to thank the other students in our group whom I have worked with and discussed various issues with related to this thesis; these are (in no particular order) Rikke Amilde Løvlid, Boye Annfeldt Høverstad, Firas Risnes Barakat and Ole-Marius Moe Helgesen.

In addition, I would like to thank my co-supervisor Ruud van der Weel, who let me use the Pro Reflex tracking system at his lab at Dragvoll. I would also like to thank two of his Master students, Ørjan Sakariassen and Erlend Refseth Pedersen, who helped me set up and calibrate the Pro Reflex system.

A final thanks goes out to Jon Klein for not only writing the breve simulator, but also for helping me with various technical issues, relating to both the simulator itself and how to program it.

Trondheim, 2nd of June 2006.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
  - 1.1.1 Different types of learning . . . 1
  - 1.1.2 Why learning by imitation? . . . 2
- 1.2 Research question . . . 2
- 1.3 Working hypotheses . . . 2
- 1.4 Methodology . . . 3
- 1.5 Reader's guide . . . 4

2 Background **5**
- 2.1 Imitation in developmental psychology . . . 5
- 2.2 Imitation in neuroscience . . . 11
- 2.3 Imitation in computer science . . . 13
  - 2.3.1 The correspondence problem . . . 21
- 2.4 Discussion . . . 22

3 Related work **24**
- 3.1 Jacobs' mixtures of experts . . . 24
- 3.2 The HAMMER architecture . . . 26
- 3.3 The MOSAIC architecture . . . 27
- 3.4 The differences between MOSAIC and HAMMER . . . 28

4 Design **33**
- 4.1 The multiple paired models architecture . . . 33
  - 4.1.1 Why multiple paired models? . . . 34
- 4.2 Flow of data in the system . . . 36
- 4.3 Pseudocode . . . 39
  - 4.3.1 Activation of the architecture . . . 40
  - 4.3.2 Training of the architecture . . . 40

5 Implementation **41**
- 5.1 At the core of the architecture: the neural networks . . . 41
- 5.2 Specification of inputs and outputs . . . 43
  - 5.2.1 The inverse model . . . 43
  - 5.2.2 The forward model . . . 44
  - 5.2.3 The responsibility predictor . . . 44
  - 5.2.4 The likelihood estimator . . . 45
  - 5.2.5 Calculation of λ . . . 46

5.2.6 The feedback controller 46
5.2.7 The plant 47
5.3 What is learned where 47
5.3.1 The inverse model 48
5.3.2 The forward model 48
5.3.3 The responsibility predictor 48
5.4 The learning algorithm 48
5.5 The difference between learning and action generation 49
5.6 Simplifications 50
5.7 Running the system 51
5.8 The breve simulator 53
5.8.1 Tiny Dancer 54
5.9 Gathering movement data with the Pro Reflex system 56

6 Experiments **60**
6.1 Experiment 1 - The two Ks 61
6.1.1 Description 61
6.1.2 Goal of the experiment 61
6.2 Experiment 2 - The cheerleader 62
6.2.1 Description 62
6.2.2 Goal of the experiment 62
6.3 Experiment 3 - YMCA 63
6.3.1 Description 63
6.3.2 Goal of the experiment 63
6.4 The different parameters of the architecture 64
6.4.1 The learning rate δ 64
6.4.2 The gain K for the feedback-error 64
6.4.3 σ in the likelihood function 64
6.4.4 Calculation of λ, re-visited 65
6.4.5 A note on run time for evaluating parameters 65
6.4.6 Stopping criterion 66
6.4.7 Speed of the different breve versions 66
6.5 A note on the iteration and integration stepsizes in breve 67

7 Results **68**
7.1 Description of the plots 68
7.1.1 Performance during the training period 68
7.1.2 Performance of one epoch 70
7.1.3 The performance of the system compared to the desired state 71
7.1.4 The performance of the system with λ values superposed 71
7.1.5 Attractor plots 71
7.2 The two Ks 72
7.2.1 The learning of each of the modules 72
7.2.2 Switching between controlling modules 73
7.2.3 Attractor plots 74
7.2.4 Was the goal met? 74
7.3 The cheerleader 87
7.3.1 The learning of each of the modules 87
7.3.2 Switching between controlling modules 87

7.3.3 Attractor plots 88
7.3.4 Was the goal met? 89
7.4 YMCA 101
7.4.1 The learning of each of the modules 101
7.4.2 Switching between controlling modules 101
7.4.3 Attractor plots 102
7.4.4 Was the goal met? 103

8 Conclusion 117

9 Future work 120

Bibliography 122

Glossary 128

A Attachments 131
A.1 Source code 131
A.2 Videos 131

List of Figures

2.1 The AIM framework. 10
2.2 Gaussier's architecture 15
2.3 Ito and Tani's architecture 17
2.4 The RNNPB, linguistic-behaviour binding 18
2.5 Tani, experimental setup 19
2.6 Cangelosi's robots 20
2.7 Matarić's approach to imitation 22

3.1 The mixtures of experts architecture 25
3.2 HAMMER, passive route 26
3.3 HAMMER, active route 27
3.4 The MOSAIC architecture 29
3.5 The MOSAIC architecture, action production and observation . . 30

4.1 The architecture of my implementation 35
4.2 The system architecture 37
4.3 Training of the system 38
4.4 A closer look at the training of the system 39

5.1 The recurrent neural network used in the architecture 42
5.2 The sigmoid function, plotted on inputs in the range [-5, 5]. . . 43
5.3 The imitator watching the teacher 50
5.4 The Tiny Dancer, aka Elton 55
5.5 The ball joint 55
5.6 The revolute joint 55
5.7 The universal joint 56
5.8 The setup of the Pro Reflex system 57
5.9 Tracking with the Pro Reflex system 58

6.1 The two Ks motion 61
6.2 The cheerleader motion 62
6.3 The YMCA motion 63

7.1 Total training period, the two Ks 75
7.2 Performance at the last epoch, the two Ks 76
7.3 Desired and actual trajectory, the last epoch, the two Ks 77
7.4 Performance at epoch 1000, the two Ks 78
7.5 Desired and actual trajectory, epoch 1000, the two Ks 79
7.6 Performance at epoch 1057, the two Ks 80

7.7 Performance at epoch 1058, the two Ks 81
7.8 Performance at epoch 1100, the two Ks 82
7.9 Performance at epoch 1500, the two Ks 83
7.10 Desired/actual state and λ plots, the last epoch, the two Ks . . . 84
7.11 Attractor plots, module 1, the two Ks 85
7.12 Attractor plots, module 2, the two Ks 86
7.13 Total training period, the cheerleader 90
7.14 Performance at the last epoch, the cheerleader 91
7.15 Desired and actual trajectory, the last epoch, the cheerleader . . 92
7.16 Desired/actual trajectory and λ plots, the last epoch, the cheerleader 93
7.17 Performance at epoch 2000, the cheerleader 94
7.18 Desired/actual trajectory with λ plots, epoch 2000, the cheerleader 95
7.19 Desired/actual trajectory with λ plots, epoch 7200, the cheerleader 96
7.20 Desired/actual trajectory with λ plots, epoch 7400, the cheerleader 97
7.21 Attractor plots, module 1, the cheerleader 98
7.22 Attractor plots, module 2, the cheerleader 99
7.23 Attractor plots, module 3, the cheerleader 100
7.24 Total training period, the YMCA 104
7.25 Performance at the last epoch, the YMCA 105
7.26 Desired/actual trajectory and λ plots, the last epoch, the YMCA 106
7.27 Performance at epoch 2000, the YMCA 107
7.28 Performance at epoch 9000, the YMCA 108
7.29 Desired/actual trajectory and λ plots, epoch 2000, the YMCA . 109
7.30 Desired/actual trajectory and λ plots, epoch 9000,the YMCA . . 110
7.31 Desired and actual trajectory, epoch 2000, the YMCA 111
7.32 Desired and actual trajectory, epoch 9000, the YMCA 112
7.33 Attractor plots, module 1, the YMCA 113
7.34 Attractor plots, module 2, the YMCA 114
7.35 Attractor plots, module 3, the YMCA 115
7.36 Attractor plots, module 4, the YMCA 116

Chapter 1

Introduction

1.1 Motivation

The main goal of this thesis is to use imitation learning to teach a robot how to dance, by recognizing and generating movements. I will be the teacher, and the simulated robot will learn to imitate my moves. The robot has to recognize what move I am doing, and generate the appropriate movements. The robot will have to switch between moves as I switch mine while dancing. This approach is very close to the way humans learn how to dance. When learning how to dance, a human will observe another person doing a specific movement. The imitator will then try to recreate the demonstrated movement. The movement may not be very similar at first, but with training the movement of the imitator becomes closer to that of the demonstrated trajectory. In this Master's thesis, the imitator is a robot, trying to learn the movements demonstrated by imitating them. It has neural networks controlling the joint velocities of its body. The neural networks start out with random weights, but are trained in order to match the demonstrated trajectory more closely.

I have chosen the name “Dancing robots” for the thesis, since it sounds fun and fresh and more accessible to people outside computer science. It could very well be called “Sequential movement in a robot taught by imitation learning”, but the former title is more catchy. Dance is something everyone can relate to, and is not necessarily the first thing that comes to mind when discussing computer science. In addition, dance has a very strong imitative element. The main focus of the thesis is to study and implement an architecture that allows for imitation learning to take place in a simulated situated agent.

1.1.1 Different types of learning

Broadly speaking, there are two types of knowledge: declarative (like Paris is the capital of France) and procedural (how to hit a baseball with a baseball bat). Declarative knowledge is deeply rooted in the good old-fashioned artificial intelligence (abbreviated GOFAI), where rules were stored and inference was made based on the rules. These rules were often hard-coded into the program itself by the designers, leading to brittle systems that were not very noise-tolerant, and did not work well outside their designated domain. Most notably, they often did not learn from experience in their environment, but instead relied on the

designer having thought of all the possible situations the artificial intelligence might encounter.

The failure of GOFAI lead to a new paradigm: an artificial intelligence based on biological principles, namely the neural structures found in the brain. The knowledge is represented in a distributed fashion. This knowledge corresponds more to procedural knowledge, as it is not explicit but represents some type of behaviour in the environment. Learning is divided into two categories: 1) unsupervised learning, where the artificial intelligence creates its own clusters and notions about the input data, and 2) supervised learning, where there is a teacher present, guiding the training of the network in the correct direction. A third variation which resembles supervised learning is reinforcement learning, which does provide some feedback on the performance of an agent, but the feedback is only a scalar (i.e. good or bad), not an error vector that can be used to tune the neural network.

1.1.2 Why learning by imitation?

Learning by imitation is a supervised learning approach, since there clearly is a desired state to achieve. It is a user-friendly approach to making a robot behave the way you want it. Instead of fiddling with joint angles, torque, et cetera you could "program" a robot simply by showing it what to do. It is not hard to fathom how much easier it would be to program robots this way.

Imitation is also important in the development of language. Imagine being able to show a robot a certain movement and making the robot learn a word corresponding to the action. This ties the field of imitation to research in audio/vision comprehension, i.e. it requests a multi-modal approach to be fully integrated in the real world.

I believe imitation will help people use robots, and information technology in general. I see my work as a contribution to bridging the gap between people and machines, i.e. making the human-machine interface easier to understand for computer-illiterate people.

1.2 Research question

In order to summarize the previous sections, I have formulated the research question for the Master's thesis:

Research question. *Is it possible to implement a multiple paired models architecture that will separate the input space to different modules by self-organization, and use the multiple paired models architecture as a framework for imitation learning?*

1.3 Working hypotheses

To bridge the gap between the research question and the experiments that are to be done, some working hypotheses have been formulated:

Hypothesis 1. *The multiple paired models architecture will self-organize the control of different movements to different modules.*

Hypothesis 2. *The use of context information will help the modules self-organize as intended, to such an extent that the λ values will follow the context values in terms of transitions, i.e. when one movement is finished, there will be a different module having the highest λ value.*

Hypothesis 3. *The multiple paired models architecture will discover the relationship between the context information and the discrete movements, so that the responsibility predictor will reflect the context information.*

1.4 Methodology

Working on this thesis has mainly followed the following procedure:

1. Studying the relevant topics in the literature. After reading about the background of imitation learning (chapter 2) and related work done in the field of computer science (chapter 3) the inspiration for my own implementation became clear.

2. Design of the architecture and experiments to verify the working hypothesis. The working hypothesis was formulated along with the design of the architecture.

3. Implementation of the architecture. As the implementation phase began (chapter 5), the design of the experiments (chapter 6) also started. The research question and working hypothesis were formulated to clarify what I wanted to achieve with the experiments. During implementation of the architecture, the architecture itself and the design of the experiments changed. Taking the abstract model down to computer code revealed a lot of details that had not been thought of beforehand. Often this lead to a re-visit of the related work studied earlier, to better understand the work presented. Seeing the work from an implementer's perspective led to new insight into the related work, and to redesign of my own architecture. During this period I collaborated with Firas Risnes Barakat on implementing the humanoid robot in breve (the simulator used, see section 5.8). The architecture itself was implemented in MatLab.

   Neural networks were used at the core of the architecture, to implement the forward/inverse models and the responsibility predictor. Training data for breve was gathered using the Pro Reflex system.

4. Conducting the experiments. As the implementation of the experiments began, the experiments were redesigned and new ones were made. After a period of testing and debugging the architecture and the experiments, a final set of experiments were decided upon (chapter 6).

5. Analyzing the results, chapter 7. This was closely related to both the design and implementation of the experiments. When analyzing, the effect of tweaking the parameters of the architecture became more clear. This is a time-consuming process, since running an experiment can take several hours. This often lead to a re-visit to the design stage of the experiment, as well as the implementation stage. The discovery of subtle programming errors at the analysis stage meant more debugging and going back to the implementation stage.

6. Writing the conclusion. After analyzing the results, the conclusion was written, see chapter 8. In this chapter the accomplishments and failures of the implementation was discussed.

7. Writing ideas for future work, chapter 9. During the entire process (but especially during the later phases), new ideas sprung to mind. These were written down in the future work chapter, since the time constraints of the thesis did not allow the implementation of the ideas.

1.5 Reader's guide

The reader who is mostly interested in what I have implemented in the thesis should jump directly to chapter 4, "Design" and read the rest of the thesis from there. The reader who also wants to gain insight into the background for the implementation and why it was implemented as such, should read at least chapter 3, "Related work" for the essential information for the implementation. Chapter 2 should be read by the reader who is interested in getting a broad background of the field of imitation learning in both developmental psychology, neuroscience and computer science.

For all readers, I strongly suggest recommend reading through the Glossary, so that the terminology will be common for both the reader and the author of the thesis. The thesis will use certain over-loaded concepts (such as *movement* and *motion*), that need to be defined in order to avoid confusion.

Chapter 2

Background

Imitation as human capability has been studied for a long time, in both cognitive psychology and neuroscience. Especially the field of developmental psychology has had a focus on how infants learn by imitating their parents and the people in their surroundings. Whereas imitation was thought of as a rather simple ability (manifested in the phrase “monkey see, monkey do”), it has later been acknowledged as one of the key abilities of human cognition.

During the last decade, neuroscientists have found cortical areas that seem to form a basis for the ability to imitate. This work has inspired computer scientists to implement models of how imitation works in situated and embodied systems. The work done by psychologists, neuroscientists and computer scientists form the basis of inspiration for my abstract model.

This chapter will give an insight into the work done in the field of imitation from different points of view, namely developmental psychology, neuroscience and computer science. The chapter is quite eclectic (but by no means complete); the idea is that it forms the basis upon which chapter 3, “Related work” specializes. Chapter 4, “Design” is again more specialized than chapter 3, like a pyramid.

2.1 Imitation in developmental psychology

Piaget [51] describes imitation as the continuation of *accommodation* (i.e. adjustment or adaptation) of sensory-motor schemas to the external world. The sensory-motor schemas comprise both motor and perception stimuli. The sensory-motor schemas are stored in an individual through *repetition*. *Assimilation* is the process where the schemas are maintained, both through production (i.e. generating a behaviour) and recognition of a behaviour.

Piaget sees intelligence as an equilibrium between assimilation and accommodation, i.e. learning new schemas and making these schemas suit the external world of the individual. Imitation is then the on-going accommodation of the schemas to the external world.

Piaget divides the different levels of imitation in six stages.

The first stage consists of preparing the child through reflexes, initiated by external stimuli. The reflex leads to repetition of sensory-motor experiences, effectively yielding assimilation.

The second stage (1 month) expands the reflex schemas developed in the first stage by adding external elements into circular reactions. The external elements are added as a result of experience. An example of a circular reaction is when the child hears the sound of another child crying, it begins to cry too. The external element (the sound of crying) then starts off a circular reaction (continuation of crying). The circular reaction is thus a process where new objects are directly incorporated into known schemas.

At this stage vocal imitation begins. First, *vocal contagion* appears; the voice of other individuals stimulates the voice of the child. Second, there is mutual imitation when an adult imitates the sound the child is making, at the same time as the child is producing the sound. The child will then reinforce its own sound in order to maintain the imitation. Third, the child will try to imitate a sound it has heard before but never produced on its own.

Visual imitation also becomes apparent at this stage, which can be seen in the head movements of the child. The child will follow the head movements of the experimenter.

Piaget points out that no perceptive behaviour (visual, auditory, etc) is a simple act, instead each behaviour is an assimilating activity, where assimilation of objects also takes place. After this assimilation has taken place, accommodation of *movements of organs* to *movements of external objects* can become possible. In this early stage of imitation, the consciousness of the child cannot distinguish itself from the external world, i.e. the subject and the object are one.

The third stage (6 months) introduces secondary reactions, where new circular reactions appear, enabling the child to manipulate external objects. This comes as a result of coordination of *vision* and *prehension* in the child. The secondary reactions are built on top of the primary circular reactions appearing in the previous stage. Piaget names it *conservative imitation*, since secondary schemas are not coordinated among themselves. In addition, assimilation is still the main driving force of imitation - accommodation of the assimilated schemas to pursue novel behaviour is not taking place.

At this stage, the child learns to imitate movements the child already knows (i.e. assimilated to a motor schema) and is visible to the child when it produces them. The child can not imitate movements not visible to itself, nor movements that are new (i.e. it has never produced the movement by itself).

Pseudo-imitation also appears at this stage. This is a kind of imitation that appears through training. It is different from assimilation and accommodation since the imitative action disappears if it is not being constantly trained. It can be produced by repeating an action the child has spontaneously made. The child will then repeat this action, and if the experimenter also continues to imitate, the imitation goes on. However, after the interaction has ended, the movement will not be imitated if the experimenter repeats this movement at a later point. Since this is not a movement that has first been discovered by the child through the reflex

and stored by the process of assimilation, it will not be recalled unless constantly trained. In other words, the child must first produce an action itself in order to be able to learn it properly, and the first production of an action must be through the reflex.

The fourth stage (8-9 months) is distinguished by the ability to coordinate schemas with respect to each other. This increases the mobility of the child, and also evolves a set of indices. An example of such an index can be a sound that triggers a specific behaviour, where the sound is not directly linked to the behaviour itself. The coordinations lead to acknowledging the object and to comprehending space and causality regarding objects.

It should be noted that the child takes spontaneous interest in movements just because they correspond to movement itself is practising, i.e. the schemas are an end in themselves.

The fourth stage also introduces the beginning of imitation of sounds and movements that are new to the child. Earlier, these inputs had left the child indifferent. This coincides with the general progress of intelligence. However, if the stimuli is too different from the child's experience the child will still be indifferent to them, but new stimuli that are comparable to what the child already knows yields an instant attempt to reproduce what has been perceived. In the case of seeing a new movement or hearing a new sound that are partially known to the child, the existing schemas will be modified to integrate the new element (new element meaning new movement or new sound) into the schemas. Due to the modification of the schemas, new movements and sounds cannot be directly assimilated into motor schemas. It is neither possible to directly accommodate the schemas to the models. The child has to try out new schemas to see whether they will match what it perceived. At this stage, the child can also coordinate schemas to find a combination of schemas that might suit the new movement or sound it perceived, i.e. apply *known means* to new situations.

The fifth stage (one year) marks the beginning of systematic imitation of new sensory-motor schemas. Unlike previous stages, the child now also imitates new movements invisible to it. Accommodation and assimilation is now more differentiated, leading to *tertiary circular reactions* which is more sophisticated than just merely investigation. The child now experiments with schemas to discover new objects. The process is done systematically and on a trial-and-error basis.

The sixth stage (one year and 4 months) is the last stage in the development of imitation, and now the child becomes capable of doing mental combinations of the different schemas, relieving it from the necessity of trying them out in the external world, as was necessary in the earlier stages. In other words, the child then becomes able to imitate new sensory-motor schemas instantly by *interiorising* the accommodation of schemas. Earlier, the child could coordinate the different schemas to make a combination to suit the novel perception, but now the child becomes capable of imitating new and complex movements, doing the accommodation internally without the need for external experimentation. *Deferred imitation* also

appears at this stage. The child's reproduction of the perceived movement or sound can now manifest itself after the perceiving the movement or sound, bringing imitation to the level of representation.[1]

After the acquisition of language, *representative imitation* (i.e. having a mental image of what it is about to imitate) develops spontaneously when the child is from 2 to 7 years old. At this stage the representative imitation is often unconscious since the child is very egocentric. When the child reaches 7 or 8, the representative imitation becomes deliberate (i.e. the child can use it for reasoning), taking a part of intelligence as a whole. At this stage the estimation of the person to be imitated influences the child as well. An authoritative or admired person will be imitated, whereas another child at the same age or younger is less likely to be imitated. When with persons of high esteem, the child is often unaware that it is imitating, since it confuses its own activity or point of view with those of others. This does not happen as easily when with children of the same age or younger. Another important property of representative imitation is that the representation comes before the reproduction. In stages I-V there were no images, but the deferred imitation appearing at the sixth stage implies the existence of an image. Piaget thinks that the image consists of interiorised imitation, thus leading the image to take a life of its own. The imitating child is therefore often unaware of it imitating due to this interiorised imitation.

At the age of 7 or 8 imitation of detail appears. The child also becomes conscious of imitation, i.e. it is able to distinguish external elements from itself. At this age *reflective imitation* emerges, i.e. imitation helps the child accomplish what it wants to do. Reflective imitation is thus wholly controlled by the intelligence. This coincides with an increase in the ability of perceptive activity, such as comparisons, analyses, anticipation etc.

Although Piaget claims imitation of movements invisible to the child does not appear until the fifth stage, Meltzoff and Moore [44] found that infants aged between 12 and 21 days were able to imitate facial gestures of the experimenter. They tested six infants with an average age of 14.3 days, and found all of them to be able to imitate the following movements: lip protrusion, mouth opening, tongue protrusion and sequential finger movement (opening and closing the hand by serially moving the fingers). Meltzoff and Moore suggest that the child can represent the visually and proprioceptively perceived information in a way that allows a mapping to the child's own motor system. In other words, the perception the child has made can be transformed in such a way that it can be matched to its own motor capabilities (what Piaget would call schemas). This is supposedly done by an *active matching process* and mediated by an *abstract representational system*. The ability to use these "intermodal equivalences" as Meltzoff and Moore put it, is an innate ability of humans, and not something that develops in the first months of the child's life. Indeed, in [45] they find that neonates can also imitate head movements, not only facial gestures. Meltzoff and Moore suggest that imitation in newborns is due to a process they call *active intermodal mapping* (AIM). The intermodal mapping relies on a repre-

[1] Piaget gives two meanings to the word "representation": in the broadest sense it is conceptual, in the narrow sense it is a mental or memory image (i.e. symbolic representation).

sentational system capable of uniting perception and production of human acts. In [46] AIM is discussed in more detail. The process of matching perceived behaviour with behaviour produced by the child itself is done constantly by the proprioceptive feedback loop. The loop corrects any discrepancies between the child's motor performance and the target. The detection of equivalences between acts of the self and those external is at the core of the AIM. The detection is possible since perceived and produced human acts are coded in a common framework. The framework is therefore "supramodal" since it can unify codings from different modalities.

Meltzoff and Moore think that organ identification is the first step of imitation in the newborn. This is the process of discovering *what to move* prior to discovering *how to move it*. Note that an organ is suggested to be one of the following: head, brows, jaw, lips, tongue, arms, hands, fingers, trunk, legs and feet. Meltzoff and Moore suggest (but do not state firmly) that organ identification is already present at birth, being developed by evolution.

The child must therefore learn what muscle movements generate different states of organ relations. This is done by *body babbling*, according to Meltzoff and Moore, an analogy to vocal babbling, where correspondence between muscle movements and auditory perception is learned. However, body babbling can already begin in utero. They hypothesize that the organ relations is the driving mechanism behind the mapping of externally perceived movements to those of the child's own. This mechanism can also compare what the infant is seeing to what it is doing, allowing for correction of the organ relations. The ability to correct itself also allows the child to separate the self from the other.

The AIM framework consist of three main parts: 1) the perceptual system, which produces perceptions of the other and the self, 2) the supramodal representational system, where the two organ relations (of the self and the other) are compared, which in turn feeds the 3) action system, responsible for controlling the body of the child. The framework can be seen in figure 2.1.

Meltzoff and Moore suggest that there are four developmental changes in imitation of the child: 1) The first developmental change is learning which organ relations correspond to a more high level behaviour termed as human acts. The human act is more than a simple movement, but a goal-directed organ transformation. 2) The second change is becoming aware of a matching relationship between the child and the target. When the child becomes older, it will test whether it is being imitated or not by making sudden changes in its actions. The point is to see what the adult will do when the child changes behaviour. If the adult changes its behaviour according to the change made by the child, the child will understand that the adult is doing the same as itself.

3) The third change (ca. 1 year of age) occurs when the child becomes more preoccupied with detail of the actual imitation. At this stage the child will also try to feel the unseen part of his own body and at the same time the corresponding body part of the adult. For instance, the child might touch its own tongue and the tongue of the adult, yielding an idea of what their own tongue looks like and how both tongues feels the same. 4) The fourth developmental change (which occurs at about 18 months) is when the child understands the underlying goal of the human act of the adult, even when the adult fails to achieve the goal, i.e. imitation of an inferred act. The child will imitate what the adult tried to do, which is the beginning of understanding the adult's intentions. In short, the developmental changes create higher levels of

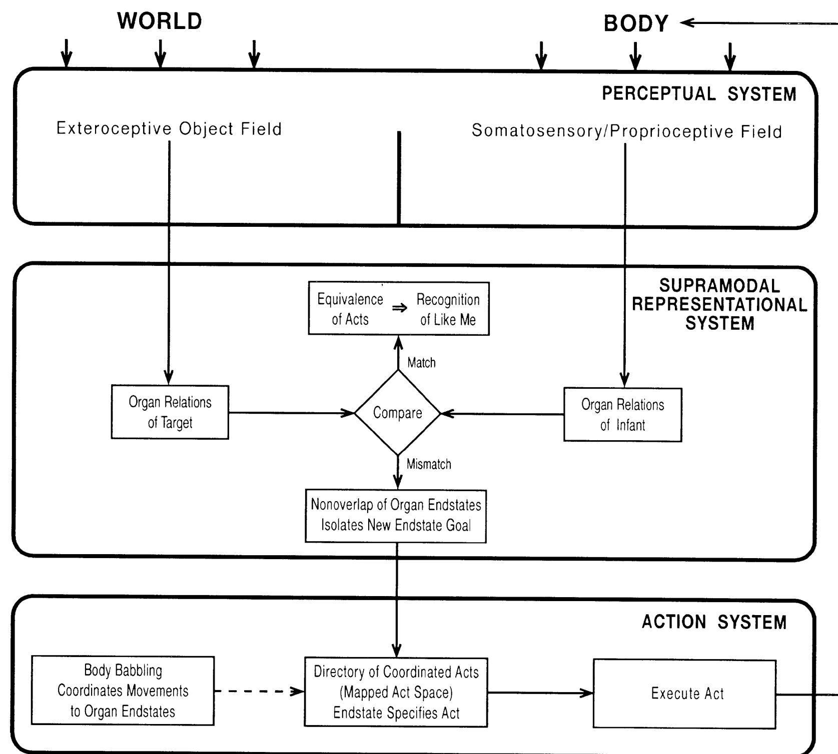


Figure 2.1: The AIM framework, made up of three main parts: 1) the perceptual system, which receives inputs from both the world and the body. 2) The supramodal representational system, which compares the organ relations of the body and that of the world. 3) The action system, which outputs the act chosen in the "Directory of Coordinated Acts". The figure is taken from [46].

abstraction regarding the child's understanding of itself compared to the other.

2.2 Imitation in neuroscience

When mirror neurons were discovered in 1996, it seemed likely that this area of the brain was responsible for imitation. After several years of research in the field, this appears to be agreed upon, and interest in mirror neurons as a basis for imitation has gained a lot of interest in neuroscience and in artificial intelligence.

Rizzolatti [54] discovered mirror neurons in 1996 by placing electrodes in the brain of monkeys. He observed that when the instructor grasped a piece of food, certain neurons in the brain of the monkey became active. These neurons were also active when the monkey performed the same action. It seemed as if the neurons were able to perform some inner simulation of the action, and that this ability is crucial to be able to learn from others. The mirror neurons code motor commands, but not directly, since they are active when both performing and observing an action. The coding must therefore be on a higher level of the action that is observed and reproduced.

The mirror neurons were found in the F5 area of the monkey, which correspond to Broca's area in humans. Soon after the discovery of mirror neurons in monkeys, the same neural behaviour[2] were discovered in humans [53, 24] in the superior temporal sulcus, the inferior parietal lobule and the inferior frontal gyrus (area 45). Area 44 and 45 consist of Broca's area, which plays an important part of the language capabilities of the brain (along with Wernicke's area[3]).

The fact that there is a link between the mirror neuron system and Broca's area makes Arbib hypothesize that the mirror neurons provide the ability to have language [4, 20]. The mirror neurons made human beings capable of imitating each other, forming the basis of language: our brains had to become "language-ready". The human race evolved so that the hands, the larynx and facial expressions could be used to form a language, and the mirror neurons enabled an individual to imitate the sound of another individual, allowing for all sorts of word-games (i.e. determining what a sound means by relating it to specific objects). The word-games would then develop into a language. The use of language must have provided a considerable advantage, since our narrow throats increase the risk of choking on food. It is not difficult to imagine how imitation would be beneficial to any race; if an individual made a discovery it could simply show it to other members of the group to spread the new-found knowledge.

In addition to facilitate learning by imitation, Gallese believes that mirror neurons are crucial to *mind-reading* [21]. Mind-reading is attributing mental states to other human beings, such as goals, beliefs, expectations etc. More

[2]There appears to be no indication in the literature [54, 37, 23] that the mirror neuron system is comprised of a specific kind of neural substrate. The mirror neurons are just part of the neocortex, they have no special physical properties themselves.

[3]Wernicke's area does the processing of auditory input for understanding speech. Broca's area produces the motor commands that result in speech, and is located close to the motor area controlling mouth and tongue movements. There is a bidirectional pathway between Broca's area and Wernicke's area [37].

specifically, Gallese thinks that the mind-reading functions according to *simulation theory*, which claims that the mental states of others are discovered by taking the perspective of the other person, i.e. putting oneself in the shoes of the other. When taking the other's perspective, pretend beliefs, goals, etc. are created, according to how yourself would feel if you were in the same situation. These pretend mental states make up your simulation of the other person, i.e. you are reading his mind by placing yourself in the same situation. Since mirror neurons fire both when observing and performing the same action, they can be seen as providing the transformation that allows the observer to place itself in the shoes of the target, i.e. allowing for both inner simulation of the actions of others and reading the mind of others (which could be argued is the same thing, since the former is subsumed in the latter).

Modeling how imitation learning works is also of great interest, since many fields of artificial intelligence (if not the entire field) would greatly benefit of some way to implement this mechanism. Schaal has discussed several aspects of modeling mirror neurons [55, 56]. Research in the field is more or less divided in two: those focusing on the perceptual part of imitation learning (i.e. transforming visual information into meaningful representations for the agent) or on the motor part (all perceptual information is already present, and is ready to be fed into a perception-action system of the agent). Schaal focuses on the latter, since the former is mostly about geometrical transformation of sensory input, also called the *correspondence problem*, see section 2.3.1. Schaal identifies three approaches to imitation learning [55] (as stated above, none of these approaches deal with the correspondence problem).

The first approach is to directly learn a control policy. Learning it directly means that the imitator must be able to observe both the internal state (i.e. the position of the different joints relative to one another) and the internal action (i.e. motor commands) of the teacher. Normally, these variables are hidden, therefore the movement primitives (i.e. moving the left arm) must be defined in a way that makes them observable (i.e. seeing the acceleration of the fingertip when balancing a pole). The imitator must then know a priori how to convert the observed acceleration of the fingertip into motor commands, i.e. the expressed movements from the teacher must be *understood* by the imitator. The observer does not know the goal of the teacher, it only learns the mapping between state and action, and therefore it is also referred to as *task-level imitation*.

The second approach is to learn a movement from demonstrated trajectories. By recording the movements of the teacher, the observer tries to imitate the behaviour based on the set of recorded coordinates. An example is placing markers on a human arm, and recording the Cartesian coordinates as the arm moves using an optical tracking system. A robot could then try to match the coordinates by approximating the trajectory mathematically. The difference from the first approach is that the demonstrator and imitator shared some common ground (i.e. the imitator would know the motor commands corresponding to the demonstrated movement), whereas in the second approach they do not, and the imitator must *find* a way to generate the motor commands that will result in a match of the demonstrated trajectory.

The third approach is model-based learning. After demonstration, the action is approximated by a predictive forward model. Wolpert has devised such an architecture for imitation learning [63, 64, 62]. Wolpert argues that the cerebellum has several coupled forward and inverse models, one coupling for each

motor primitive. This will be elaborated upon in section 3.3.

Schaal sees the modular approach as the best solution, given how well forward models can predict future states. In addition, Schaal thinks the modular approach fits nicely with the simulation theory of mind reading (as previously discussed in relation with Gallese's work), as well as a possible implementable solution to Meltzoff and Moore's AIM model (see section 2.1).

2.3 Imitation in computer science

It seems as the mirror neurons perform some inner simulation of the action being done (cf. Ziemke's work on inner simulation of perception [65, 34]), and that this ability is crucial to be able to learn from others. Programming behaviours for robots is a tedious task, requiring the engineer to specify the position of each movable part of a robot's body at any time. Programming an artificial arm would require specific control over all limbs, joints, and fingers. If one was able to implement the ability to imitate in robots, programming would be much more easy. Instead of specifying low-level commands, the action could be presented (preferably visually) and the robot would be able to learn it simply by watching. Making robots learn from imitation has been attempted earlier, but recently more bottom-up approaches have been taken.

In the previous section, Wolpert's multiple model architecture was discussed. Demiris and Hayes have also taken a multiple model approach to imitation [14]. They used a biologically plausible computational model (similar to Wolpert's model discussed above) to make a robot learn how to imitate another robot. The architecture is also comprised of paired forward and inverse models, each having the same role as described earlier. The robots were simulated. The learner robot would see the posture of the teacher robot, and would try to imitate it based on a set of postures that it knew (i.e. its *repertoire* of behaviours). Each of these behaviours also predicted how well they suited the demonstrated trajectory it was supposed to imitate. If the prediction error increased for all behaviours during the demonstration (i.e. the output of one of the behaviours did not match that of the demonstrator as the demonstration progressed), the learner was facing a novel behaviour, thus forcing it to learn the new behaviour. So the system became aware when it had to learn a new behaviour, simply by looking at how badly each behaviour actually matched the teacher robot. This architecture will be further elaborated upon in section 3.2.

Breazeal and Scassellati [6] see two fundamental problems with learning by imitation; 1) how does the robot know what to imitate and 2) how does the robot map what it sees to its own motor behaviours to imitate the action (i.e. the correspondence problem)? In order to solve the first problem the robot must have the ability to perceive the movement it is about to imitate. In addition, the robot must have the ability to determine what is important in the flow of sensory input. The second problem consists of converting what the robot saw into motor actions of its own. Breazeal and Scassellati see two approaches: the perception can be converted into motor actions in the observer (like the function of mirror neurons) or to represent the motor actions of the learner and the teacher in the same space (i.e. using Cartesian coordinates) and comparing the trajectory directly. This is similar to the direct learning of a policy approach Schaal described above.

Gaussier et al. [22] propose a neural architecture suitable for on-line learning of sensorimotor actions. The architecture can be seen in figure 2.2. The experiment is to let a robot follow another robot and reproduce the movements of the teacher robot. Both robots are autonomous vehicles. The learner robot has already learned to use its visual input to move in the environment. The learner sees the teacher by the use of a camera. Movement is based on changes in optical flow; if it expands (something is closing up on the robot) it will go backward, if it contracts (the environment is opening up) it will go forward. In addition, the robot will turn if it sees expansion points. This can be seen in the architecture as the "Reflex Action" box. This is hardwired so the robot can follow the teacher. The robot learns to imitate the movements of the teacher by constantly predicting its own actions based on the tracking mechanisms. In the case where the prediction is wrong, a new situation has been encountered. This occurs when the robot turns in order to follow the teacher. The robot then learns to imitate the sequences of turns that the teacher robot performs. The teacher demonstrates a square trajectory, a zig-zag trajectory and a more complex trajectory (more random-like in its pattern). The learner follows the teacher and learns these sequences, so it can reproduce them when the teacher is not present. In other words, the robot learns *turn sequences* from the teacher. The different sequences can be accessed in the robot; it has an internal variable coding for each sequence. The authors call this the "emotion" or the "motivation" of the robot. The infrared sensors of the robot are directly connected to this internal variable, so saturating the infrared sensors directly modifies the variable, and hence the corresponding recall of sequences. This can be seen as a simple interface to the robot. However, this also requires that the robot will not walk into obstacles that might modulate the infrared sensors, leading to a possible change in sequence recall.

Ito and Tani [32] investigated how a robot could imitate multiple patterns (in his work, a pattern is a *sequence*, i.e. multiple patterns are multiple *sequences*) demonstrated by humans. The robot was human-like in shape, having two legs, two arms and a head. The teacher would make certain patterns with his arms, and the robot would imitate these patterns. The robot would go through two phases; first it would learn the hand trajectories of the teacher. This is done by learning to predict sensory information, i.e. the model-based learning described by Schaal earlier. However, note the difference between the work of Ito and Tani on one hand and Wolpert, Demiris and Hayes on the other: In Ito and Tani's architecture there is only *one* model predicting the next state. Wolpert and Demiris had multiple paired models, competing for motor control. So even though this is also model-based learning, it is not done with multiple paired models (as Schaal was describing). The robot had a camera, supplying it with visual input. The robot would then remain still, and just watch the movement pattern of the teacher, and its neural network controller would learn to predict the perception at the next time step. Second, (dubbed the "interaction phase" by the authors) the robot attempted to follow the pattern displayed by the teacher when the teacher demonstrated the pattern. The robot would then try to predict the teacher's movements (i.e. where the arms of the teacher would be in the next timestep), and act accordingly (i.e. imitate). The robot would be familiar to certain patterns that it had been trained on in the first phase. When perceiving a pattern demonstrated by the teacher the dynamic patterns stored in the neural network is regenerated. Seeing a pattern triggers this regeneration.

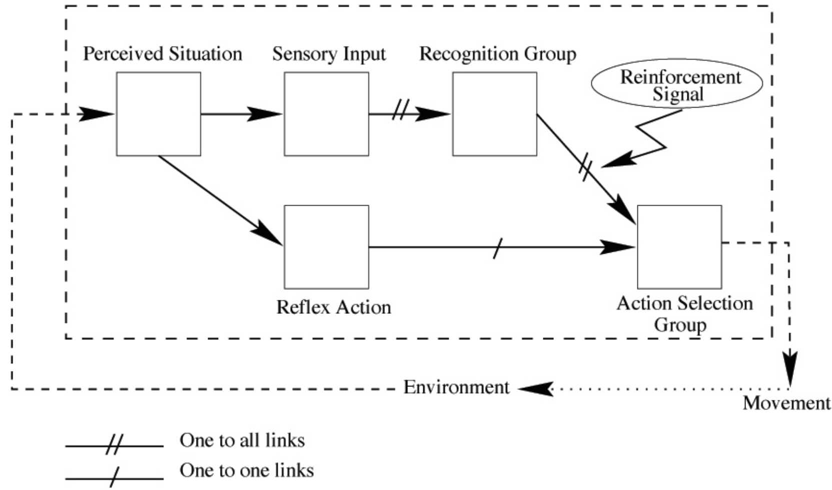


Figure 2.2: The architecture of Gaussier et al, used for on-line learning of sensorimotor actions. The "Reflex Action" box is a hardwired controller, allowing the imitator to follow the teacher. The recognition learns the relationship between sensory input and the action outputted by the "Reflex Action" box, by reinforcement or associative learning. The figure is taken from [22].

The robot is controlled by a recurrent neural network. This network consists of both motor control and mirror neurons. The architecture will be discussed in some detail. The network is augmented with two extra inputs called parametric biases (abbreviated PB, the recurrent network with parametric biases is then abbreviated RNNPB). These units are active both when generating and recognizing a pattern, which makes them the network's mirror neurons. After perceiving the patterns, the network is trained using backpropagation. The parametric biases are then self-determined by the network, i.e. the network decides itself how it will code for each pattern presented to the network. The network can be seen in figure 2.3. The inputs to the network are as follows: r_t is the robot joint angles, u_t is the teacher's hand position, p_t are the parametric biases and c_t are context units that enable memory in the network. The network predicts the next joint angles $\hat{r}_{t+1}$ and the next position of the arms of the demonstrator $\hat{u}_{t+1}$ in the learning phase. Notice how the output of the network are joint angles, *not* motor commands. The multiple model architectures of Demiris and Wolpert have an inverse model that will output the motor commands directly (see chapter 3 for more details regarding the architectures of Demiris and Wolpert). The RNNPB outputs joint angles that must be transformed into motor commands by solving the inverse kinematics of the arms.

In the RNNPB, the error of the predictions is backpropagated through the network. The backpropagation *also* determines the values of the parametric biases; the values of the parametric biases are updated by the error that is backpropagated. Normally, backpropagation only changes the weights of the network, here the error also determines the actual neuron activation of the parametric biases. The parametric bias values will then slowly change towards

stable values, and these values will code for a specific pattern. At the interaction phase there is no further learning to the network, and the parametric biases will then eventually converge to the determined values of the pattern (the values are still calculated based on the error of the prediction, but the synaptic weights of the network are not changed). The parametric biases are active both during learning and during execution of the same behaviour, behaving like mirror neurons.

The robot was trained by actual people performing four different patterns (i.e. four sequences of patterns), with their arms. The patterns were 1) horizontally swinging both arms in phase, 2) vertically swinging both arms in phase, 3) rotating both arms in opposite phase and 4) rotating both arms in phase. The robot managed to follow patterns of the teacher when trained on sets of three of the previously mentioned patterns. Interestingly, the network did not scale well with more patterns, implying that the complexity of the inner dynamics increases non-linearly when more patterns are presented to the learner. But still the robot managed to imitate three of the four patterns presented, when trained only on sets of three patterns. The robot must generalize to a great extent, since human input provide the training patterns, which produces trajectories for the same movement being slightly different from one demonstration to the next. Despite this amount of noise, the robot managed to imitate the learner.

Tani et al. [60] reviews two additional experiments with the RNNPB architecture, in addition to the experiment just discussed. In the experiment discussed above, the parametric biases code for specific patterns, i.e. there is one code for each pattern. This is self-determined during training. When the network is trying to imitate the input from the user, the parametric biases will converge over time to the same activation when a known input pattern is presented to the robot (i.e. a movement that it has been trained on). However, Tani points out that it is also possible to feed the parametric bias activation to the network, and the activations will then generate the corresponding behaviour. In other words, the parametric biases can both recognize and generate behaviour. This function is very true to how mirror neurons operate in the human brain. As mentioned, two additional experiments are reviewed in [60]. In the second experiment (where the first is the one in [32]) a robot arm learns different trajectories and the corresponding parametric biases. By manually switching between these patterns, the network regenerates the different movement patterns.

The third experiment is the most interesting. Tani uses the same network architecture, but two networks (one linguistic network and one behaviour network) are now coupled together via their parametric biases, see figure 2.4. The experimental setup can be seen in figure 2.5. The robot has an arm that it uses to point and hit the different objects. It can push the objects with its body.

During training, a set of sentences are learned along with their corresponding behaviours. The parametric biases in both network the linguistic network and behaviour network are simultaneously updated and constrained to differ as little as possible for each sequence. The sentences consist of two words, a verb and a noun. The verbs are point, push and hit. The nouns are red, blue, green, left, center and right. The robot is then trained to “push red” by manually guiding the robot, showing the sensory input for the desired action as well as providing the word input to the linguistic network. Not all possible combinations of the sentences were trained on the robot, in order to see how well it could generalize. After training, sentences were given to the linguistic network, which

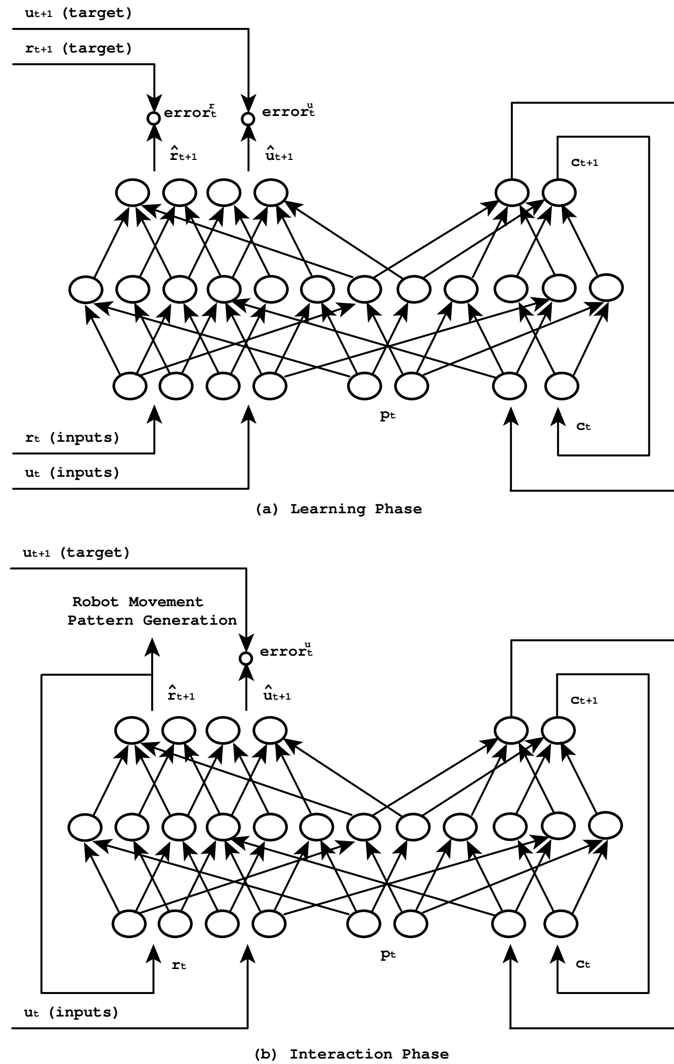
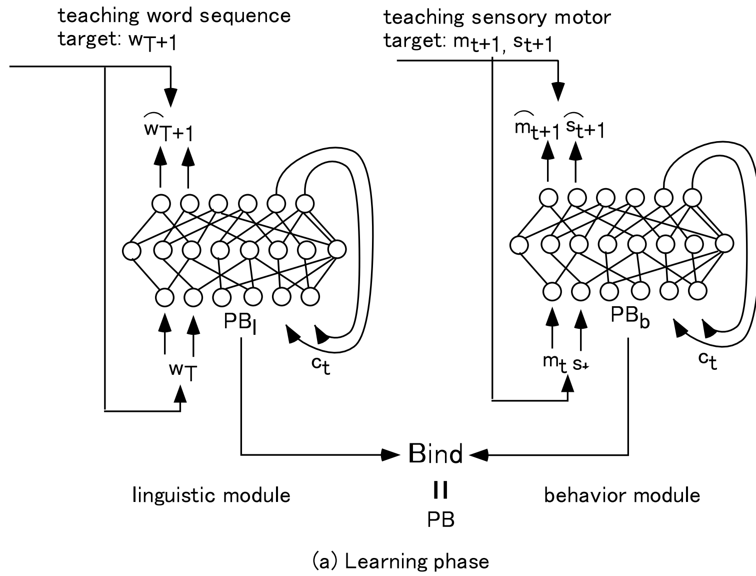
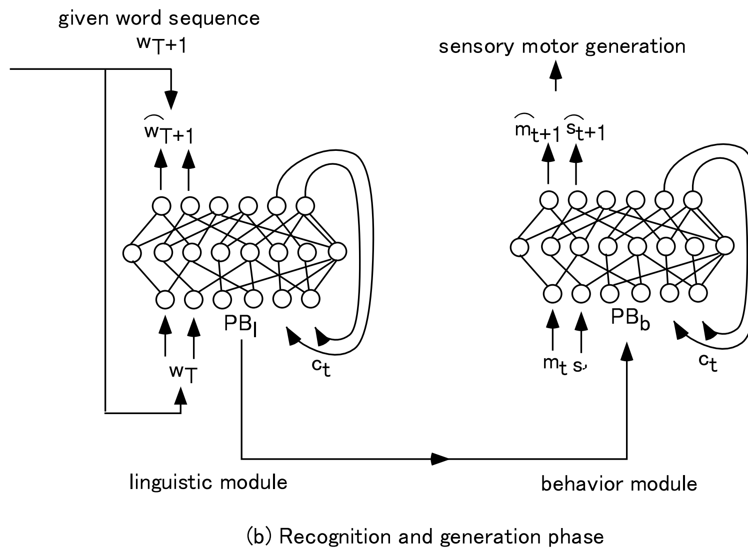


Figure 2.3: The architecture of Ito and Tani, for a) the learning phase and b) the interaction phase. r_t are the robot's joint angles, u_t are the coordinates of the user, p_t is the parametric bias and c_t is the context information. The network trains off-line on the input/output relation. During the interaction phase, the robot joint angles are fed back into the network, to make the network aware of its own actions. The parametric bias is determined through backpropagation of the error, both during learning *and* interaction (but the *weights* are not changed during the interaction phase, only the parametric bias). The figure is taken from [32]



(a) Learning phase



(b) Recognition and generation phase

Figure 2.4: The RNNPB architecture on the linguistic-behaviour binding. In a) word sequences and corresponding behaviours are learned at the same time through the connections of the parametric biases. In b) the network on the left recognizes word sequences and generates the corresponding behaviours via the parametric biases. w_t are the words that the network learns, m_t are the motor values and s_t are the sensor values. Note how the use of the word "module" is taken from [60], and is *not* according to the glossary. The figure is taken from [60].

then initiated the corresponding behaviour via the parametric biases. The robot actually managed to perform the right action for some of the unlearned word sequences. This is possible since each word is not learned separately, but in a relation with other words. The network is therefore able to generalize.

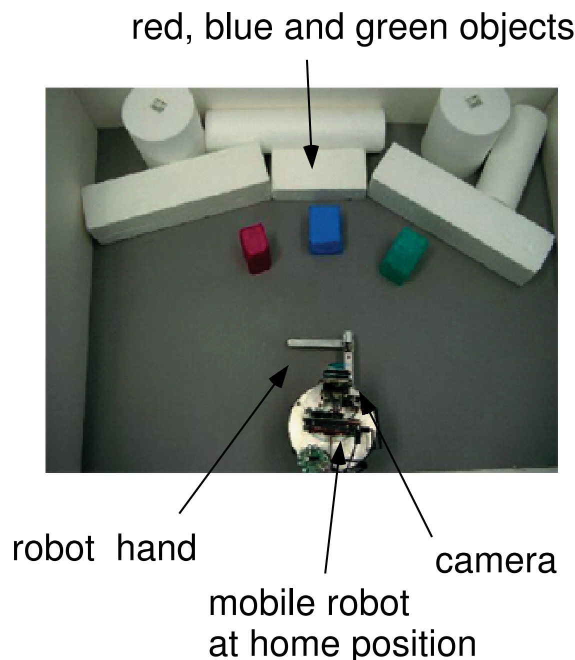


Figure 2.5: The experimental setup in [60]. The robot sees the different objects, and uses its arm to point or hit the objects. The robot is trained on data gathered by manually pushing the robot toward the desired object. The network (as can be seen in figure 2.4) then learns the correlation between sensorimotor experiences and words. The figure is taken from [60].

The experiment is influenced by the work of Arbib [4], as discussed above. The link between motor actions and language is present in the experiment of Tani et al. Language itself is also coordination of motor actions, such as controlling the larynx, lips, tongue and making facial gestures. Here, the robot has a meaningful representation of word sequences. Not only does the experiment make the link between language and motor actions, I see it also as a solution to the symbol grounding problem[4]. Tani et al. have successfully accomplished grounding sentences into meaningful representations of the robot. The robot is

[4] The symbol grounding problems is a term coined by Harnad [25]. It is a response to an argument made by Searle [57]. Searle argues that symbols have absolutely no meaning to a computer. In order to illustrate his point, Searle made an example known as the "Chinese Room" argument. The argument is as follows: suppose a human with no knowledge of the Chinese language was placed in a room. The person is handed questions written in Chinese. The person does not understand Chinese, but has a book of rules, written in English, on how to manipulate Chinese symbols. The person is thus capable of answering the questions by simply following the rules provided by the book. To the Chinese people on the outside of the room, it seems as if the person understands Chinese, when it does not understand anything at all besides following the rules.

The analogy to a computer system is simple. The variables of a computer program only

able to understand the sentences and turn them into action, without the need of explicit symbols. When the robot is told to "push red" it would *know* what it means, and perform the corresponding action. Tani et al. do not mention this as symbol grounding, perhaps since it is such a simple example. But I would not be surprised if Tani publishes more complicated examples of this behaviour and labels it symbol grounding.

Cangelosi has done a lot of work in the area of symbol grounding. More recently, he has done work on grounding by imitation [9]. Cangelosi et al. did experiments where a robot would learn by imitation from another robot [8]. At the same time, the teacher would tell the learner the names of actions and objects (i.e. the verbs and the nouns). The robots were stationary, and had arms that could manipulate objects. The experiment was done in a computer simulation, with real-world physics (such as gravity). See figure 2.6.

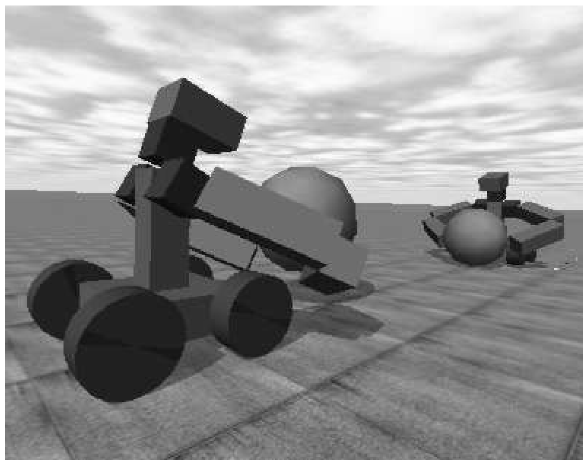


Figure 2.6: The robots in Cangelosi et al.'s experiment on learning by imitation. The learner robot imitates the teacher robot while the teacher communicates the names of actions and objects. Note that the imitator does not *see* the teacher (I will make the same simplification, see section 5.6). The imitator learns the relation between action and objects. The figure is taken from [8].

The learner was already programmed to manipulate objects. The learner has an on-line mimicking algorithm that produces movement dynamics which is fed to a neural network. The neural network then memorizes action patterns that it receives from the teacher robot.

The learning robot would then learn the relationship between words and

make sense to the programmer, the computer only follows a set of rules that it knows it can perform on the variables. The computer program cannot understand anything of the program at all. The symbols themselves are extrinsic to the computer, i.e. they mean nothing to the computer itself.

Harnad proposes a solution to this problem. By grounding symbols in a bottom-up fashion, they will be meaningful to a computer. Grounding symbols means making a link between a symbol and its experience in sensor data. For instance, the symbol of an apple will be grounded with the perceptions of the shape of an apple, its smell, how it tastes and what if feels like to hold. The next time the computer is presented the symbol "apple" it would know what an apple *is*, it would mean more than just a symbol.

their actions by imitating actions while being told the name of the action. The imitating robot would later learn composite actions based on descriptions of simple actions. The composite action "grab" would be presented like "close left arm + close right arm = grab". "grab" would be a new word to the robot, and it had to combine the actions "close left arm" and "close right arm" in order to perform "grab". This is possible since the robot has grounded representations of the actions "close left arm" and "close right arm".

Tani and Cangelosi actually implement the ideas put forth by Arbib, which in turn builds on the discovery by Rizzolatti et al., that the mirror neurons enable imitation learning by representing internal motor actions. Language plays an important part, either in the form of generating internal symbols (such as in the case of Tani) or as explicitly as in the work of Cangelosi. This again goes hand-in-hand with Arbib's idea that the mirror neurons are what enable us to have language. The link between language (i.e. having internal symbols of some sort) and motor action is strong.

Another approach to imitation is taken by Matarić [42, 41]. She has a modular architecture where four important aspects of imitation are considered. The modules are: 1) a mechanism for selective attention, allowing the robot to focus on important features of the visual input (i.e. hands or tools). 2) A mapping system that transforms visual input to meaningful representations for the robot (i.e. transforming coordinates). This system is referred to as the mirror system of the architecture. 3) Motor primitives that can be composed to form more complex motor behaviours. This module resembles (if not completely equal to) Brooks' subsumption architecture[5] [7]. 4) A learning mechanism, based on matches found between observed movements and executable movements. The learning mechanism is biased towards reusing existing motor primitives by composing them hierarchically. The architecture can be seen in figure 2.7. Matarić attacks both problems put forward by Schaal, as her architecture deals with both the visual processing part of imitation as well as the actual motor behaviour.

The modular approach has been tested on several test-beds, including simulated humanoids, a robot dog (a Sony AIBO[6]), mobile robots, and a real humanoid robot, and found to work well. Matarić has mainly a symbolic approach to imitation, but has done connectionist implementations as well [5].

2.3.1 The correspondence problem

The correspondence problem is one of the major problems in the field of imitation learning [49]. It concerns the problem of mapping sensor input from

[5] In Brooks' subsumption architecture, different motor primitives (represented as modules) can be arranged hierarchically in layers. Higher layers represent more abstract actions, whereas low-level layers account for simple actions. The point is that the low-level actions are given higher priority over the higher levels, and can therefore override actions given from the higher levels, since many modules might issue a motor command at the same time. The idea is that low-level actions deal with simple tasks needed to stay alive (such as avoiding obstacles) whereas the higher levels represent more abstract goals. There is no complex computation being done, it is a purely reactive architecture. Several modules can therefore "fire" at the same time (i.e. issue motor commands). The arrangement into layers with different priorities is supposed to deal with the selection of motor commands "on-the-fly" without any computation as well. Brooks' architecture is probably the most successful reactive architecture, however it cannot escape the limitations of its designer. When several layers and modules are present, it becomes harder to make it work.

[6] http://www.sony.net/Products/aibo/

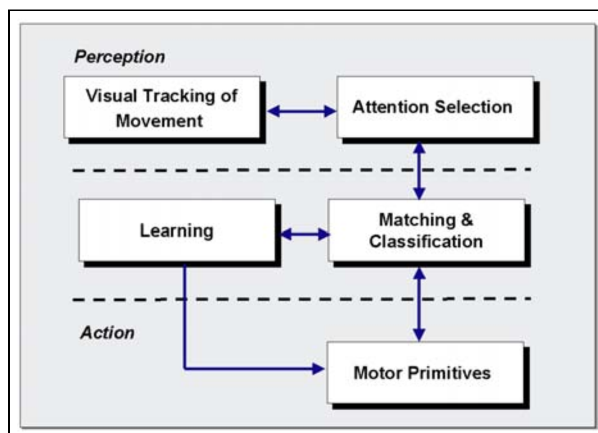


Figure 2.7: Matarić's modular approach to imitation. Note that the use of the word *module* is not according to that of the glossary, instead it simply means "an independent unit". Each of the modules correspond to what Matarić regards important concepts of imitation. The figure is taken from [42].

one frame of coordinates to another. For instance, when the teacher performs a gesture that is to be imitated, it is perceived visually by the imitator. However, the imitator needs to *transform* the information hidden in the visual data onto its own motor capabilities. This is no simple task. Most researchers avoid the problem by making simplifications, such as Demiris' approach where he is able to read the joint angles of the demonstrator [14]. I have taken the same approach in my implementation, see section 5.6.

To have a truly autonomous situated imitative agent in the real world, it needs to have the ability to extract this information from visual input. This will require collaboration with research done in the image processing field.

2.4 Discussion

Comparing Piaget and Meltzoff and Moore it seems like Meltzoff and Moore have the most plausible explanation regarding the development of early imitation in the newborn. They go against Piaget who claims that certain abilities (i.e. being able to imitate acts that include unseen parts of the child's own body) is only evident after about a year or so. Their model incorporates a mechanism allowing for comparison between the organ relations of the other and the self. Since this mechanism also can correct errors in the self, it provides a solution to the dualism problem that Piaget mentions, i.e. that the child is not able to discern itself from the other. Since the child is able to tell that there is an error in its own movements compared to that of the other, it must know what constitutes the self and what is external.

After reading about mirror neurons it seems like they could be the biological equivalent of the matching system that Meltzoff and Moore have in their framework. Although mirror neurons are a relatively new discovery, they are gaining importance and interest from different fields (such as artificial intelligence and

neuroscience).

Recent evidence shows that children with autism have no mirror neuron activity [11]. In an experiment where children with and without autism were told to observe and imitate facial expressions, both groups of children performed equally well at the imitation task, but the children suffering from autism did not have any mirror neuron activity, as opposed to the children without this disorder. The children suffering from autism must rely on some other visual and motor attention mechanism to achieve the same effect (the imitation of the facial gesture), but still they have a hard time understanding the emotions conveyed by the facial expression. This supports Gallese's theory that the mirror neurons make an individual capable of taking the perspective of another. Lacking this ability coincides with the social problems children with autism have, i.e. having problems functioning socially due to lack of ability to understand how other people feel.

In short, imitation learning and mirror neurons are gaining a lot of support of being a crucial component for the development of human cognition. The renowned neurologist V. S. Ramachandran believes mirror neurons will help explain the complex mental abilities of human beings by predicting that "mirror neurons will do for psychology what DNA did for biology" [52].

For me as a computer scientist, there is a lot of work to be done to implement these ideas, as part of the on-going goal to discover true artificial intelligence. An important question is how to implement the different models proposed. I believe a bottom-up approach to intelligence (also known as sub-symbolic artificial intelligence) is the most promising route of one day achieving true AI. There exists an array of techniques that could be used (for a collection of the most common ones, see [47]).

If placing the computer scientists discussed above along a line where connectionist approach would be to the far left and a symbolic approach to the right, Tani, Cangelosi and Gaussier would be to the left, Demiris and Hayes would be somewhere in the middle (since they began using hard-coded forward models to do predictions, but later on tried techniques like bayesian belief networks) and Matarić to the right. This is a very coarse clustering, since both Demiris and Hayes and Matarić have made use of connectionist methods in their body of work, but it separates the works discussed here.

Another coarse clustering would be to look at how imitation is implemented. Either there is one network doing everything, or there are multiple experts approaches. Wolpert, Matarić and Demiris and Hayes belong to the latter, whereas Cangelosi, Tani and Gaussier would be in the former category.

As a closing note, Breazeal and Scassellati puts one type of imitation learning above all others. That is when the student learns both the goal and how to achieve the goal simply from watching. They call this **true imitation**. A system that would be able to perform true imitation would without doubt represent a major breakthrough in the field of artificial intelligence.

Chapter 3

Related work

For this thesis, the focus has been on designing and implementing a multiple paired inverse and forward models architecture (which will be described later on, in chapter 4). Forward and inverse models[1] are concepts from control theory, however neuroscientists who have studied motor control and learning have begun to use the inverse/forward model pairing to explain motor control in the brain. Especially Wolpert uses the inverse/forward pairing to explain how motor control takes place in the cerebellum [64].

This chapter presents work related to the concept of having multiple paired models[2] share the control of an entity. The differences between HAMMER and MOSAIC will also be discussed[3], but first Jacobs' mixtures of expert will be introduced, since it was the first architecture that used several neural networks that would compete for controlling the output.

3.1 Jacobs' mixtures of experts

Jacobs was the first to develop the idea of having a mixture of experts [33] to overcome the storage problems related to neural networks. A network can only hold a certain amount of information, and the division of the input space to specific networks will help the networks specialize on certain computational functions, to avoid interference effects. This is accomplished by using several expert networks and a gating network that decides which network is the expert for a given case of input data. This is an easy task to do if one is aware of the

[1] A forward model takes as input the state of the environment and the action being performed, and predicts the next state of the environment. An inverse model takes as input a goal that is to be achieved, the state of the environment and predicts which motor actions must be made to achieve the desired goal [36].

[2] Note that throughout this thesis I will use "multiple paired models" to mean an architecture that has several *modules*. Each module consists of a forward and inverse *model*. A module therefore corresponds to a box holding both an inverse and forward model. Multiple paired models is the architecture that comprises several modules (i.e. it could have been called *multiple modules*, but I do not want to add any further confusion by deviating from the term already used in the literature).

[3] Jacobs' mixtures of experts is not discussed along with HAMMER and MOSAIC, since it is not an architecture used for imitation, and it does not have the inverse/forward model coupling. The mixtures of experts architecture is mentioned in this chapter for historical reasons since it was the first architecture that used several neural networks and had a soft limit gating mechanism to govern the output of the architecture.

division of subtasks before training the individual networks, however Jacobs' mixture of experts learns to allocate experts to regions of the input space. The experts compete over having control of the final output, and it is the gating network that assigns probabilities to each of the networks. The architecture can be seen in figure 3.1.

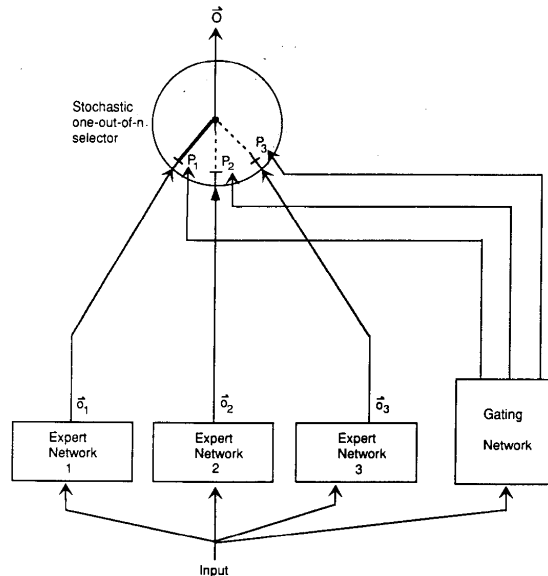


Figure 3.1: The mixtures of experts architecture. Each network competes for control over the output nodes. The gating network determines which expert will govern the output nodes. A soft limit approach is used, allowing for the output of the experts to overlap, as opposed to a winner-take-all approach, where only one expert would be chosen. The figure is taken from [33].

The idea is that the architecture will self-organize into dividing the input space to the different experts. When the input space has many clusters, the idea is that an expert will focus on one of the clusters. In addition, since a soft limit approach is taken, the experts can overlap and the correct output for a given input can be shared among the experts. The mixture of experts was used to discriminate vowels recorded from different speakers. The different experts would then specialize on different classes of vowels to discriminate them from each other.

The architecture was also expanded to include hierarchies [35], where the nodes of the tree constitute two experts and a gating network. The root node is thus the last gating network, determining what to output from the previous gating networks.

Even though his architecture was not used for imitation specifically, nor did he use an inverse/forward coupling, he was the first to propose an architecture where multiple neural networks split the input space amongst themselves to facilitate the learning of each network.

3.2 The HAMMER architecture

The hierarchical attentive multiple models for execution and recognition of actions (HAMMER) [14] is a dual-route architecture. The duality can be found in a passive and an active imitation route. The passive imitation route consists of imitating directly; the posture of the demonstrator is extracted from visual information, and a movement matching module produces the motor commands required to achieve the current posture in the motor system of the imitator. This ongoing process enables the system to learn novel behaviours. The passive imitation route is inspired by the Active Intermodal Mapping hypothesis made by Meltzoff and Moore (see section 2.1). The passive imitation architecture can be seen in figure 3.2.

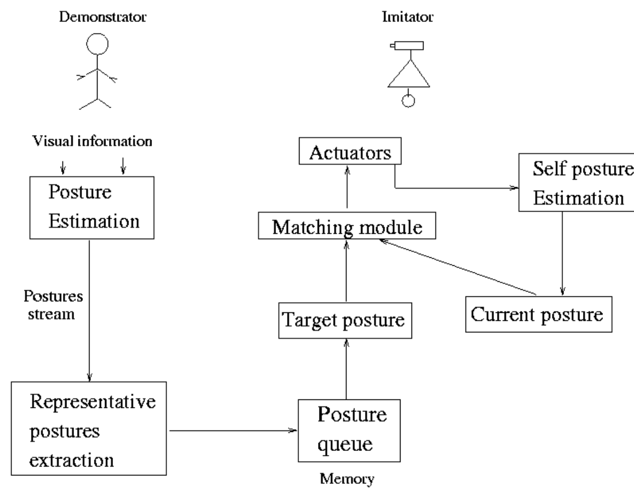


Figure 3.2: The passive imitation route of HAMMER. Based on visual information, the posture is extracted from the demonstrator. The matching module then computes which motor commands are required to achieve the same posture in the imitator as that of the demonstrator. The figure is taken from [14].

The active route consists of multiple paired inverse and forward models, where one pair of inverse/forward models constitute one behaviour. The imitator has a *repertoire* of movements that it knows. The imitative process consists of finding the one behaviour in the repertoire that matches the behaviour of the demonstrator. When a behaviour is observed, each inverse model outputs the motor commands required to achieve the desired position in parallel. Note that the input to the system is the state of the observer. The idea is that the imitator now puts oneself in the shoes of the other, and generates motor commands based on what it would do if it was in the same state. The output of the inverse model is fed to the forward model, which predicts what the next state will be. The predictions of the forward models are compared with the demonstrated trajectory at the next timestep, and when the demonstration is completed, the behaviour with the highest confidence value (i.e. whose predictions did best match the demonstrated trajectory, calculated at each timestep through the demonstration) is chosen as the behaviour corresponding to that of

the imitator.

The active imitation route (see figure 3.3) thus consists of trying out different behaviours in parallel, and selecting the behaviour that best matches the actual demonstrated trajectory.

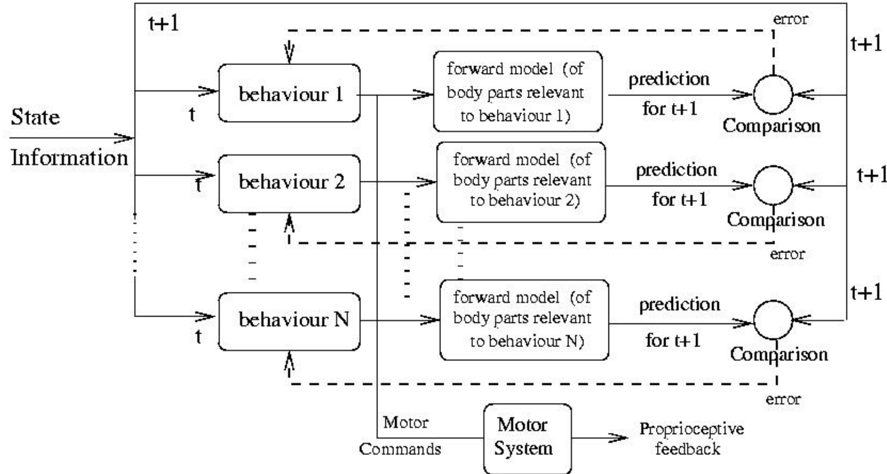


Figure 3.3: The active route of HAMMER. Each behaviour (i.e. inverse model) outputs the motor commands that achieve the desired state. The coupled forward model predicts the outcome of the motor commands on the environment. If the forward model predicts well, the coupled inverse model will be chosen as the behaviour for the demonstrated movement. The figure is taken from [14].

The HAMMER architecture has a lot of extensions and variations[4]. By extending it with an attention mechanism [15], significant computational savings were achieved. The attention mechanism would provide sensory information only to certain modules that perform well, excluding other modules that do not seem to provide the solution to the imitation problem. Since having multiple models leads to the problem of coordinating them, the attention mechanism was invented to ease the computation involved in trying to decide which module was best suited. For instance, if the behaviour to be imitated consisted of picking up a cup, there is no point in considering a module that throws darts. It is computationally expensive to consider the modules from start till end that are not relevant at all for the demonstrated trajectory. The idea of the attention mechanism was thus to filter out on an early stage the behaviours that were clearly not suitable to generate the correct behaviour.

3.3 The MOSAIC architecture

The modular selection and identification for control (MOSAIC) [63] is also made up of multiple paired inverse and forward models. The forward model is fed an efference copy of the total motor command and the current state. The

[4]Note that in [14] the architecture was not referred to as HAMMER, although it is clearly the same architecture. I have chosen to consequently call it HAMMER.

inverse model is fed the current state and the desired state, and outputs a motor command to achieve the desired state. In addition, each pair of inverse and forward models has a responsibility predictor, that can predict the suitability of the module, based on contextual information. Evidence from a PET study is found to support Wolpert's multiple forward and inverse models architecture. Chaminade [10] found that when the goal or the action of a movement was not displayed, different areas of the brain were activated in order to construct either the *means* that achieved the goal or the *actual goal* (depending on which was absent). The MOSAIC architecture can be seen in figure 3.4.

For each module, multiplying the responsibility predictor and prediction error of the forward model gives a *responsibility signal*. The responsibility signals of all the modules are then normalized, and the final motor command output is based on the normalized responsibility signals. The responsibility signal value also determines how much the forward and inverse models will learn; it plays two important roles in MOSAIC: 1) in the decision taking process (then as the *responsibility predictor* and 2) in the learning process (when it has been multiplied with the likelihood of the forward model, then as the *responsibility signal* which is used to gate the learning of the networks). If a forward model makes good predictions, the forward model and the inverse model will be receive more of its error signal for training than a forward model with higher prediction errors.

The calculation of the predictive error can only be done at $t+1$, i.e. at $t = s$ the forward model makes a prediction, and at $t = s + 1$ it is possible to see how well the forward model predicted the next state of the environment. At $t = 0$ there is an evident problem: there is no prediction done at $t = -1$ to compare with. This is where the responsibility predictor comes in: it allows selection of a module based on contextual information *prior* to actual movement. Effectively, at $t = 0$ it is the responsibility predictor alone that determines the suitability of each module.

MOSAIC differs from Jacobs' mixture-of-experts in the way the different networks are coordinated: in Jacobs' architecture, there is one gating network whereas in MOSAIC there are several modules determining the gating of the different modules.

MOSAIC was first an architecture for sensorimotor control and learning [63, 64, 39, 27], but in later works it has been extended to include imitation as well [62]. When dealing with imitation, the coupling of the forward and inverse models are a bit different: instead of feeding the total motor command into each forward model, the output of the inverse model is fed to its paired forward model. The difference between the two can be seen in figure 3.5.

3.4 The differences between MOSAIC and HAMMER

MOSAIC was originally intended as an architecture for sensorimotor control and learning, not imitation, as opposed to HAMMER. MOSAIC is intended to directly model neural activity in the brain, and is therefore implemented using neural networks [27]. HAMMER is also supposed to be a biologically plausible model for neural activity in the brain, however it is implemented using a variety

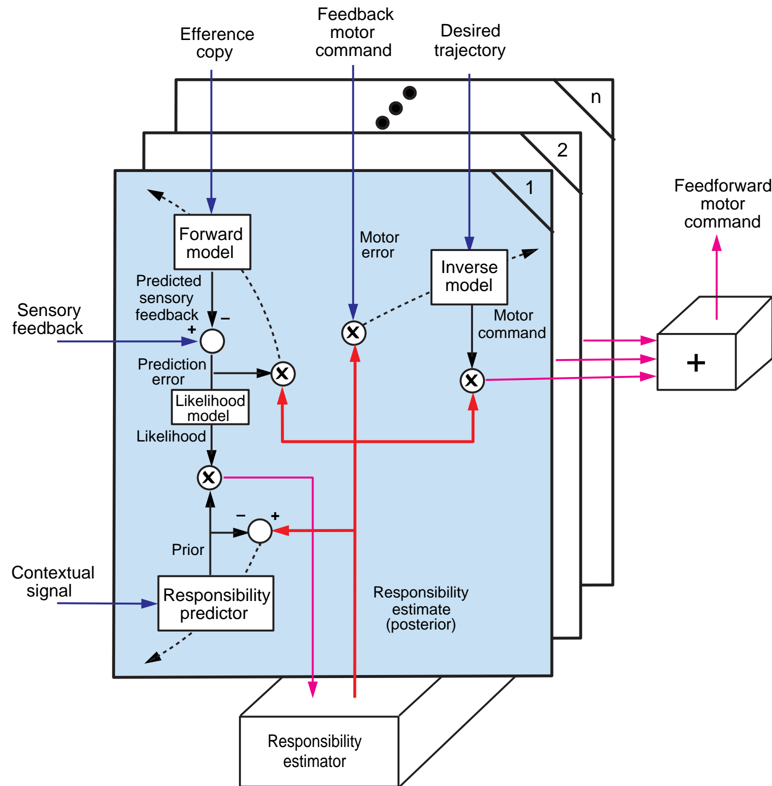


Figure 3.4: The MOSAIC architecture. The inverse model receives the desired state of the environment, and produces the motor command that will achieve the desired state. The feedback motor command is used to train the inverse model. The feedback error command is also added to the final motor output command, although this is not shown on the figure (this is more easily visible in [27]). The feedback error motor command [38] is based on the differences between the desired state at timestep t and the actual state at timestep $t+1$ (i.e. how well the motor commands managed to achieve the target state), multiplied with a gain. Each forward model receives an efference copy of the motor command being executed. Based on the motor command, the forward model predicts the next state of the environment. The prediction is compared to the actual next state of the environment, and a likelihood is computed. The likelihood assumes the prediction error has a gaussian distributed noise term with standard deviation σ. The responsibility predictor receives contextual sensor information, and outputs a prior responsibility that is multiplied with the likelihood. This product yields the responsibility signal, and is normalized across all the modules. It is theoretically possible that all the modules receive the same strength in the responsibility signal, i.e. that all modules are equally suited to control the movement. Notice how the responsibility signal determines how much the inverse model, the forward model and responsibility predictors receive of their error signal. If the responsibility signal is high, the models will receive more of their error signal. The different modules undergo competitive learning, since the good predictors at a certain instant will learn more than the bad predictors at the same time. For more detail on the different parameters of the MOSAIC architecture, see section 6.4. The figure is taken from [64].

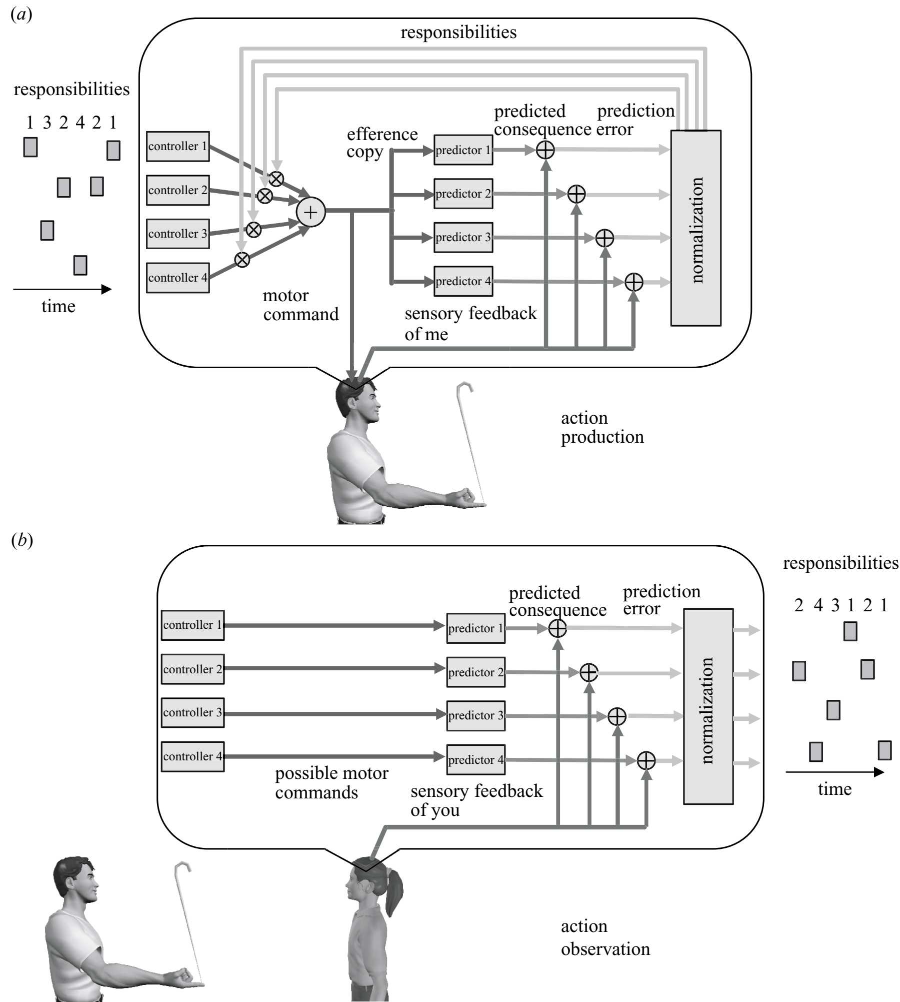


Figure 3.5: In a) MOSAIC is the same as discussed earlier (and seen in figure 3.4. Notice the subtle difference in b), where the output of each inverse model (i.e. controller) is fed to its paired forward model (i.e. predictor). Based on how well these forward models predict, responsibility signals are computed. The sequence of responsibility signals is then used to imitate the behaviour that was observed, by replaying the sequence of responsibility signals. The figure is taken from [62].

of techniques. In [14], the forward models were hand-coded, and the inverse models were PID[5] controllers. Later on, bayesian belief networks[6] were used for learning the inverse/forward coupling [13, 12] and in [58, 16] minimum variance controllers[7] were used as inverse models. Even though both architectures model what goes on in the brain, it is evident that the implementation of MOSAIC is more biologically plausible. Implementations of the HAMMER architecture have been more pragmatic, trying out different solutions that might work.

However, if we do not consider how the architecture is implemented (i.e. what goes on inside each box in the architecture), the differences are not that big. HAMMER has the direct coupling between inverse and forward models (i.e. that the output of the inverse model is fed directly to the forward model, see figure 3.3), whereas in the MOSAIC architecture, the forward model was fed the *sum* of the motor outputs, as can be seen in figure 3.4. This inverse/forward coupling was present in HAMMER from the beginning was added later on in MOSAIC (as I pointed out in the caption of figure 3.5). This is probably related to the fact that the MOSAIC architecture started out as an architecture for motor control, *not* for imitation learning specifically. I think the inverse/forward coupling is a very important part of the architecture for imitation learning, as it best shows how the different modules are responsible for both producing an action and predicting its consequences.

Another difference is the responsibility predictor of the MOSAIC architecture. This gives the modules to have a prior conception of how well suited the module is for the given movement, and that there is an explicit signal that could bias the selection of modules. The HAMMER architecture does not possess such

[5]PID stands for Proportional-Integrative-Controller, which is a way of controlling a system based on an error term [50]. The error term is the difference between the actual state and the desired state of the system. If there is a large error term, the PID controller should output a large control signal to correct the state of the system. The correction is based on three terms: 1) the proportional error, which is simply the difference between the actual state and the desired state, multiplied with a constant, 2) the integrative error, which is the summation of the previous error terms, and 3) the derivative error, which computes the rate of change of the error over time. The idea is that the proportional error will correct any sudden changes of the system, the integral error will account for errors in the past and that the derivative will predict the error term in the future, by looking at the rate of change of the error. These error terms must be tuned in order to yield good performance, which is often not a trivial task.

[6]Bayesian belief networks learn a distribution over a set of variables [47]. The naive Bayes classifier (ibid) assumes that all variables in a dataset are independent of one another. This is a strong assumption, which is why it is called a *naive* classifier. Normally, you have some sort of relationship between different variables. Bayesian belief networks represent the relationship between variables, by stating conditional independence between subsets of variables. The conditional independence assumptions are represented by a directed acyclic graph, where a node corresponds to a variable. It is the arcs of the network that show the conditional independence, a node j is conditionally independent of other nodes that does not descend node j given its immediate predecessors. The Bayesian belief network can also be used for inference; it can compute the probability distribution of any subset of variables knowing the values of another subset of variables.

[7]The minimum variance controller is based on a principle that the central nervous system aims to minimize the noise that results of signal strength [26]. I.e. a strong signal will produce more unstable behaviour, as a result from the noise that is inherent in the central nervous system. The variance of the noise depends on the signal strength, and it increases with the strength of the signal. The desire to minimize the variance of the noise explains the smoothness of arm movement trajectories, displayed in bell-shaped velocity profiles. Sudden and abrupt changes in the trajectory of the arm will require a large control signal, which will be subject to more noise than smooth trajectories. Therefore, smooth trajectories are optimal, and is therefore the preferred solution of the central nervous system.

a method, and must take all the modules into consideration. This is probably why Demiris and Khadhouri developed the attention mechanism for the multiple models, so that the search for the appropriate model could be simplified. Wolpert avoids this problem, by utilizing the context signal.

The inverse/forward coupling of the HAMMER architecture makes it more appealing, since it better shows the modularity of the architecture (as I mentioned previously in this section). Demiris does not state that neural networks should *not* be used as forward and inverse models - quite the contrary in fact, since he has used a variety of methods to implement the different models. The responsibility predictor in MOSAIC seems like a clever way to help coordinate the different modules, and therefore I will combine these two architectures in my own design, as will be described in section 4.1.

Chapter 4

Design

This chapter describes the design of the architecture. The chapter shows the conceptual model, and also how data flows in the system. Pseudocode for the learning and activation of the architecture is also provided.

4.1 The multiple paired models architecture

I have designed a mixture of the HAMMER and MOSAIC architecture. In MOSAIC [62], there are differences between action production and observation: when producing an action, each forward model is fed the sum of all the inverse model motor commands (see figure 3.5). When *observing* an action for imitation, the output of each inverse model is fed into the paired forward model instead. In other words, there is a subtle difference between action observation and generation in MOSAIC.

My approach is similar to the HAMMER architecture and the MOSAIC architecture in observation mode, since each inverse model is fed into the corresponding forward model. However, HAMMER does not learn on-line to the same extent as my implementation; in the HAMMER architecture the models are determined beforehand, either by hand-coding them or learning them by motor babbling[1].

The difference between learning and production of behaviours in my implementation is simply that when learning, the responsibility predictor, inverse model and forward model are adjusted according to the error signal. See section 5.5.

When imitating, motor commands are being *inhibited* (i.e. not sent to the actual motor system) in both HAMMER and MOSAIC during the recognition phase. In the HAMMER architecture, the behaviour with the highest confidence value is selected as the appropriate behaviour after the demonstrated behaviour has finished. In MOSAIC, the recognition phase results in a symbolic string of activated modules. MOSAIC then uses this string of activated modules to play back the action that was observed, see figure 3.5.

[1]Motor babbling is the process where an infant explores its own motoric capabilities by waving its arms, kicking its feet et cetera. At the same time, the infant learns the connection between proprioceptive information and movement of the limbs [46].

In my implementation, motor commands *will be* sent to the motor system, *both* when learning and generating behaviours (motor commands will be described in more detail in section 5.2.1). How is this different from HAMMER and MOSAIC, except for *not* inhibiting the motor command? As mentioned above, the HAMMER architecture does not learn on-line in the self-organizing way I will implement it. For the MOSAIC architecture: in [62], it is not mentioned *how* the MOSAIC architecture when used for imitation is trained (recall that the MOSAIC architecture is slightly different when in action generation mode and action observation mode). The only logical conclusion is therefore that it is trained like when it is in action generation mode, see figure 3.4. The *novelty* in my implementation is therefore that I *train* my architecture with the principles from MOSAIC's *action generation mode* (i.e. as is done in [27]) but use the principles on the architecture as seen in *action observation mode*, something that has not been done by Wolpert.

The resulting behaviour of my implementation can be thought of as the imitation done by a child, where it does not know it is actually imitating when observing an action, corresponding to Piaget's sixth stage of imitation (see section 2.1). Alternatively, it can be thought of as being yet another simulation of one's own motor capabilities, i.e. mentally rehearsing the movement generated by the multiple paired models. However, I personally prefer the analogy to the child imitating directly - it is truly *learning by imitation*, since it tries to recreate the postures of the demonstrator, and match its own postures to that of the demonstrator while watching the demonstrator.

In a sense, I use the "best of both worlds" in my implementation, as discussed in section 3.4. I use the essence of the HAMMER architecture (the pairing of inverse and forward models), along with the responsibility predictor of the MOSAIC architecture, as well as the concept of how the responsibility signal is used to gate the learning of the neural networks.

4.1.1 Why multiple paired models?

The brain is inherently modular. There are different regions that deal with specific functions. However, whether there are multiple paired models within each region (for instance the cerebellum as proposed by Wolpert [64]) is not certain. Nevertheless, the multiple paired inverse/forward approach seems like a very plausible explanation to how the brain works. And it is also easy to implement on a computer.

In addition, the multiple paired models architecture can be seen as a solution to the problem of trying to compress too much information into one network. If a network has learned some concepts, the presentation of new concepts will most likely interfere with the already stored information. This is also called *catastrophic forgetting* [1, 2, 3]. By having the several neural networks that will learn subsets of the total input space, the problem of catastrophic forgetting can be avoided. Ideally, the control architecture would be one massive neural network, that would self-organize into a perfect controller for the robot[2].

[2]Note that I do not contradict the obvious modularity of the brain with this statement. The argument is this: if we knew how the brain learns *and* self-organizes, a massive neural network would theoretically develop the same areas that would have the different functions as found in the brain. I do *not* mean that a single huge feed-forward neural network could model the complexity of the human brain, when I say "massive neural network" I think of a high-

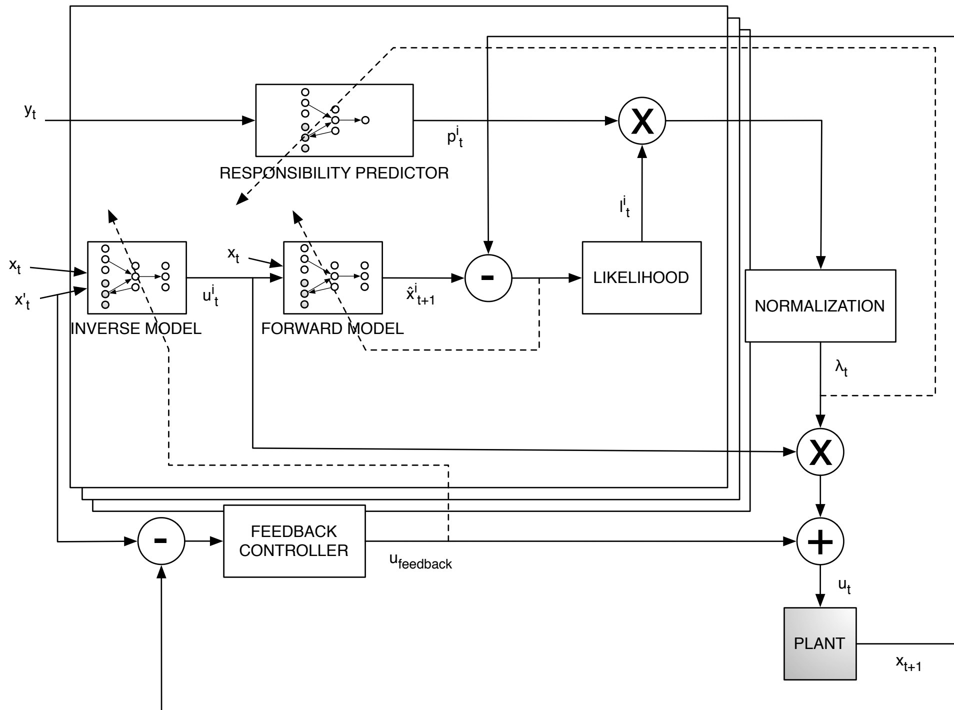


Figure 4.1: The architecture of my implementation. The same architecture is used for both learning and generating a motion. Also note that the responsibility signals are multiplied with the backpropagated errors. The dashed errors show the flow of the error signal through the different neural networks. x_t is the current state, $x_t^{'}$ is the desired state, $\hat{x}_{t+1}^{i}$ is the predicted next state at the ith module, y_t is the context info, u_t^i is the motor command output from the ith module, u_t is the total motor command, u_{feedback} is the feedback error motor command, p_t^i is the prior estimation of the suitability of the ith module (the predicted responsibility), l_t^i is the likelihood and λ_t is a vector containing the responsibility signal for each module (which is also used to gate the learning of the networks). λ_t is normalized. The plant (i.e. the simulator, see section 5.8) is shown in gray, since it is the only *external* entity to the architecture.

However, we still do not understand how the brain works and how it is able to learn, so we are forced to use the current known training algorithms, such as backpropagation. Although it is clearly not biologically plausible (although some evidence has been found to indicate that the known training algorithms are indeed "implemented" in some way in the brain [18]), it is an algorithm that works rather well, with the problem of sometimes getting trapped in local minima.

The goal of the multiple paired models architecture is to exploit the advantages of both a localist and distributed representation, in addition to overcoming the problem of catastrophic forgetting: the localist representation makes it easy to tell where a certain concept is stored, whereas a distributed representation is noise-tolerant and can still function even if some of the nodes of the network become destroyed. By having a multiple paired models architecture with competing neural networks with a softmax gating mechanism this is hopefully achieved. Once the architecture possesses several neural networks, each can be trained on a smaller task, making the correspond to a localist representation. Since each network focuses on a smaller subtask (one could say a network learned a *motor primitive*, such as raising the arm), the output of these networks can be combined to form more complex movements. Each neural network has a distributed representation in itself, making it robust to noise. With the softmax gating mechanism (i.e. allowing more than one module control the robot at a given timestep), the neural networks themselves become part of a distributed representation, since they are not mutually exclusive.

Wolpert argues well for the existence of inverse/forward models in the cerebellum [64], see also section 2.3. Especially the idea of having forward models to compensate for delay in the central nervous system is an appealing idea, since the interaction between the multiple paired models architecture and the simulator will have delays (see section 6.5). The forward models also allows to implement the *predictive* aspect of imitation, i.e. that the imitator predicts what the next move of the demonstrator will be. The predictive capabilities of the brain is without doubt very sophisticated[3].

In addition, the feedback error motor learning scheme developed by Kawato [38] (see section 5.2.6) makes the training of the inverse model feasible in a very simple yet effective manner. The combination of these two factors make the inverse/forward coupling an attractive choice for implementing an architecture for imitation.

4.2 Flow of data in the system

On the highest level of granularity, the flow of data is depicted in figure 4.2. The multiple paired models architecture is implemented in MatLab. breve provides the physical simulator controlled by the multiple paired models architecture (discussed in more detail in section 5.8). MatLab sends motor commands that are to be executed in the simulator. breve then executes the motor commands,

---

order matrix that would allow for networks to develop within the network. It all depends on the level of granularity. From a bird's eye view, the brain is a huge neural network (although extremely complex), but looking more closely, it consists of many connected neural networks.

[3]In fact, the renowned neuroscientists Llinàs thinks prediction is the most important function of the brain [40].

and after execution (i.e. after 13 discrete steps, see section 6.5 page 67 for more information regarding the discrete time steps) returns the current state of the environment to the multiple paired models architecture residing in MatLab. The multiple paired models then computes the next motor commands (see figure 4.2), based on the state information it retrieves from breve.

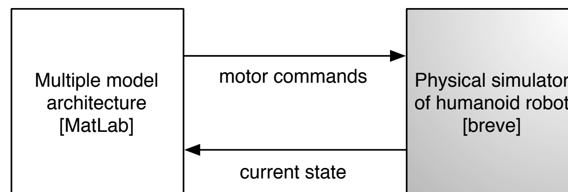


Figure 4.2: The overall architecture of the system. Motor commands are sent from MatLab to breve via sockets, and breve returns the current state of the environment. The gray box corresponds to the plant in figure 4.1.

The training phase of the system is shown in figure 4.3. Each experiment has its own MatLab script file containing the parameters for each experiment. By executing one of the files (for instance `breve_breve_matlab_Ks.m`), a multiple paired models structure will be created and then trained on. The experiment file also contains the target data and the contextual information that is used to train the multiple paired models architecture. The target states and context information is provided by another function, named after the experiment (i.e. `Ks`).

A more detailed view of the training of the multiple paired models structure is shown in figure 4.4. Input to the training function is the target state, context and the multiple paired models that is to be trained, in addition to the state of the environment. These are all sequences of vectors, and the order of the vectors is of course crucial. Since the vectors represent a motion in time, the vectors can not be shuffled, they must be presented in the same order every time.

In figure 4.4, the state of the environment is the state returned from the simulator. When $t = 0$, this is the initial state of the simulator. The state, desired (i.e. target) state and context sensor input are fed to the activation function of the multiple paired models. This is in correspondence with figure 4.1, where the desired state, current state and context information is fed into the different models. Based on the actual state of the environment, the accuracy of the prediction of the previous state can be determined, and the error signal for the forward model can be calculated. This is the *prediction error* of the module, and it plays an important part in deciding how suitable it is for controlling the motor output.

Similarly, the *feedback error* is calculated based on the difference between the desired state at the previous timestep and the current state (see section 5.2.6 for a more detailed description). Based on how well the multiple paired models architecture performed at the last timestep, the error signals are computed along with the responsibility signals. Recall that the responsibility signals λ are used to gate the learning of the different neural networks. This is done by adding the λ value to the training equation for updating the weights of the neural network, as seen in equation (5.6), page 48. Notice how the λ value function as the temperature in simulated annealing - it will effectively determine how much the

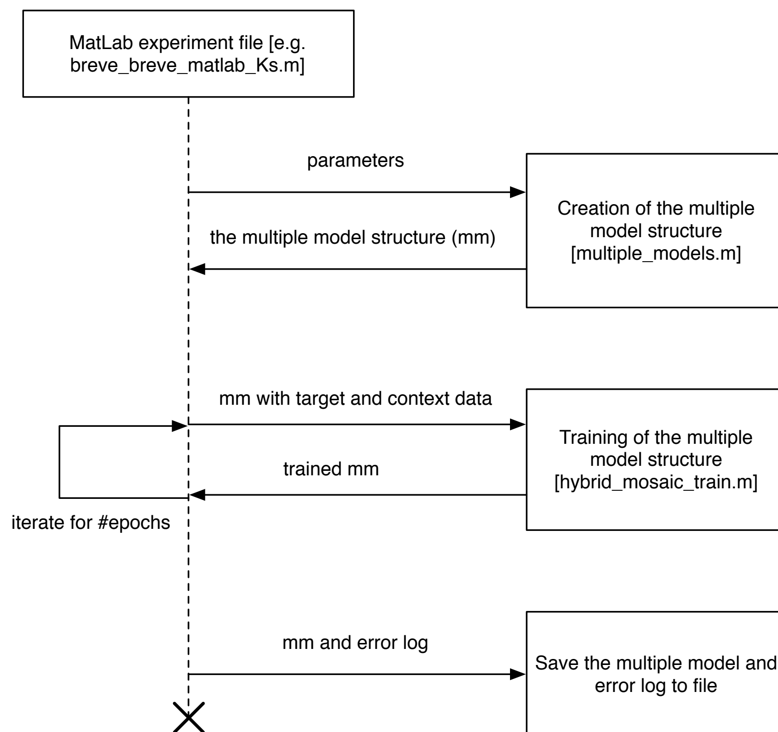


Figure 4.3: The training of the system. By executing the MatLab experiment file, the multiple paired models are created based on the parameters (see chapter 7 for specification of the different parameters) and subsequently trained. After training, the multiple paired models structure and error log is saved to a file.

network learns.

In addition, predictions are made for the state of the next timestep, and motor commands are generated from the inverse models. The motor commands are fed to the simulator, which returns the state of the environment after execution. This will be fed as the current state to the activation function at the next timestep.

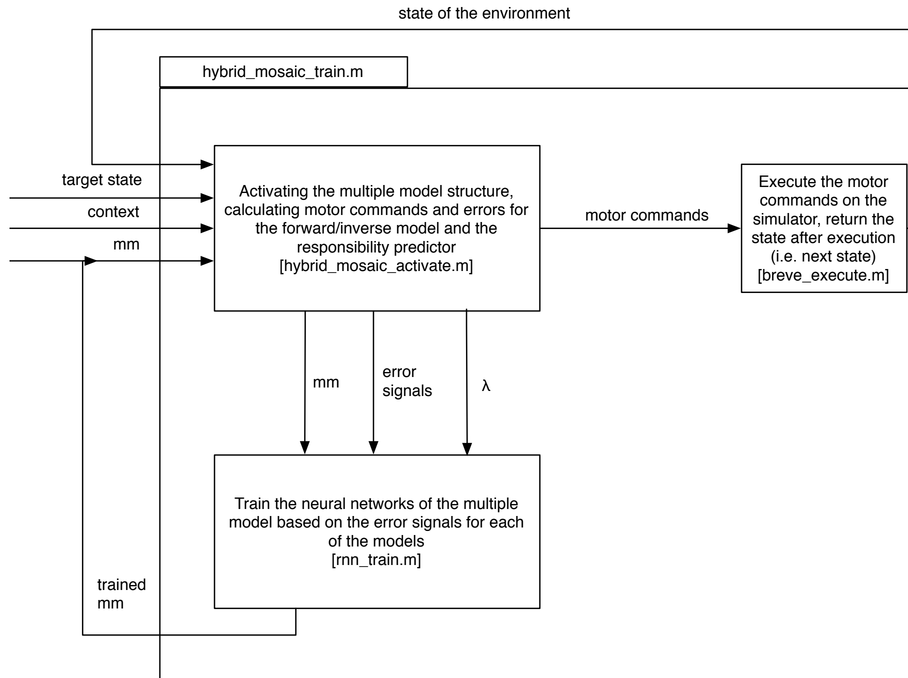


Figure 4.4: The contents of the training box in figure 4.3. Input to the training function is the target state, context info and multiple paired models. *mm* is the multiple paired models structure, λ are the responsibility signals. In addition, the current state is obtained from the simulator after execution of the motor commands. (When $t = 0$ the current state is the initial state of the simulator.) This is input to the activation function of the multiple paired models. The flow of activation can be seen in figure 4.1 (the dashed arrows indicating training of the models in figure 4.1 is what happens in the box directly below the activation box). After training, the trained multiple paired models is either returned if there are no more target states to train after, or it is fed again into the activation function if there are more target states to train on.

4.3 Pseudocode

This section describes in more detail how the architecture will be implemented. The pseudocode is still quite high-level, but should provide a good enough un-

derstanding of how it will be implemented. For further details, see chapter 5 and the attached source code. References to sections where more detailed descriptions can be found are given in parenthesis.

4.3.1 Activation of the architecture

This is how the architecture is activated, i.e. how data flows through the system and produces a motor command. For more detail on the inputs and outputs of each model, see section 5.2.

Algorithm 1 Activate the multiple paired models architecture. Inputs: current state, desired state and context information

```
 1: for all modules do
 2:    activate inverse model with current state, desired state (5.2.1)
 3:    activate forward model with current state, inverse model output (5.2.2)
 4:    activate responsibility predictor with context information (5.2.3)
 5:    compute prediction error (5.2.4)
 6:    compute λ (5.2.5)
 7: end for
 8: normalize λ values
 9: compute the feedback error motor signal u_fb (5.2.6)
10: compute total motor command (5.2.7)
11: return motor command, u_fb, prediction errors, λ
```

4.3.2 Training of the architecture

This describes how the multiple paired models architecture is trained.

Algorithm 2 Training the multiple paired models architecture.

```
 1: gather training data, i.e. desired states and context information
 2: for the number of training epochs do
 3:    clear memory from the modules
 4:    current state = initial state of the simulator
 5:    for all desired states do
 6:       motor commands, u_fb, prediction errors, λ = activate the multiple
          paired models architecture with current state, desired state, context
          information (algorithm 1)
 7:       current state = send the motor commands to the simulator
 8:       train the neural networks with the error signals (5.3)
 9:    end for
10: end for
```

Chapter 5

Implementation

This chapter describes the implementation of the multiple paired models architecture. The system is implemented using a functional programming paradigm, i.e. the system does not change states but instead does operations on data structures and returns new copies of the data structures, without changing the data structure that was passed to the function. This should be kept in mind if reading the source code which accompanies the thesis.

5.1 At the core of the architecture: the neural networks

The brain of the system consists of artificial neural networks that control the simulated robot. Neural networks can be seen as distributed processing units, modeled after how the brain works. The human brain is made up of simple units, called neurons. The neurons are connected via synapses. The synaptic strength determines how much a certain neuron will affect the neuron it is connected to. The connection between two neurons can be either inhibitory or excitatory. The neurons are quite simple processing units, but when many neurons are connected they make up a powerful processing unit. Each neuron would not be much use on its own. Information stored in the network is thus distributed, as opposed to localized. The distributed storage of information makes the network very robust to noise and to loss of neurons. In an artificial neural network, the neurons are often referred to as nodes, and the synaptic strengths are normally called weights.

The forward and inverse model and the responsibility predictor of each module is implemented using simple recurrent networks (also known as Elman networks [19]). A recurrent network has connections between all the hidden nodes, see figure 5.1. This enables the network to have memory, since it remembers the hidden layer activation at the previous timestep. The recurrent neural network was chosen since all the experiments deal with sequences of movements, where the previous state will influence where we will go at the next timestep. Figure 5.1 shows a recurrent neural network with 3 nodes in the input layer, 3 nodes in the hidden layer and 3 nodes in the output layer. The size of the network will later be abbreviated to `input-hidden-output`, e.g. 3-3-3.

The recurrent neural network is instantiated by calling the method `rnn` with

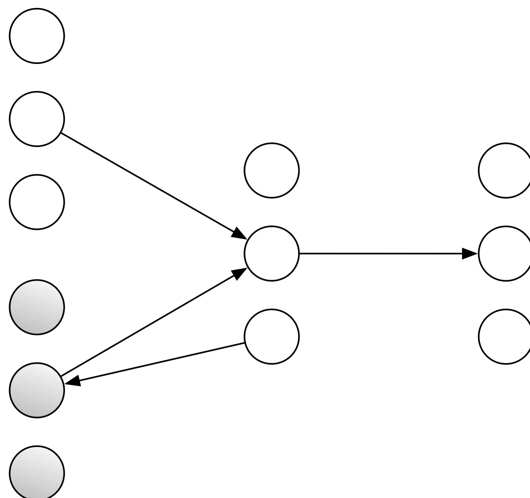


Figure 5.1: The recurrent neural network. The arrows indicate all-to-all connections between the nodes. The grey nodes are the context nodes, i.e. the output of the hidden nodes are transferred to the context nodes (shown by the arrow pointing in the opposite direction) and given as input to the hidden layer at the next timestep. Note that each node has also a *bias* input, not shown on the figure.

the desired size and learning rate, like this: `rnn([ input hidden output ], learning_rate)`[1]. The function returns a network structure holding the weights and activation of the network.

Activation flows in the network from left to right. The activation level at the input nodes is multiplied with the corresponding weight for each node in the hidden layer (along with the context nodes), and is summed up at each hidden node, along with the bias[2]. The sum of the incoming values at every node is squashed through the sigmoid transfer function, see equation (5.1). The sigmoid function bounds the range of the output values of the nodes to [0, 1]. A plot of the sigmoid function can be seen in figure 5.2.

$$sigmoid(x) = \frac{1}{1 - e^{-x}} \tag{5.1}$$

To activate the recurrent neural network, the method `rnn_activate` must be called, along with the network structure and input vector: `rnn_activate(rnn_network, input_vector)`. The method returns the network structure, holding the activation levels for the network. The method cannot return just the output values, as the hidden node values are needed for both backpropagation and as the context input for the next timestep.

[1]Note that for all the functions I have written, it is possible to type `help function-name`, to get a description of the function, as well as the inputs and outputs of the function.

[2]The bias node is an additional input which is always 1. The weight of the bias is also trained by backpropagation.

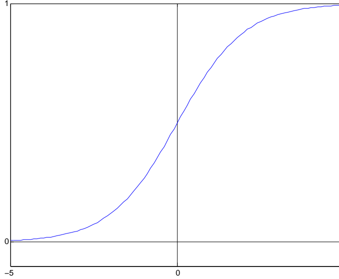


Figure 5.2: The sigmoid function, plotted on inputs in the range [-5, 5].

5.2 Specification of inputs and outputs

The inputs to the multiple paired models structure are the current state, the desired state and the context signal. The output is the motor commands that are to be applied to the simulator. The input/output data flow are shown in figure 4.1. The inputs and outputs of each model will now be specified.

The size of the input/output vectors differs from experiment to experiment, since the degrees of freedom in the simulator differs. The size of the neural networks is listed for each experiment, see for instance section 7.2. The size of the input and output layer of the neural networks equals the size of the input/output vectors for that model.

5.2.1 The inverse model

The inverse model takes as input the current state and desired state and outputs the motor command that needs to be applied to achieve the desired state. The inverse model is often called the *controller* or the *behaviour*, since it is the model responsible for generating motor commands. It is implemented using a recurrent neural network, as described in section 5.1.

Inputs

x_t The current state. It is represented as a vector with values in the range of [0, 1]. Each value represents the state of one of the joint angles of the humanoid robot, e.g. right elbow joint angle, left elbow joint angle and so on. The joint angles are originally in the range of [$-\pi/2$, $\pi/2$], but are scaled to the range [0, 1]. This is normal procedure when formatting input to a neural network.

$x_t^{'}$ The desired (or target) state. It is represented the same way as the current state x_t. The vector shows where the state of the simulator should be at the next timestep.

Outputs

u_t^i The motor commands that need to be applied to achieve the desired state. The output is a vector, where each element is in the range of [0, 1]. The outputs consist of joint angle velocities, with a direct mapping between

the current state x_t and motor commands u_t^i, i.e. if element $x[j]_t$ was the right elbow joint angle, $u[j]_t^i$ would be the right elbow joint angle velocity. The output values are scaled to the range [-1, 1], since the joint angle must be able to move forward and backward. In addition, the output is multiplied with an output gain M (not to be confused with the feedback error gain K), which effectively makes the output range [$-M$, M]. The joint angle velocities are specified as radians per seconds in breve, and they can be more than [-1, 1], thus the need to multiply with the gain.

5.2.2 The forward model

The forward model takes as input the current state of the environment and the motor commands being applied to the environment, and outputs a prediction of what the next state will be *after* the application of the motor commands. It is implemented using a recurrent neural network, as described in section 5.1.

Inputs

x_t The current state of the environment. It is exactly the same as described in 5.2.1.

u_t^i The motor commands outputted from its paired inverse model, as described in 5.2.1.

Outputs

$\hat{x}_{t+1}^i$ The predicted next state, if the paired inverse model will govern the motor output. For each call to the breve simulator (i.e. when the motor commands are sent to the *plant*), the engine iterates 13 times (see section 6.5). The forward model must therefore predict what will happen 13 discrete steps in the future, given the current state and the motor commands that will act on the environment for the next 13 timesteps. The output vector is of the same format as the current state x_t i.e. joint angles of the simulated robot.

5.2.3 The responsibility predictor

The responsibility predictor receives the context signal as its sole input. The output is an *a priori* estimation of how well the module is suited to control the robot. It is implemented using a recurrent neural network, as described in section 5.1.

Inputs

y_t The context signal. It is represented as a vector where the values are either 0 or 1. Only one element of the vector is high at the time. The number of elements in the vector corresponds to the number of modules. Each imitative action consists of sequences of movements. The context signals are always discrete values corresponding to discrete movements, set by me manually for each experiment. Haruno regards "all sensory information that is not an element of dynamical differential equations as contextual

signal" [27] - the discrete context signals I have manually set are certainly *not* elements of dynamical differential equations, and should therefore be valid as context signals.

Say an experiment consists of two movements. The multiple paired models architecture will then have two modules. If the first movement is to raise the left arm, and the second to raise the right arm, the context signal would be [0 1] during all the target states that correspond to the left arm movement, and [1 0] during all the target states of the right arm movement. It can be thought of as someone shouting "Do movement A!", "Do movement B!", or in the case of the YMCA movement (described in section 6.3), a pitch-detector that sends a signal for each of the letters in the dance.

Outputs

p_t^i The *a priori* estimation of the suitability of the module to control the robot. This is a scalar in the range of [0, 1], where a higher value means more suitable. The suitability is based solely on the context signal given to the module.

The following computations are not done by neural networks, but their inputs and outputs are still listed, to facilitate the understanding of the architecture.

5.2.4 The likelihood estimator

The likelihood estimator outputs a value describing how well the module actually predicts the next state. This is calculated by taking the difference between the predicted state and the actual next state, assuming it is influenced by gaussian noise, with a standard deviation of σ.

Inputs

- The difference between the predicted next state and the actual state, i.e. $\hat{x}_{t+1}^i - x_{t+1}$.

Outputs

l_t^i The likelihood that this inverse/forward coupling is actually modeling what happens in the environment, considering gaussian noise influences the models. The output is calculated the following way:

$$l_t = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-|x_t - \hat{x}_t^i|^2/2\sigma^2} \tag{5.2}$$

If the difference between the predicted state and the actual state is close to zero, the output of the likelihood estimator will be high. As the difference between the predicted state and actual next state increases, the likelihood will be smaller.

5.2.5 Calculation of λ

This is shown as the box labeled "Normalization" in figure 4.1. The prior responsibility is multiplied with the likelihood estimation, and the results are normalized over all the modules.

Inputs

$p_t l_t$ The product of the responsibility predictor and the likelihood estimation of each module. The multiplication is indicated by the large "X" circle leading into the "Normalization" box.

Outputs

λ_t The normalized responsibility signals. The output is a vector, holding the responsibility signal for each of the modules. The ith element of the vector is calculated as follows.

$$\lambda_t^i = \frac{p_t^i l_t^i}{\sum_j p_t^j l_t^j} \tag{5.3}$$

This allows for soft-max competition between the modules.

5.2.6 The feedback controller

The feedback controller [38] computes the difference between the target state at time t with the actual state at time $t+1$. It then outputs the motor command needed to move the system towards the desired state. This is the *feedback error* or the *feedback error motor signal*. The difference between the desired state at the previous timestep and the actual state at the current timestep is used as *joint angle velocities*, since the state of the environment corresponds directly to the joint angles of the simulator. The resulting joint angle velocities are multiplied with a constant K. In the beginning of the training, the modules will make bad predictions and generate bad motor commands. The feedback controller will correct the bad movements by coarsely pulling the system in the correct direction. As the training progresses, the modules will become better at controlling the system, and ideally the difference between the desired state and actual next state will become zero, yielding u_{feedback} zero as well.

The feedback error motor signal is also used as the *training signal* of the inverse model[3]

Inputs

- The difference between the target state at the previous timestep and the actual state at the current timestep, i.e. $x_t^{'} - x_{t+1}$. The difference is computed by the large - circle leading into the feedback controller.

[3]Why is this used to train the inverse model? Training an inverse model is a hard problem, since there are many ways to achieve a certain desired state[36]. Imagine if you want to move your left arm from the top of your head to your thigh. There are many ways that you could move your arm and wind up with the arm on your thigh. The feedback error motor signal represents a simple way to find such a trajectory, since it will simply pull the arm in towards the desired state.

Outputs

u_{fb} The feedback error that will move the environment towards the desired state. I implemented a slightly simplified version of the feedback error learning algorithm. In [38] the feedback error was calculated in the following manner:

$$\delta_\tau = K_P(\theta_d - \theta) + K_D(\dot{\theta}_d - \dot{\theta}) + K_A(\ddot{\theta}_d - \ddot{\theta}) \tag{5.4}$$

Where K_P was the proportional gain, K_D was the differential gain and K_A was the acceleration gain. $\dot{\theta}$ and $\ddot{\theta}$ signified the first and second derivative, respectively, and the d subscript indicated the desired state. In [48] $K_P = 60$, $K_D = 1.2$ and K_A is not even mentioned. Seeing how much bigger K_P was regarding to K_D, I decided to make $K_D = K_A = 0$, hence sparing me for the calculation of the first and second derivative of the current state and desired state.

5.2.7 The plant

The plant represents the simulator; the actual world where the motor commands will be applied. It receives motor commands that are to be applied and returns the state of the environment *after* applying the motor commands.

Inputs

u_t The total motor command. The motor command is computed as the sum of each module's motor command output, multiplied with the corresponding λ value. After normalization, the modules which are good at predicting will have high λ values and will influence the final motor command to a greater extent. In addition, the feedback error motor command u_{feedback} is added to the final motor command. The total motor command u_t is calculated as follows:

$$u_t = u_{fb} + \sum_i u_t^i \lambda_t^i \tag{5.5}$$

The multiplication can be seen as the large "X" circle leading into the large "+" circle. The "+" circle represents the adding of the feedback error motor command.

Outputs

x_{t+1} The state of the environment is returned *after* applying the motor commands. As discussed elsewhere (section 6.5), this corresponds to 13 discrete timesteps in the breve engine.

5.3 What is learned where

As shown in figure 4.1, there are three neural networks that are to be trained in the architecture. None of the networks are trained beforehand or hard-coded in any way, they all start out with random weights and are trained based on their error signal. All the error signals are multiplied with the λ value of the

module, and the training happens in all the neural networks at each timestep when training. All the models use the training rule discussed in section 5.4.

5.3.1 The inverse model

The inverse model learns how to produce the motor commands to achieve the target state, given the current state and a target state. The error signal of the inverse model is the feedback motor error command, u_{feedback}, see section 5.2.6.

5.3.2 The forward model

The forward model learns to predict the consequences of a motor command applied in the environment. It takes as input the current state and the motor commands that are to be applied. It outputs a prediction of the next state. The error signal is therefore the difference between its own prediction and the actual next state, i.e. $\hat{x}_{t+1} - x_{t+1}$.

5.3.3 The responsibility predictor

The responsibility predictor learns to predict the suitability of the module given the context information. In order to compare how well the forward model predicted the next state, a comparison must be made of the prediction at the previous timestep with the actual state at the current timestep. However, at $t = 0$, there is no previous comparison of the forward model to be compared with. This is where the responsibility predictor comes in: it predicts the suitability of the module *prior* to movement. The error signal is the λ value of the module[4].

5.4 The learning algorithm

The neural network must be trained to achieve the desired computational function. Deciding which learning algorithm to use is an important decision when using neural networks. The learning algorithm used in the multiple paired models architecture is the back-propagation through time (BPTT) algorithm [61]. BPTT is a supervised learning algorithm, i.e. there is a teacher that guides the neural network towards better performance. For each set of inputs, the teacher know the outputs, and can tell the network how well it performed. The performance error is then back-propagated to the different nodes in the network to adjust the weights. This process is repeated several times, until the performance of the network reaches some criterion, i.e. a low error term. The weights of the neural networks are updated according to the following equation:

$$\Delta w = \delta \frac{dx}{dw} \lambda (x - \hat{x}) \tag{5.6}$$

δ is the learning rate, $\frac{dx}{dw}$ is the derivative of the activation function (the sigmoid function, as shown in equation (5.1)), and $x - \hat{x}$ is the difference between the desired output and the actual output (not to be confused with the state

[4]Since the error signal actually *is* the λ value, it is not multiplied with itself to form the total training signal, as happens with the inverse and forward models.

variables x mentioned earlier, here x means any vector). When λ is high, Δw will be higher, and the network will learn more. When λ is close to zero, Δw will be very small, and the network will not learn as much. All the neural networks start out with random weights. The weights are drawn from a normal distribution with mean 0 and standard deviation 0.2. The weights are trained after each step through the vectors of desired states. If an epoch consists of 100 desired states, the networks are given 100 training passes[5].

Although artificial neural networks are modeled after the brain, the same can not be said for the back-propagation algorithm. Few people believe that there is an error signal traveling backwards through the brain in order to adjust the synaptic strengths. Hence, it is not seen as a very biologically plausible way of training a neural network. However, from an engineering point of view it is a useful method to train a neural network capable of handling noise and being able to generalize. Other training methods exist, such as the Hebbian learning principle[6] or genetic algorithms[7], however back-propagation is quicker and more efficient. A huge effort is being made by leading researchers over the world to discover how the brain is able to self-organize, but until a better training algorithm is discovered the ones already known in the literature will be used. The focus of this master's thesis is not on developing new learning algorithms, but on using already known techniques to solve a problem.

5.5 The difference between learning and action generation

The architecture is used the same way for both learning a behaviour and for generating a behaviour, as described in section 4.1. In both cases, the architecture must receive the desired states and context signals as input, and the multiple paired models structure outputs a motor command to be applied to the environment. The difference is that when learning, the weights of the neural networks are updated after each step through the desired states. When in action generation mode (typically after training to see how well the imitative behaviour was learned), the weights are not updated.

In chapter 7, the plots of the *performance* of the multiple paired models architecture is shown. Notice that all the plots (except for the attractor plots) show the performance *during training*. The logging of the performance is done

[5]Another method often used in neural networks is *batch learning*, where the error of the neural networks is accumulated over an epoch, and the weights adjusted after the epoch. The argument for batch learning is that it represents a more fair way to update the weights. When updating the weights "on-line", the network might be pulled in a certain direction at after training at timestep t, which might actually worsen the performance at timestep $t+1$, i.e. it should not have been changed at timestep t. This method has not been tried out in this Master's thesis, but it is something that could be examined in the future to see if it leads to increased performance

[6]A training algorithm where a weight between two nodes is increased if they are high at the same time, see for instance [43].

[7]An evolutionary approach to finding the weights of a network. The algorithm starts by generating a population of random networks, and subsequently tests each of the network at a given task. The best ones are passed on to the next generation. The best networks constitute the parents of the next generation; the new networks are mixed from that of the parents (crossover) and mutated. This procedure continues until a network with good performance is evolved [30].

during the training of the architecture. To be strict, the performance plots should be logged *without* training, but that would have required to train for one epoch, and run the multiple paired models architecture again to log the performance. This would have required to double the amount of runs, since the logging is done at every epoch, which is not very desirable, since the training of one experiment (the YMCA, see section 6.3) took 11 hours to complete. However, if the networks were trained using *batch learning* the logging of the performance would be correct, since the multiple paired models architecture would then run through the desired states and adjust the weights after one epoch. However, the difference between logging the performance during training and without training should be minimal, since the weights of the neural networks change only by a little for each timestep through an epoch. It is left as future work to try out batch learning.

5.6 Simplifications

The imitator watches the teacher as shown in figure 5.3. However, the simulated robot does not "see" the movement it is to imitate. In fact, it does not have a vision system at all. Instead, it is fed the joint angles it is supposed to imitate. The arrow signifies a transformation of the input data, namely the mapping of visually perceived coordinates to intrinsically meaningful coordinates. The arrow hides the complexity of transforming visual input to motor-coordinates of one's own body. A substantial simplification is made: instead of inputting raw video to the system, the system is fed the joint angles of the limbs of the dancer. This is to avoid the *correspondence problem*, see section 2.3.1.

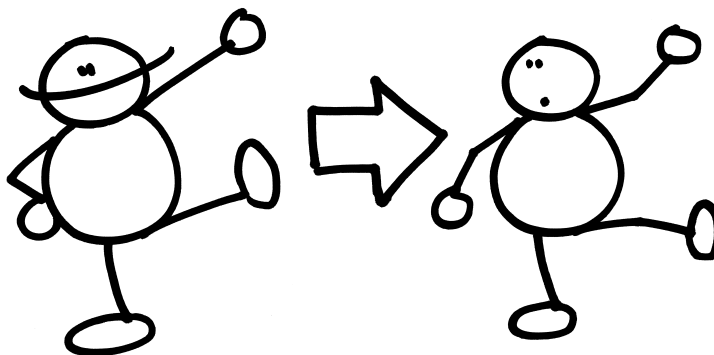


Figure 5.3: The imitator on the right watches the teacher on the left. The arrow hides the correspondence problem, i.e. the mapping from visually perceived coordinates to the imitator's own motor coordinates.

In a crude manner, this corresponds to our ability to perceive movements of other creatures and map the visually perceived information to our own motor capabilities. Experiments done by Desmurget and Prablanc [17] show that humans in fact use estimation of joint angles when imitating. The same approach is taken by Demiris [14] and Cangelosi [8]. The simplification is made because the focus of this master's thesis is on switching between multiple controllers (i.e. inverse models) by using a multiple paired models architecture, not on

transforming visual data to joint velocities, which is a huge task in itself. Cangelosi and Demiris use the same argument. In addition, this corresponds to the "simulation theory of mind", i.e. where one is putting oneself in the shoes of another [21].

The degrees of freedom is limited in each experiment. The humanoid robot implemented in breve has a total of 20 degrees of freedom (see section 5.8.1), but to simplify the learning process fewer degrees of freedom were used than were available. For many of the movements, certain degrees of freedom were not used at all. By limiting the degrees of freedom in each experiment, the training of the robot proceeded faster since the total output space was smaller than when having the full 20 degrees of freedom.

In addition, the humanoid robot is *not* balancing or supporting its own weight. The humanoid is actually hanging in mid-air, see section 5.8.1. To simply make the humanoid stand up is a very hard task that I did not want to deal with in this thesis.

5.7 Running the system

In order to run each of the experiments a specific file must be loaded, both for breve and MatLab. In chapter 7 the files required to run an experiment is listed along with each experiment. First of all, the paths of the system must be set. There is a convenient script file written to do this, called `add_paths`. Just navigate to the root directory of the source code, and run the script by typing

```
> add_paths
```

at the MatLab prompt. A message is printed, informing that the current path must also be added to the MatLab class path. This is because MatLab uses a Java class file to communicate with breve. In order for the class file to load faster, it should be put on the classpath. Type

```
> edit classpath.txt
```

to make sure that the absolute path to the class file is present. Then MatLab should be quit and restarted. If the path was not saved before quitting MatLab, `add_paths` must be run again.

Before starting the simulation, the proper experiment file must be loaded in breve. In order to synchronize breve and MatLab, a small change has been made to one of the breve class files (see section 5.8). Either use the attached breve application (note: it will only run on Mac OS X) or perform the following task: navigate to the `classes` folder of your breve copy (it is preferable to use breve 2.3 IDE). Under Mac OS X it is the following path: `path/to/breve/breve.app/Contents/Resources/classes`. Open the file `Control.tz`, and change the method `+ to iterate` to `+ to manual-iterate`, and save the changes. This makes MatLab able to have full control over breve (again, see section 5.8 for more details) and it is very important that this change to the source code is made, without it the attached .tz files will not run.

After having completed the initial preparations for the system, load the breve source file listed in the experiment list (for instance `Tiny Dancer - K4DOF.tz`), and start the simulator. The breve simulator will then start, and will be waiting

for network connections from MatLab. Note that the breve source file does not do anything by itself, it merely builds the robot and provides an interface to it. The multiple paired models architecture is implemented in MatLab. In order to run the same experiment, simply start the MatLab script file that is specified along with the breve source file. This is done by typing

```
> breve_breve_matlab_Ks
```

at the MatLab command prompt. The experiment will now begin running, and you can watch the robot as it learns how to behave. In order to speed up the simulator, press the pause button (more on this below). After having made it through the entire run (this may take several hours, depending on the number of training epochs) a file is saved holding the multiple paired models structure and an error log of the experiment. The filename is the same for every run, i.e. `Ks.mat`, so if you run the same experiment several times, be sure to rename the file so it will not be overwritten the next time you run an experiment.

It is very important that the breve simulator is run before MatLab, since MatLab tries to connect to the breve simulator to execute motor commands via sockets, and if breve is not running at this time, an error will be produced in MatLab.

If you want to see the result of the training, type

```
> hybrid_mosaic_run(hybrid_mosaic, ...
  hybrid_mosaic(1).desired_state, hybrid_mosaic(1).context)
```

at the MatLab command prompt. `hybrid_mosaic_run`[8] will iterate through an entire epoch with the given multiple paired models structure. The desired state and context constitute the input to the system.

To see some statistics of the results, type:

```
> analyze_this(hybrid_mosaic, hybrid_mosaic_error_log, [])
```

The function `analyze_this` will plot the performance of the entire run, the performance of the last epoch, a plot of the desired state versus the actual state at the last epoch and a plot of the desired state versus the actual state along with λ values (see section 7.1 for more descriptions of the different plots). If you want to analyze a certain epoch, type:

```
> analyze_this_epoch(hybrid_mosaic, hybrid_mosaic_error_log, epoch, [])
```

Where `epoch` is the epoch you want to examine.

In order to see attractor plots, call the following function:

```
> plot_attractors(hybrid_mosaic, hybrid_mosaic_error_log, ...
  @hybrid_mosaic_run, [])
```

Notice that if you replace the empty matrix `[]` with a filename prefix, both `analyze_this`, `analyze_this_epoch` and `plot_attractors` will save PDF files of the figures produced. For more information on what the different plots actually show, see section 7.1.

[8]The name `hybrid_mosaic` is a left-over from the early days of the implementation, when I did not know what to call my implementation. It is left as it is, due to lack of time to change all the names of the source code files.

5.8 The breve simulator

The simulator used in the experiments is the breve simulator[9]. The simulator uses the ODE (Open Dynamics Engine)[10], an open-source physics simulator. breve can visualize simulations using OpenGL. The building blocks of a simulation is written in a language called steve, which is an object-oriented language. breve serves as a front-end to ODE. Programming-wise it is a lot easier to create an object in breve than using ODE directly, since ODE is a C++-library and steve is a high-level object-oriented language.

Writing the mathematical code required for the simulation to run is more easily done in MatLab[11], a programming environment with a huge library of mathematical functions (MatLab is an abbreviation for Matrix Laboratory). It was therefore desirable to use breve in conjunction with MatLab. Our group had not worked with the breve-MatLab combination before, not even the breve simulator was familiar to the group other than by name. The first stage of implementation for this thesis was to try and make MatLab and breve work together.

It is possible to link the breve source code to MatLab by defining an interface written in C, but this turned out to be quite hard to do. Instead, a solution where MatLab communicates with breve via sockets was chosen. In breve it is possible to call functions via web requests to the breve network server. MatLab can then make a request to the breve network server[12] and the reply from breve is read as a string of text. MatLab already had a function *urlread* capable of passing web requests, but this function turned out to have quite an overhead with parsing HTTP headers, and would sometimes fail if the headers were badly formed. Instead, a simple Java class was written that performed the connection to breve via sockets. The Java class had a much lower overhead, and performed a lot better than *urlread*. By doing 100 000 very simple calls to the breve simulator, an average call took 0.0081 seconds. Using the newly implemented Java class, an average call was 0.0024 seconds, more than three times faster[13].

Synchronization was an important issue; since the communication between MatLab and breve was going to be via sockets, some variable delay would be introduced to the communication. Moving joints in breve works by setting joint velocities. For the next iterations of the breve simulator, these joint velocities are applied and the new positions of the joints are calculated. If the motor commands arrive at different times due to network latency, the state of the simulator might be different than what it was in the last epoch of training. This can be seen as a kind of noise due to the neural system, however I wanted to keep it as clean and noise-free as possible in the beginning, and rather add noise at the end. Therefore, the source code of the breve simulator was altered slightly. Firas Risnes Barakat, a student in our group who also was working on the breve simulator, discovered that the main `iterate` method in the Control class was the one calling the `iterate` method in the subclasses. But if the

[9]http://www.spiderland.org/breve/

[10]http://ode.org

[11]http://www.mathworks.com

[12]More specifically, a HTTP GET request where the file name points to the function being called in breve. Arguments to the function can also be passed via the request.

[13]Note: the time spent in socket communication depends on the complexity of the function being called in breve. The results mentioned here only show how much faster the Java implementation is compared to MatLab's *urlread*.

name of the method was changed, it would not be called. I therefore thought out that by calling `iterate` from the class containing the simulator code, it was possible to have total control over the number of iterations of the breve engine between each call from MatLab. This way, MatLab would call breve with the joint velocities and number of iterations, and upon completion of the iterations the state of the simulator was returned to MatLab. In the time between the next motor commands were computed, there was no movement in breve, thus making the breve/MatLab relationship synchronous.

This allowed for a clear separation of the code; the code pertaining to the construction of the simulated object was written in steve whereas the code for the actual simulation (the neural network, the training algorithm, starting and stopping breve, etc.) was written in MatLab. Firas played an important role in working out how to configure breve correctly, and his intimate knowledge of the simulator was of great help to me in the beginning.

5.8.1 Tiny Dancer

The humanoid robot that served as actuator of the motor commands can be seen in figure 5.4. The humanoid robot is called Elton[14]. Elton has a torso, with a head attached on top. The head is actually what makes Elton stand up straight. It might not be easy to see, but he is actually hanging in mid-air. This is because it would be *much* harder to control Elton if the multiple paired models architecture had to deal with making him stand as well. The head is connected to an invisible point, giving the robot the appearance of standing upright. Elton has 20 degrees of freedom, which will now be elaborated upon.

Attached to the torso are the arms and legs. Both the shoulder joints and hip joints are *ball joints*, see figure 5.5. The joint has three degrees of freedom, it can tilt up and down, go side-to-side and twist, also defined X,Y and Z in terms of joint angles in breve. The X-axis is specified along the *norm*[15] of the attachment point of the joint. The Z-axis is defined as a twist from the attachment point. The Y-axis is derived from the other two axes.

The elbow and knee joints are made up of *revolute joints*, see figure 5.6. The revolute joint can move back and forth (defined as the X joint angle in breve), but it cannot go from side to side or twist. It has one degree of freedom.

The ankle joints of Elton are made from *universal joints*, see figure 5.7. A universal joint can move up and down, and from side to side (defined as X and Y joint angle, respectively), but it cannot twist. It has two degrees of freedom.

The wrists and head are fixed to the lower arms and torso, respectively. If they were to be implemented as moving joints, the head would be attached with a ball joint, and the wrists universal joints. The wrist and head joints were not implemented because I did not plan to use them for any imitative movements, however it would require only a few lines of code to implement joints for the wrists and head.

[14]The name "Elton" was given early in the implementation, because I suddenly thought of the song "Tiny Dancer" written by Elton John, a song that was released on the album "Madman Across the Water" in 1971. However, I have refrained from calling the humanoid robot Elton in other parts of the thesis, to avoid confusion for the readers who might not read the specific background for the name, or this implementation subsection. (The album is brilliant, by the way.)

[15]The norm must be specified for each joint when it is attached to another object.

Figure 5.4: The Tiny Dancer, aka Elton. It resides in the breve simulator. The breve simulator communicates with MatLab via sockets, as shown in figure 4.2.

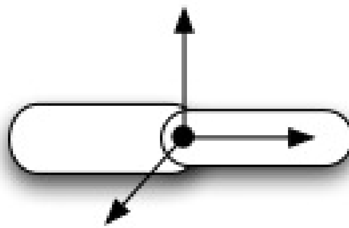


Figure 5.5: The ball joint used to connect the arms and legs to the main body of Elton. The arrow shows the axes of rotation. Picture taken from the breve documentation, http://www.spiderland.org/breve/breve_docs/classes/BallJoint.html

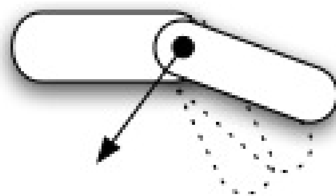


Figure 5.6: The elbow and knee joints are revolute joints. The arrow shows the axis of rotation. Picture taken from the breve documentation, http://www.spiderland.org/breve/breve_docs/classes/RevoluteJoint.html

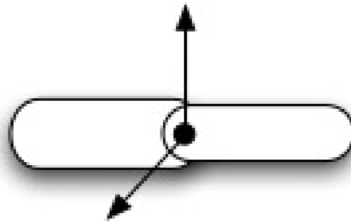


Figure 5.7: The ankle joints of Elton, called universal joints. The arrows show the axes of rotation. Picture taken from the breve documentation, http://www.spiderland.org/breve/breve_docs/classes/UniversalJoint.html

5.9 Gathering movement data with the Pro Reflex system

Data is collected by using the Pro Reflex tracking system at the NevroUtvikling (NU)-lab at the institute of psychology. The system is able to track the position of fluorescent balls within a certain area by using five infrared cameras. The setup of the cameras can be seen in figure 5.8. When the balls are attached to a person, the movement of the balls of the person can be tracked over a period of time, yielding Cartesian coordinates. From these coordinates, the joint angles can be calculated.

The robot is thought to have the same physical properties as that of the dancer, i.e. the joint angles of the dancer correspond directly to the joint angles of the robot. The joint angles of the dancer will therefore correspond to the target state of the robot. The inverse model will then have to produce the motor commands that will lead to the desired state.

14 balls were used to track the movements of my body. A picture of me with the tracking balls can be seen in figure 5.9. The balls were attached using velcro and tape. Quite often a tracking would be ruined due to one of the balls falling off. In addition, some trial and error was required in order to stay within the range of the cameras all the time.

The raw data acquired after the tracking had to be processed manually in order to be useful for the simulator. The tracking of the balls is quite noisy. Sometimes Pro Reflex would miss a ball entirely, creating a gap between the tracked ball (which had been given a specific name) and the “new” ball that would appear afterwards. Since Pro Reflex lost track of the original ball, it thinks the re-emergence *of the same ball* is a new ball, since it has not tracked it before. These gaps needed to be filled manually, in addition to defining the “new” ball as being the same ball that was momentarily lost. In order to produce a trajectory for the entire movement, any discrepancies like the ones mentioned above must be corrected. If for instance the tracking of the shoulder ball would be lost for a second half-way through the movement, the gap between what Pro Reflex knows as the “shoulder ball” and the new unidentified ball (even though they are the same ball) must be corrected, or else the trajectory of the shoulder ball will only show the first half of the movement. This takes quite some time, and requires manual examination of all the data recorded.

After filling the gaps and redefining the balls, the joint angles of the dancing

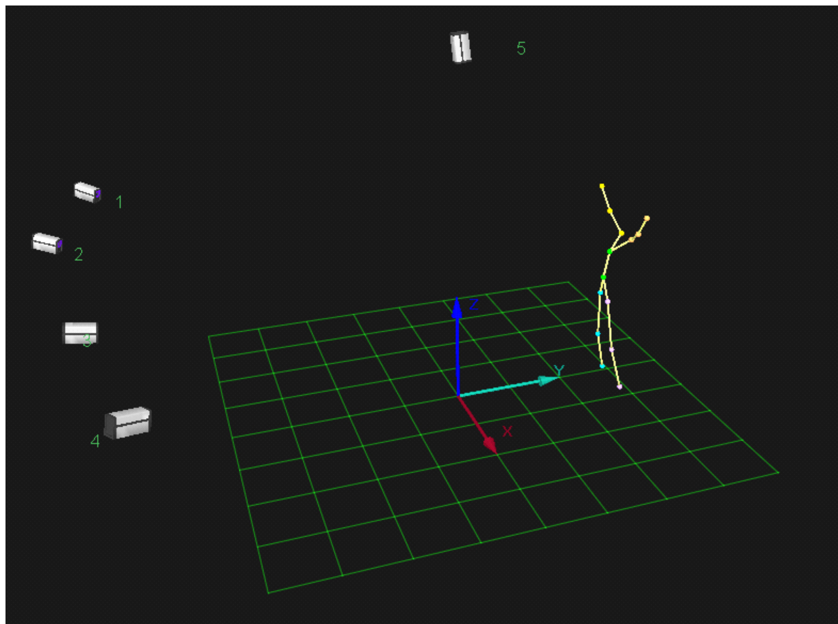


Figure 5.8: The setup of the Pro Reflex system. The five cameras are shown; four in the front of me and one hanging from the roof, along with the stick figure representation of me. To better understand how the stick figure was generated, see figure 5.9 which shows a picture of me with the fluorescent balls used for tracking. The grid defines the area where I can move and still be tracked by the cameras (or more correctly, the *volume* where I can move, since there is a limit on how high the balls can go without losing track of them). The axes show the origo of the system.

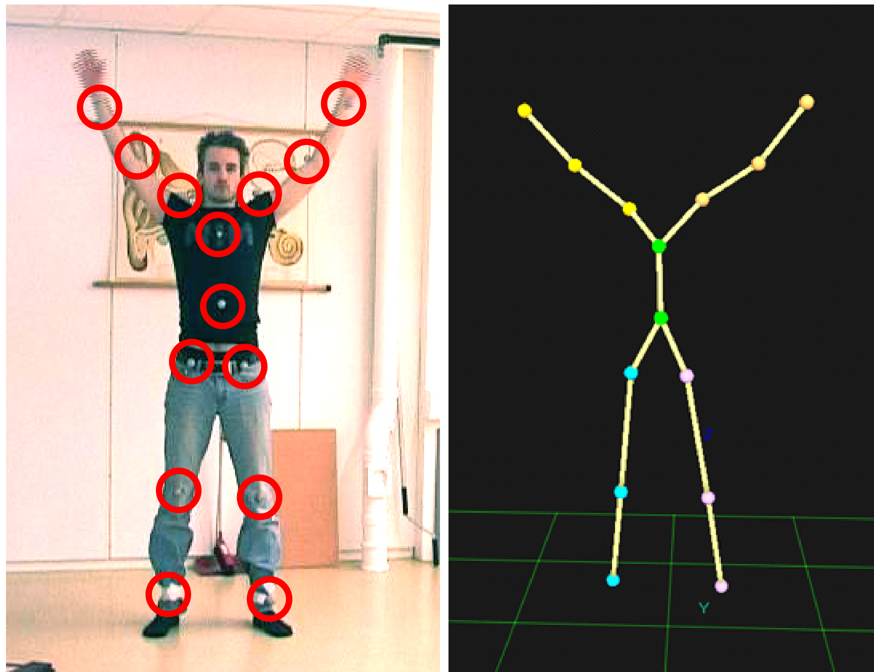


Figure 5.9: The fluorescent balls attached to my body can be seen on the left. The balls are in red circles, since they are a bit tricky to see. On the right is the stick figure representation that results from the tracking of the balls. The balls used for tracking are fairly small and light-weight, and attaches to the body with either velcro or tape. However, the balls attached to my clothes had a tendency of falling off during a movement, which meant that the tracking of the motion needed to be redone.

movements were calculated using an Excel software plug-in called *PCReflex*, which was developed by Innovision Systems[16]. The plug-in was about 10 years old; this was noticeable during use. It was not very user-friendly and consisted of some malicious code that rendered my installation of Excel useless, requiring a re-install of Microsoft Office. I actually spent three days struggling with the software before I understood what I could do with it and what I could not do with it (actually, the process included discovering what I must *avoid* doing in order to not harm the PC I was working on). In spite of the problems, I managed to make good use of PCReflex after a while, and exported the joint angles of the tracking data. PCReflex had a very useful function that allowed the angle between two balls to be plotted against a given plane. The angle between the shoulder and elbow projected in the XZ-plane could then be used as the target for the X joint angle of the shoulder of the humanoid robot. This was repeated for all the angles required for the movement. The data was imported as a text file and loaded into MatLab. Although the data was quite noisy (as can be seen in figure 7.31), no filtering or averaging was done on the data. It was up to the neural networks to do the filtering of the noise. This was a choice made deliberately on my part, to show how well neural networks deal with noise. The sampling frequency of the Pro Reflex system was 200Hz, so the dataset was reduced by taking every 10th sample to match the 20Hz frequency of the breve simulator[17] (see section 6.5 for information on the iteration step size in breve).

[16]http://www.innovision-systems.com

[17]To see exactly how this was done, see the function `YMCA4DOF_desired_state.m` in the source code.

Chapter 6

Experiments

After testing the neural networks to verify that the learning algorithm was correctly implemented, testing with multiple paired models ensued. The experiments are described below, and the results are described in the next chapter. The first two experiments used data collected from breve itself, whereas the last experiment used data gathered from the Pro Reflex system. Each experiment is described, and a goal for each experiment is presented. The pictures shown of each experiment are the *target* states of the experiments, i.e. what the multiple paired models architecture will learn to imitate. The videos found in the attachments to this thesis will give a better understanding of what the different movements look like. Note that for all the experiments, the order of movements is fixed. I have not tried reshuffling the movements during training or during recognition, to see whether the multiple paired models would be able to recognize the movements when presented in an unexpected manner. This is something left for future work.

The goal presented for each experiment is the link to the working hypotheses, i.e. what it sets out to confirm. The working hypotheses are listed in section 1.3.

For the first two experiments, training data was gathered from the breve simulator itself. For the third experiment, human recorded data was used for training (see section 5.9 for more details on how the data was recorded). These training data sets are different from what constitutes more "normal" training sets for neural networks. Normally, a predefined input → output relation is learned. In my case, the input data will *vary over time*. Recall that the input to the inverse and forward model consist of the current state of the environment (see figure 4.1). The current state of the environment changes as motor commands are applied to it, so therefore the inputs to the neural networks in the inverse and forward model changes as well as the training progresses, since the training will ensure better motor commands. Thus, there is no static input/output relationship to be learned, the input data is more dynamic depending on how the multiple paired models architecture performs.

It should be noted that all experiments start out from the same position, see figure 5.4, except for the YMCA experiment, where the arms are a bit closer to the body in the initial position. Attached to the thesis are videos that show both the teacher and the imitator. They have been given names that correspond to the names of the experiments (i.e. *Ks.mov*).

6.1 Experiment 1 - The two Ks

6.1.1 Description

The experiment is called "The two Ks" since the target state was to move the left leg and arm upwards, to form the letter "K". The data needed for training (i.e. desired states) was collected by controlling breve from MatLab; the joint velocities of breve was set from MatLab and the resulting joint angles were collected from breve. The joint angles constitute the motion that make up the "two Ks". The inverse model could easily be trained using the target velocities (since I knew them), i.e. by direct learning[1]. Instead, the corresponding joint velocities were discovered using the feedback error learning approach (see section 5.2.6). The inverse model thus had to discover the joint angles I had already set.

The "K" on the left side and on the right side was regarded as one movement, and therefore two paired inverse/forward models were used in this experiment. The context information was according to the left and right "K". This can be viewed as someone shouting "Left K!", "Right K!". The experiment has four degrees of freedom, see section 7.2 for more details.

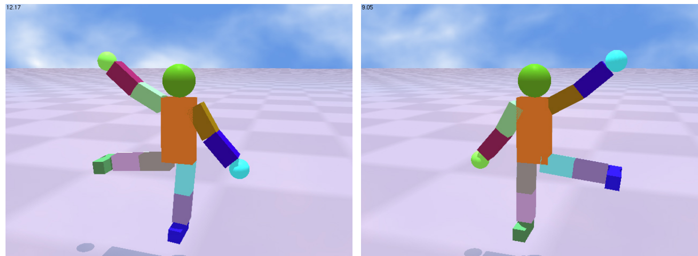


Figure 6.1: The two Ks, one to the right side and one to the left. Imagine that the letter "K" is formed with the left arm/leg and the right arm/leg.

6.1.2 Goal of the experiment

This is the first experiment done with the multiple paired models architecture, and it is therefore quite simple to allow debugging, but still complex enough (i.e. it has two movements) to show that it works.

Hypothesis 1: The experiment has two discrete movements and therefore two modules. The experiment was designed to show that the multiple paired models will self-organize and assign one movement to each module, on a small and not too complex motion (see the Glossary to clarify the meaning of *motion* and *movement*).

Hypothesis 2: The experiment was designed to show that the context information will aid the simple separation of the two movements.

[1]Direct learning of an inverse model is done by applying motor commands to the environment and recording the resulting state, and then reversing the input/output order for training of the inverse model. It is a simple and effective method, but it is not goal-oriented and does not tackle the obvious problem that there may be many motor commands that can lead to the same situation - the inverse model will learn only *one* such mapping by direct learning [36], and it may not be the best one.

Hypothesis 3: The experiment was designed to show that the responsibility predictors of each module will follow the context information.

6.2 Experiment 2 - The cheerleader

6.2.1 Description

The name of the experiment comes from a crude approximation to the movements of a cheerleader. The legs and arms are moved about, to simulate jumping and waving behaviour. There are three different movements, and therefore three different paired inverse/forward models. The joint velocities were again controlled from MatLab, and the target states were collected from breve. The context information was set to match the movements, and can be thought of as someone telling the imitator to do movement 1,2 and 3. The experiment has seven degrees of freedom, see section 7.3 for more details.

Figure 6.2: The cheerleader, each of the three movements. The name was chosen because the movements look a bit like what cheerleaders do during American football matches. However, the movements are not modeled after any specific movement. It is best to see the video (called `cheerleader.mov`) to get a better grip of what the motion looks like.

6.2.2 Goal of the experiment

The experiment is designed to show that the multiple paired models architecture will self-organize in a more complex situation than the first experiment. This experiment has three modules, one for each movement, and more degrees of freedom.

Hypothesis 1: The experiment was designed to show that the architecture will self-organize with more modules and more movements, and will handle the more complex situation.

Hypothesis 2: The experiment was designed to show that the context information will be crucial to separate the movements, as the number of movements was increased from the previous experiment.

Hypothesis 3: This experiment tests to a greater extent whether the multiple paired models architecture will understand the relationship between the context information and discrete movements.

6.3 Experiment 3 - YMCA

6.3.1 Description

This movement corresponds to forming the letters "YMCA" with the arms. The motion is from the dance to a song with the same name by a band called the Village People from New York, released in 1978[2]. The data for the movement was collected using the Pro Reflex system at the NU-lab at Dragvoll, see section 5.9. The context signal was divided to match the drawing of the letters with the arms, as can be seen in figure 6.3. The context information can be thought of as the output from a pitch-detector that would correspond to a specific letter. The experiment has four degrees of freedom, see section 7.4 for more details.

Figure 6.3: The YMCA motion, spelled out from left to right. Notice how the *M* does not look quite like it should; the upper arms should be aligned with the lower arms, i.e. my hands should be touching my shoulders. This was not done because the Pro Reflex tracking system would often confuse the fluorescent balls it they came too close. There is one ball on my wrist as well as my shoulder, and these two must be kept apart for Pro Reflex' sake.

6.3.2 Goal of the experiment

The experiment tests whether the multiple paired models architecture is suitable for imitating a human being. The previous two experiments were designed to show that the architecture works, this experiment seeks to demonstrate that the architecture is capable of dealing with real-world imitation.

Hypothesis 1: The experiment was designed to show that the multiple paired models architecture is capable of self-organizing with more movements, more modules and noisy data recorded from the real world.

Hypothesis 2: This experiment will test whether the context information helps the architecture to self-organize. In the previous experiments, the boundaries between the movements have been more easy to detect (i.e. they could more easily be detected just by looking at the desired states), since the movements were hand-coded by me. Now the different movements will be more prone to noise, and therefore the context information should play an important role in separating the control of movements into separate modules.

[2]For more information about the Village People, see for instance Wikipedias article at http://en.wikipedia.org/wiki/Village_People

Hypothesis 3: This experiment was designed to show that the multiple paired models architecture will realize the link between context information and the different movements under noisy circumstances, and that the responsibility predictor will reflect that the architecture has realized this relationship.

6.4 The different parameters of the architecture

The articles describing the architecture are sparse on details concerning the actual values on some of the parameters. The parameters were just educated guesses in the beginning, but after trial and error some values were settled upon. Notice however, that some of these variables differ from experiment to experiment. In chapter 7, these variables are specified for each experiment.

6.4.1 The learning rate δ

The learning rate was set to $\delta = 0.01$ early on in the experimentation, and remained at that value. Having some previous experience with backpropagation in neural networks, I knew that 0.01 was a good starting point. Also, 0.01 was used in [27]. Some variations were examined by varying δ from 0.01 to 0.1, but it was normally left at 0.01.

6.4.2 The gain K for the feedback-error

Initially $K = 60$, since that was the value used in [48]. It was quickly discovered that this value was way too high; the outputs of the inverse models are scaled to the range $[-1, 1]$, with $K = 60$ the joint angles were too high, leading to very abrupt and jerky movements. $K = 30$ was also tried out, with the same result. Then it was lowered drastically to $K = 0.5$, which yielded behaviours that were more smooth in their appearance, since the feedback error would not pull so strongly in certain directions. After more trial and error, it was raised to $K = 2$, which quickly helps the network find the correct joint angles. During the initial testing of the framework, the target states was gathered from breve itself, so I knew the correct joint velocities, which were never more than 1 in any direction, so these values seemed quite reasonable for the architecture to function well.

Later on, when the sampled data became available, it was not certain what the exact speed of the joint angles had been during movement, and therefore the output scaling and K needed to be adjusted again. This was done using trial and error.

6.4.3 σ in the likelihood function

$\sigma = 1$ for the initial testing, making the likelihood function the standard normal distribution. However, after several experiments it became clear that the likelihood function did not discern between different predictors, yielding the same value for all the forward modules. I tried setting $\sigma = 0.1$, which would give an increased likelihood for the predictor that was close to the actual state.

This made the models separate the a lot more easier. The normal distribution has 68% of the data within one standard deviation from the mean. The

range of the predictions are [0, 1] for each of the output nodes, which makes $\sigma = 1$ a fairly large set, covering a lot of the variations (remember though that the likelihood is based on the summed differences between actual and predicted state). When $\sigma = 0.1$, only small variations from the zero mean would give a high probability of being accurate, dividing the input space more clearly between the paired models. The σ value was varied from 0.1 to 0.15 to 0.2, but either 0.1 and 0.2 was used for the most part.

6.4.4 Calculation of λ, re-visited

As can be seen in the architecture of the system (figure 4.1), the λ parameter determines the learning of each of the neural networks (i.e. the forward and inverse models, and the responsibility predictor). During the initial testing, it became obvious that λ was close to the same value for all models (i.e. 0.5 when there were two models and 0.33 when there were three models. Indeed, in [27] Haruno indicates that the learning could proceed competitively, where modules with large λ values would get proportionally more error signal than those with smaller λ values. Exactly how this is done is not mentioned, so I devised the following biasing scheme: a constant (called the proportionate error constant P) was used to multiply the winning lambda value after the normalization. The lambda values were then normalized again. The winning lambda value would now be proportionally bigger than the other lambda values, allowing for more of the error signal to be sent to the winning lambda value.

An example is due: in the extreme case, the lambda values will be exactly the same, i.e. for three paired modules they will be `[ 0.33 0.33 0.33 ]`. The winning lambda value is multiplied with $P = 5$, yielding `[ 1.65 0.33 0.33 ]`. When normalized again, the lambda vector will be `[ 0.7143 0.1429 0.1429 ]`, more than doubling the winning lambda value, and increasing the distance to the other modules.

Note that the computation was done *after* the calculation of the final motor output, i.e. it did only affect how much error signal was sent back to the neural networks, not the motor output commands (recall that the motor output from each paired inverse/forward model is multiplied with λ before summation).

6.4.5 A note on run time for evaluating parameters

In order to determine what effect changing one parameter has on the model, a certain amount of epochs must be run. For most of the experiments I ran several thousand epochs, however the minimum was a 1000 epochs. Depending on how many steps was in the simulation, a 1000 epochs would take at least one or two hours. Needless to say, this led to a high cost of each of the experiments. This was before any attempts to optimize the breve/MatLab system. In addition, the breve simulator was not always as stable as one could have hoped for. Several times I left the system to run overnight, only to come in the next morning to find that the breve simulator had crashed. However, since communication between breve and MatLab was done via sockets, MatLab would just halt its execution of the simulation when breve crashed, so the training performed up to the point where breve crashed was not lost. I ran the simulation itself as a script instead of a function, which meant that all the variables in the script became available

to the general workspace if there was an unfortunate event that stopped the experiment.

Since the feedback error motor learning approach is used, the neural networks are trained on-line, i.e. by constantly interacting with the environment. This makes the system depend on the speed of the simulator, and is what makes the experiments so costly.

6.4.6 Stopping criterion

Closely related to run time of the system is knowing when to stop. In the initial experiments visual verification could be made of the progress of the experiment. However, another approach could be to look at how much the feedback controller influenced the system. When the feedback motor commands are close to zero, the models are good at predicting the next state and performing the correct motor movement.

For the experiments done in this thesis, there has been no specific stopping criterion. The experiments have run for the given number of epochs. This was mainly done because I simply did not know what kind of feedback error I should be considering good enough to be able to stop the simulation. Since each experiment was quite costly, it became more important to know how much time an experiment would take than to experiment with dynamic stopping methods.

6.4.7 Speed of the different breve versions

As previously mentioned, the run time of an experiment is typically several hours. In order to reduce run time, I first tried running breve without graphics, by using the `breve_cli` (breve command-line-interface) program. However, breve was very unstable when running without graphics. I tried out both 2.3 and 2.4beta, and both would crash soon after the program had started. After trying several times, none of the programs would run for the more than a couple of minutes, crashing with an uninformative "Bus error" or "Segmentation fault". The command-line-interface version of breve without graphics was clearly not suitable for doing experiments.

However, the command-line-interface version of breve with graphics was actually a bit faster. The graphics window did not update when motor commands were issued, which is why I suspect it was a bit faster. Some tests were done regarding simulation speed with both 2.3 and 2.4beta, with average and max/min times for 100 epochs, where each epoch consisted of 160 discrete timesteps (which in breve would be 2080 timesteps, since the data each iteration of the breve engine yields thirteen steps in the breve world). Each of these test runs ran for 100 epochs to even out any delays in network traffic. The results can be seen in table 6.1.

However, the fastest way to run breve was actually discovered by accident. By pressing "pause" on the breve simulator it runs a lot faster. According to the personal email communication with the author of breve, Jon Klein, this is due to the following: when in pause mode, the simulator will not need to lock breve when a web request is being made. Since I call the `manual-iterate` method for each web request, this will make the breve engine stop completely between the times I actually call it.

Version	Mean	Max	Min
2.3 cli	2.0061	2.1188	1.8868
2.3 cli (without graphics)	N/A	N/A	N/A
2.3 IDE	1.7914	1.8641	1.7200
2.3 IDE (pause mode)	1.3056	1.3882	1.2604
2.4b cli	1.9291	2.0180	1.8323
2.4b cli (without graphics)	2.5142	2.8606	2.2179
2.4b IDE	2.2106	3.2831	2.0467
2.4b IDE (pause mode)	1.3531	1.3990	1.3026

Table 6.1: Run times for 100 epochs, with each epoch consisting of 160 discrete steps, i.e. 2080 timesteps in the breve simulator.

breve 2.4b (both the IDE and cli versions) were initially not stable enough to allow for enough data to be collected (probably due to the beta status). However, after contacting Jon Klein about this issue, I was given a new version of 2.4b which did not crash.

breve 2.3 IDE in pause mode is slightly faster than breve 2.4 IDE in pause mode, and in addition the graphics rendering seemed a bit smoother in breve 2.3, making breve 2.3 IDE in pause mode the chosen breve version for the experiments.

6.5 A note on the iteration and integration stepsizes in breve

There are two different variables determining how far the simulation runs between each call to the `iterate` method, these are `iteration-step` and `integration-step`, where the former is bigger than the latter. `iteration-step` determines how many seconds are run for each call to `iterate`[3]. `integration-step` on the other hand, determines the length of each discrete step in the world within the `iteration-step` window. So if `iteration-step = 0.05` and `integration-step = 0.004` (the default values), the breve simulator will be iterated $12.5 \approx 13$ times for each call to `iterate`. This is important since my forward models shall predict sensory states in the future, and by knowing the relationship of these values, I know that for one external iteration of the breve engine, 13 small steps are taken, i.e. a length of time that is suitable for the forward models to predict[4].

[3]Actually, how often `iterate` is called, but since I have overridden that method it does not determine how often `iterate` is called, however it still determines the amount of seconds that is run for call to `manual-iterate`.

[4]This can be seen in the breve source code. By opening `Control.tz` it is possible to see that `iterate` calls `worldStep` in `kernel/internalFunctions/breveFunctionsWorld`, which in turn calls `slRunWorld` in `simulation/world.cc`. `slRunWorld` calls `slWorldStep` while the accumulated `integration-step` size is less than the total (i.e. `iteration-step` size). `slWorldStep` takes a single step in the breve environment.

Chapter 7

Results

After conducting the experiments, the multiple paired models architecture was saved along with the error log for the training period (as shown in figure 4.3). This chapter presents the analysis of the results. For each of the experiments, the parameters used to instantiate the multiple paired models architecture is listed, as seen in figure 4.3.

7.1 Description of the plots

There are five different types of figures for each of the experiments presented in this chapter. I will begin with describing the different figures, so they can easily be interpreted when discussing them. Each of the following subsections (i.e. sections numbered 7.1.x) will describe one figure each.

7.1.1 Performance during the training period

The first figure (7.1) shows the performance of the multiple paired models architecture for the entire training run. It has five components:

Performance of the forward models

See figure 7.1. The first two plots show the performance of the forward model for each module. The dashed line shows the summation of the prediction errors at each epoch[1], the solid line shows the prediction error multiplied with λ. The solid line thus shows the prediction error when it was in charge, i.e. had a high responsibility signal. When a forward module is not in charge, its responsibility signal is low and it will not receive a lot of its error signal to correct its bad output, and the error will remain high. Therefore, the error multiplied with λ is a better way of showing the *performance* of the forward model. In figure 7.1 they make up the two first plots, but the number is dependent on the number of modules in the multiple paired models architecture.

[1]Recall that one epoch is one pass through the entire desired state and context values, i.e. all the timesteps of an epoch. One experiment is typically run with several thousand epochs.

The total feedback error motor command

In figure 7.1 this is the third plot. It shows the summation[2] of the feedback error motor command for each epoch. This plot will show how the feedback error motor command decreases as the training passes. Ideally, it should go to zero, since $u_{\text{feedback}} = 0$ means that there was no need to correct the multiple paired models architecture, it has become a perfect controller.

The ratio of the feedback error motor command to the actual motor command

The feedback error motor command plot will show how it decreases with training, but it is also interesting to see how much of the actual motor command consist of the feedback error motor command. The third component is therefore the mean of the ratio of the feedback error motor command to the actual motor command for each epoch. For each timestep in an epoch, there is a ratio of the feedback error motor command to the actual motor command. The plot shows the mean ratio for an entire epoch, along with the standard deviation for each epoch. This plot is more illustrative than the previous plot, since it shows how the inverse models increasingly produce good motor commands, lessening the need for the feedback error motor command to correct the motor output. Ideally, this ratio should decline to zero. In figure 7.1 it is the fourth plot.

The number of transitions between the winning modules for each epoch

The fifth plot shows how many transitions there were between different modules during an entire epoch. There will always be at least one. When the winning module changes (i.e. the module with the highest responsibility signal λ), another transition is counted. If a multiple paired models architecture had two modules labeled A and B, and the order of the winning modules were A - B - A, that would count as three transitions (imagine the first transition happening at $t = 0$). This is perhaps the plot that shows best that the multiple paired models architecture self-organize over time. (Note, it does not show *how* the multiple paired models architecture self-organize over time, only that they do so.) It was intended for the number of transitions to equal the number of modules, i.e. if there were three modules A, B and C the order of the winning modules would be A - C - B (or another combination of non-recurring winning modules). This is the fifth plot in figure 7.1.

Summation of the absolute error for each epoch

The last plot in figure 7.1 shows the summation of the absolute error for each epoch. The absolute error is the absolute value of the difference between the desired state and actual state throughout an epoch. This plot shows how the performance of the multiple paired models architecture increases over time, as the absolute error decreases.

[2] Again, the summation is done over the entire epoch. The sum is then plotted at the corresponding epoch in the figure.

7.1.2 Performance of one epoch

The second type of figure (7.2) shows the performance of the multiple paired models architecture for an epoch. This is the performance achieved during *training*. It is possible to see the performance of the multiple paired models architecture at different plots because the actual state was logged during the training phase. That does *not* imply that for each epoch the multiple paired models was trained, and subsequently run without training to see how well it performed, that would have been too time-consuming (see section 5.5). The performance at epoch k shows the performance of the multiple paired models architecture as it was trained at epoch k. It has three components:

Performance of the forward models with λ and responsibility predictor

The first component consists of two plots; it shows the performance of each of the forward models in the multiple paired models architecture during an epoch. The correctness of the predictions of the forward model (first plot) and the responsibility predictor determine the responsibility signal (i.e. λ value), which is shown in the second plot. The plots show how well the forward model predicted, and when it was rewarded with a high responsibility signal. Normally, when the prediction error is low, it will have correspondingly high responsibility signal, and vice versa. In figure 7.2 the first four plots show the performance of the two forward models with their corresponding λ and responsibility predictor values. The responsibility predictor should ideally follow the λ signal. However, note that the responsibility predictor is always high for all the plots. This indicates that it does not work as intended. The multiple paired models architecture does not get the information provided by the context signals. Since this is common for all the plots, it has not been discussed for each experiment, instead it is discussed in chapter 8.

Ratio of the feedback error motor command to the total motor command

The second component shows the ratio of the feedback error motor command to the total motor command for each timestep in an epoch. After training, the ratio should be close to zero through the entire epoch. This plot shows when the inverse models produced bad motor commands, requiring the feedback motor to help correct the situation. It is the fifth plot in figure 7.2.

Superposition of all the λ values, along with context information

The last plot (the sixth plot in figure 7.2) shows the responsibility signals of the different modules in the same plot, making it easy to see when the winning module changes from one to another. The context information is shown in the same plot, as different shades of gray. The context information was thought to help the multiple paired models architecture self-organize in a way that would correspond to the context signals, see section 5.2.3. Ideally, the multiple paired models architecture would discover this relationship, and use the context signals to segment the responsibility of certain movements to particular modules (as was the intention from the design of the system). The λ plots would ideally follow

the context information plots (recall that the context information plots directly correspond to the movements in the motion).

7.1.3 The performance of the system compared to the desired state

The third type of figure (7.3) plots the performance of the system along with the desired state of the system for an epoch. The plot makes it easy to see the *performance error* of the system. There are different degrees of freedom for each experiment, and the figure has one plot for each. Each plot has a legend showing which colour is the desired state and which is the actual state of the system. The figure makes it easy to see how well the resulting behaviour matches that of the desired behaviour, irrespective of which module was in charge at what time. In addition, a video is produced for each of the experiments, showing the performance of the system compared to the desired trajectory. The video is among the attachments to the thesis. The plots can be seen in figure 7.3, which has 4 degrees of freedom, with one plot for each of them.

7.1.4 The performance of the system with λ values superposed

This figure (7.10) is a mixture of two previous plots; in this figure the performance of the system is plotted along with λ values, making it easy to see which module controlled which part of the total movement. The background colour is a paler version of the module's original colour used in previous plots (i.e. the first module is plotted in blue, and in this diagram the first module is plotted using a light blue colour).

7.1.5 Attractor plots

The fifth type of figure (7.11, 7.12) shows the attractor plots of the hidden layer of the forward and inverse models for an epoch. There are plots for each module, so the number of figures will depend on the number of modules in the multiple paired models architecture. The plots are done according to the procedure described in [32]; during an epoch the neural activity of the hidden layer is recorded. After the epoch, the activity of two adjacent nodes are plotted against each other. If there were four nodes at the hidden layer, the plots would be of nodes 1-2, 2-3 and 3-4, i.e. if there are n nodes there will be $n - 1$ attractor plots. At the beginning of each epoch the memory of the network is blank, therefore the initial transient trajectory is not included in the plot.

There is one subtle difference: Ito and Tani plotted the activity of the *context* nodes, not the hidden nodes (see figure 2.3). However, the context nodes represent the recurrent feedback loop, which is basically the same as the hidden layer nodes in my networks. The difference is that Ito and Tani had fewer context nodes than hidden nodes, but in my networks all the nodes at the hidden layer have recurrent connections, alleviating the need for context nodes. So my plots read "Hidden node 2" vs "Hidden node 1" instead of "Context node 2" vs "Context node 1", as it is written in [32]. Also, the trajectories in [32] were computed for each of the PB values determined for each pattern, this is obviously not done in my case, since I do not use the RNNPB.

For both the forward and inverse model two plots are shown: one for the entire epoch and one *filtered* plot. The filtered plot is made by only plotting neural activation where the λ value of the corresponding module was above 0.1. I.e. the neural activity would only be plotted when the module was actually contributing to the motor output. This was done to make it possible to see the difference of the attractors when the module was doing something useful (i.e. having a high λ value) compared to the attractors recorded during the entire epoch. Note that the attractors are only plotted for the last epoch. This is because I did not want to log the neural activity during the training run - the log files were already in the 10s of MBs (the results gathered from the YMCA was 271MB, for instance).

Why plot the attractors? I expected the attractors of the hidden nodes to show nice shapes, indicating a stable memory, as in [32]. However, Ito and Tani state that they selected *randomly* two context nodes, and plotted their attractors. I therefore wanted to plot *all* the attractors, to see if all the plots would display nice shapes. As the figures will show, there were many plots that did not show nice attractors at all, instead they were erratic and hard to interpret. I suspect that Ito and Tani also plotted all the context node attractors, and picked the ones with nice attractors.

This concludes the description of the various plots. Notice that at every plot, there will be a reference to the section describing the plot, in case that the reader wants to refresh the memory of what the plot showed. The results of the experiments presented in chapter 6 will be presented.

7.2 The two Ks

Name of MatLab file	`breve_breve_matlab_Ks.m`
Name of breve file	`Tiny Dancer - K4DOF.tz`
Degrees of freedom in breve	4
Length of desired state	120
Number of paired inverse/forward models	2
Number of epochs trained	4000
Feedback error gain K	2
Output gain M	2
σ in the likelihood function	0.20
Error proportion P	5
Size of forward/inverse neural network	8-4-4
Size of responsibility predictor network	2-2-1
Learning rate δ	0.01
Active joints	Right shoulder X joint angle, Left shoulder X joint angle, Right hip Y joint angle, Left hip Y joint angle

7.2.1 The learning of each of the modules

Figure 7.1 shows a clear separation in the degree of learning of the two modules. The second module actually has an increasing total error, while the error * λ is low for most of the time. This indicates that the module has found its niche early

on, and specializes on that movement. The first module has a decreasing total error and error * λ curve, indicating it is responsible for most of the movement. A look at the performance plot of the last epoch (figure 7.2) shows that the second module is responsible for the movement of the simulator for about 25 timesteps, the rest is controlled by the first module.

Figure 7.1 also shows that around epoch 1000, the system only had two controlling modules. Indeed, figure 7.4 shows that at the 1000th epoch the switch between the modules was almost as intended by design. However, it is also clear that the feedback error motor command was quite large during this period, indicating that the second module did not produce very good motor commands. By looking at the figure of the performance of the system at the same epoch (figure 7.5) it is clear that the left shoulder joint and left hip joint are badly controlled. The second module was the winning module during this period of time, and received more of its error signal for training. This should have made the second module more competent in controlling the latter part, but it fails to make use of its proportionally bigger error signal to correct the performance of its inverse model.

7.2.2 Switching between controlling modules

It is at epoch 1058 that the first module gets control of the last timesteps for the first time. At epoch 1057 (figure 7.6) the first module is gaining importance towards the end of the run, and in epoch 1058 (figure 7.7) it has become the winning module for the last timesteps - it is at the very last timestep that the first module has gained more control than the second. This is more evident in epoch 1100 (figure 7.8). From this point on, the first module takes more charge towards the end of the movement, effectively reducing the period that the second module is in control. By epoch 1500 (figure 7.9), the controlling period of the first module is even further increased. Compared to the last epoch, the entire controlling period of the second module has been shifted to the left, i.e. earlier on in the sequence. At epoch 1500 it controls the robot from timestep 85 - 110, whereas in the last epoch (figure 7.2) it is in charge from timestep 80 - 105 (the timesteps are approximate values, read from looking at the plot themselves). The plots for epoch 1500 and the last epoch also show that it is during the switching period that the feedback error motor signal is stronger.

What does this mean? It seems like the two different modules specialize in some part of the movement early on, especially the second module. The modules start out with random weights, so initially they could be equally suited for any movement. Since this is a self-organizing system, it goes to show that intentions put there by design might not be the way the system will find, as is the case of the context values and the winning modules.

Figure 7.10 shows which module controlled which part of the movement, as was the state at the last epoch. The first module controls most of the movement, but it seems as the second module is crucial for making the left shoulder and hip change directions. Up until timestep 85 the second module is in control and outputs a near constant velocity to move the left shoulder and hip. But it is the second module that changes the direction of the movement. In this respect, the modules have separate areas of responsibility according to intention. Moving a joint in a straight line requires the multiple paired models architecture to output a constant velocity. The first module controls all the movement of the

right shoulder and hip, but it is the second module that does the crucial part of the second movement. In this respect, the modules have separated nicely, as was intended. However, figure 7.2 also shows that the ratio of the feedback error motor command to the motor output from the multiple paired models architecture is bigger during the entire second period of the movement (i.e. as defined by the context info), indicating that the modules are not completely confident of the movement.

7.2.3 Attractor plots

Figure 7.11 shows that the forward model of the first module seems to follow a pattern in its attractor. The inverse model attractors are more chaotic, and do not display a certain pattern. The filtered plots are almost entirely equal, which is not surprising since the first module is in control of the movement most of the time.

The attractor plots of the second module (figure 7.12) are more interesting, they clearly show a distinct pattern of neural activity. The filtered plots are almost identical to the total plots here as well, indicating that the module is somehow constantly in the same attractor, but it is only at a certain time that this attractor is a good predictor and controller of the robot. This could explain why the second module fails to become the controlling module for the last period of the movement (as was discussed in section 7.2.1); it is specialized for that particular movement and even though it was trained for the entire latter part of the movement, the first module did more easily adapt even when receiving a lower ratio of the error signal.

7.2.4 Was the goal met?

The goal (see section 7.2.4) was to show that the multiple paired models architecture works, and that it would self-organize into having one module controlling for one behaviour. Although the separation of the modules did not happen exactly as intended, the separation was quite close to intention. The multiple paired models architecture as a whole works as intended, i.e. it is a good controller for the simulated robot. In terms of hypotheses the results show:

Hypothesis 1: The different movements did self-organize to different modules, although there was some overlap between the movements.

Hypothesis 2: The switching of the controlling module was almost as intended, but this was not due to the context information. Instead, the multiple paired models architecture managed to understand these changes without the context information.

Hypothesis 3: The context information was not used at all, as can be seen on the plots. No relation between context information and movements were discovered, due to the responsibility predictor output being high constantly.

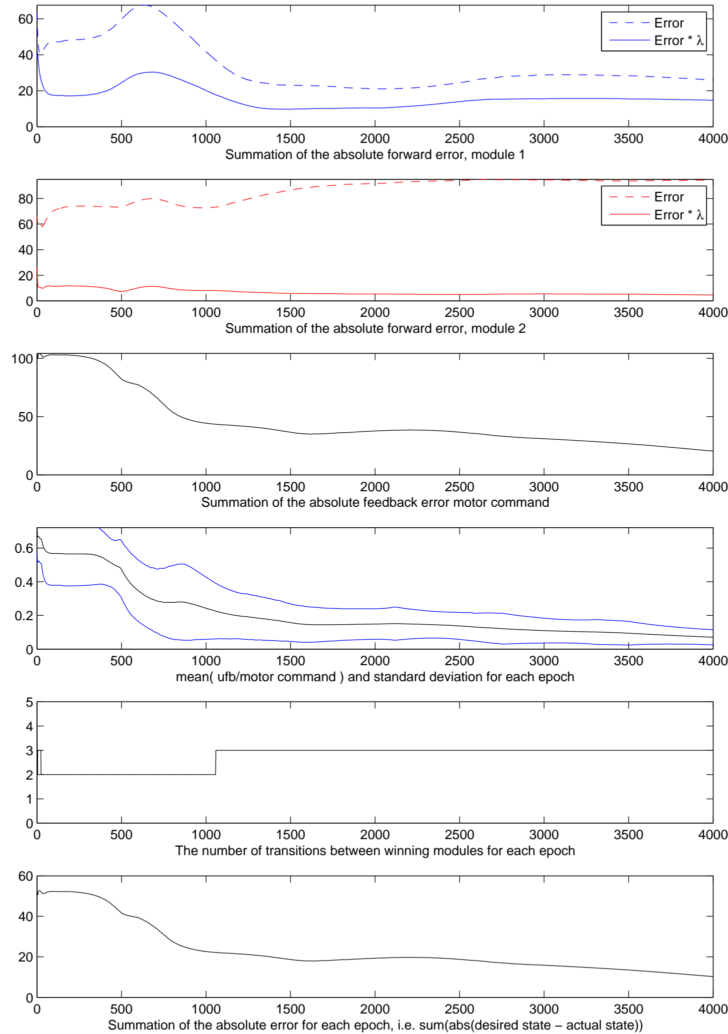


Figure 7.1: (7.1.1) The training period of the two Ks. The x-axis shows the number of epochs. The prediction errors of the first (blue) and second (red) module can be seen in the two upmost plots. Plot number three shows how the feedback error motor command decreases over time, as the predictions of the forward models get better. The ratio of the feedback error motor command to the total motor command is shown in the fourth plot. It is the *mean* ratio of each epoch. Notice how it moves towards zero as the training progresses. The fifth plot shows how many transitions there are between winning module at each epoch. The last plot shows the error of the multiple paired models architecture as a whole.

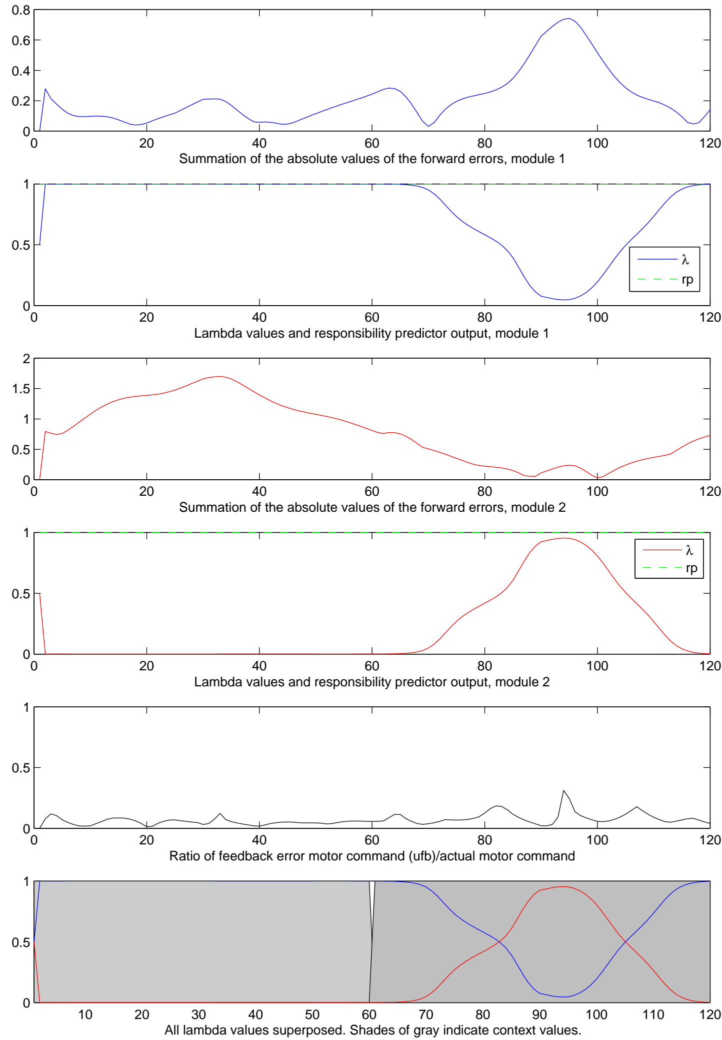


Figure 7.2: (7.1.2) The last epoch of the two Ks. The x-axis shows the timesteps of the epoch. The two plots in blue show the performance of the forward model and the responsibility predictor and the resulting λ values for the first module. The plots in red show the same for the second module. The ratio of the feedback error motor command to the total motor command is shown in the fifth plot. The ratio remains close to zero, with the highest point during the control period of the second module. The λ values of both modules are superposed, along with the context information in the sixth plot. The λ values should ideally follow the context values, however they are not very far from doing so.

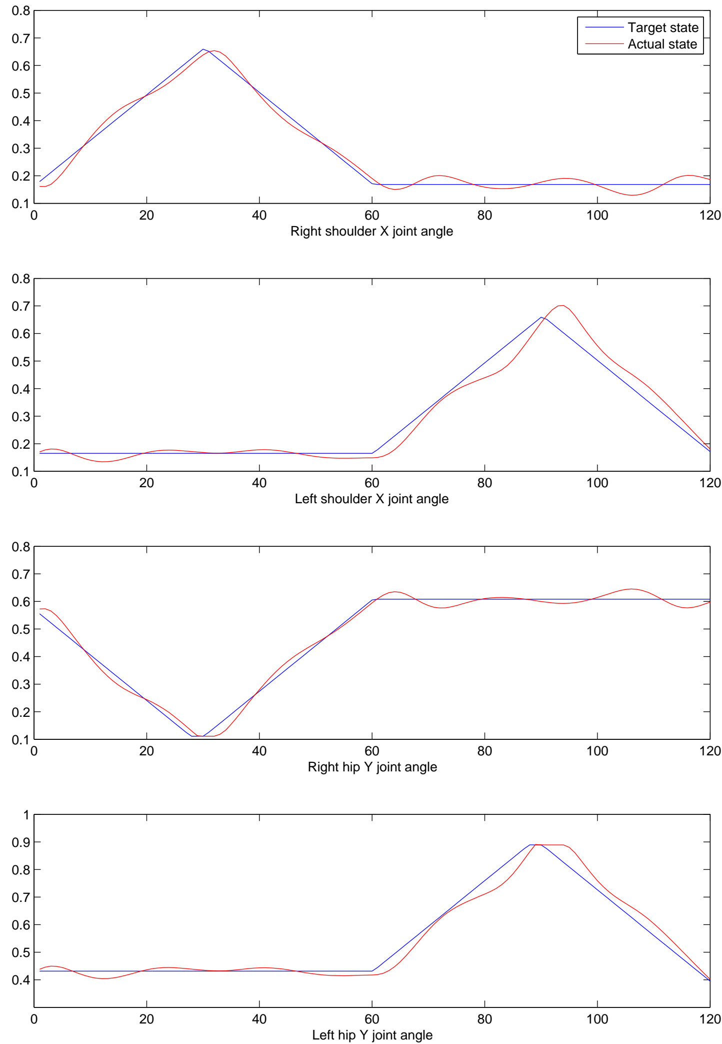


Figure 7.3: (7.1.3) Performance of the multiple paired models architecture compared to the desired trajectory, the last epoch, the two Ks. Each plot shows the desired and actual state for one of the degrees of freedom (which can be read on the label at the x-axis). The x-axis shows the timesteps. The plots show that the multiple paired models architecture performs well.

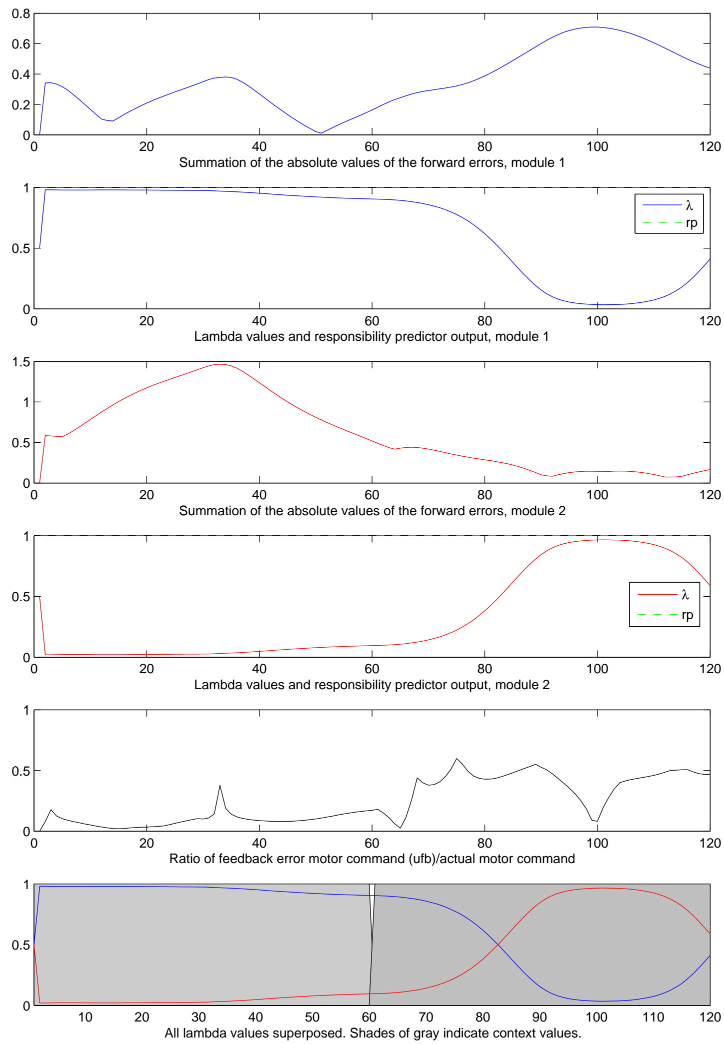


Figure 7.4: (7.1.2) The performance at epoch 1000, the two Ks. There is a clearer separation of the λ values, however notice the higher ratio of feedback error command to total motor command, indicating that the second module does not produce good motor commands.

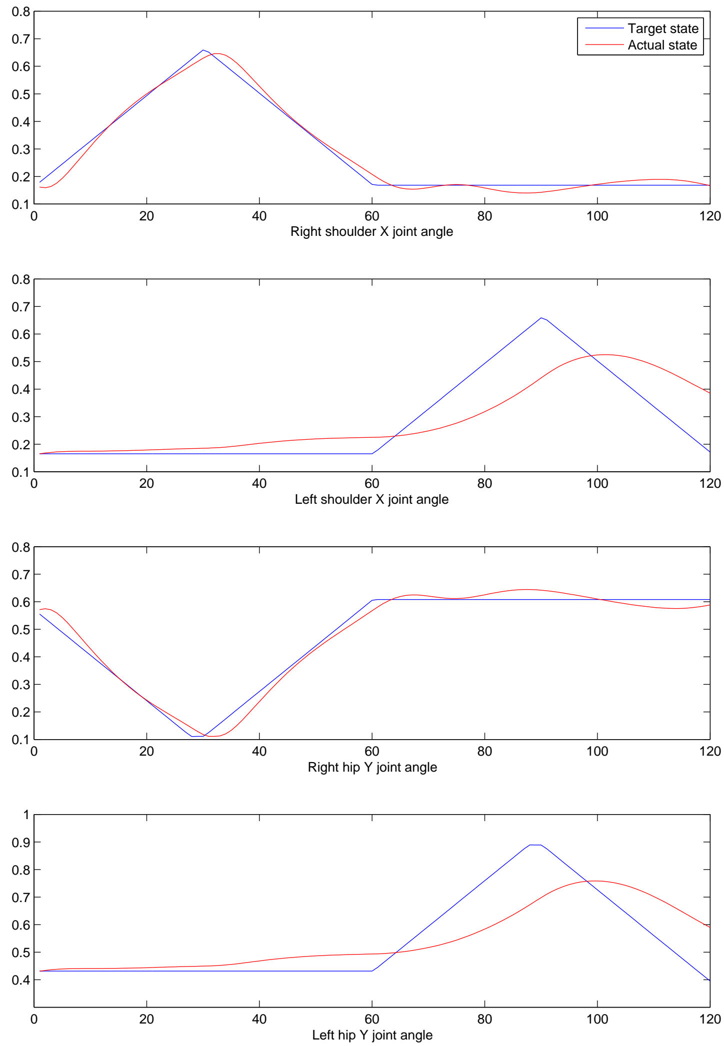


Figure 7.5: (7.1.3) Desired and actual trajectory, epoch 1000, the two Ks. The right shoulder X joint angle and the right hip Y joint angle are controlled nicely throughout the epoch, the control of the left shoulder and hip is more erroneous.

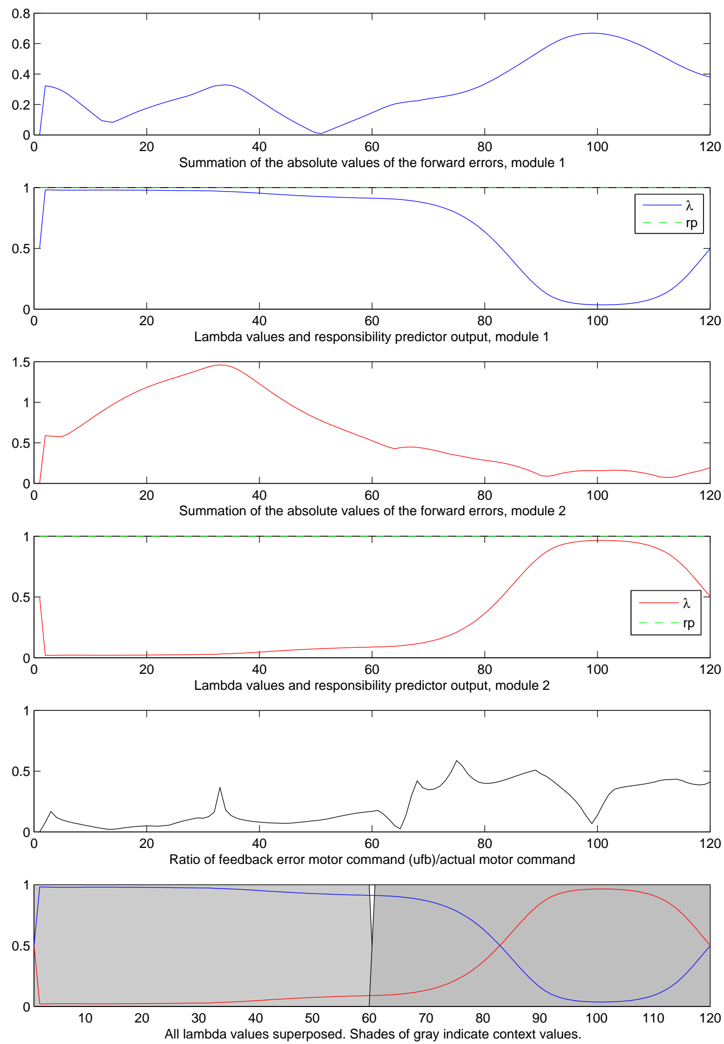


Figure 7.6: (7.1.2) Performance at epoch 1057, the two Ks. This figure, along with figure 7.7, shows when the first module starts to gain more control over the last part of the motion.

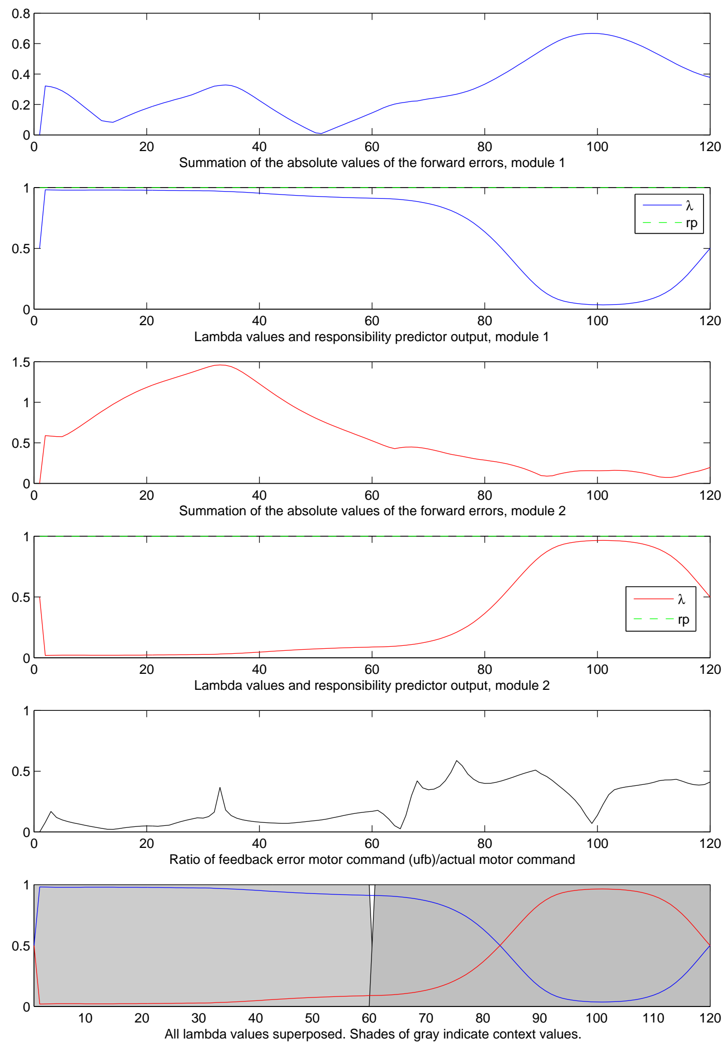


Figure 7.7: (7.1.2) Performance at epoch 1058, the two Ks. The first module is now the winning module at the last timestep. After this epoch, the first module will continue gaining control, as can be seen in figure 7.8.

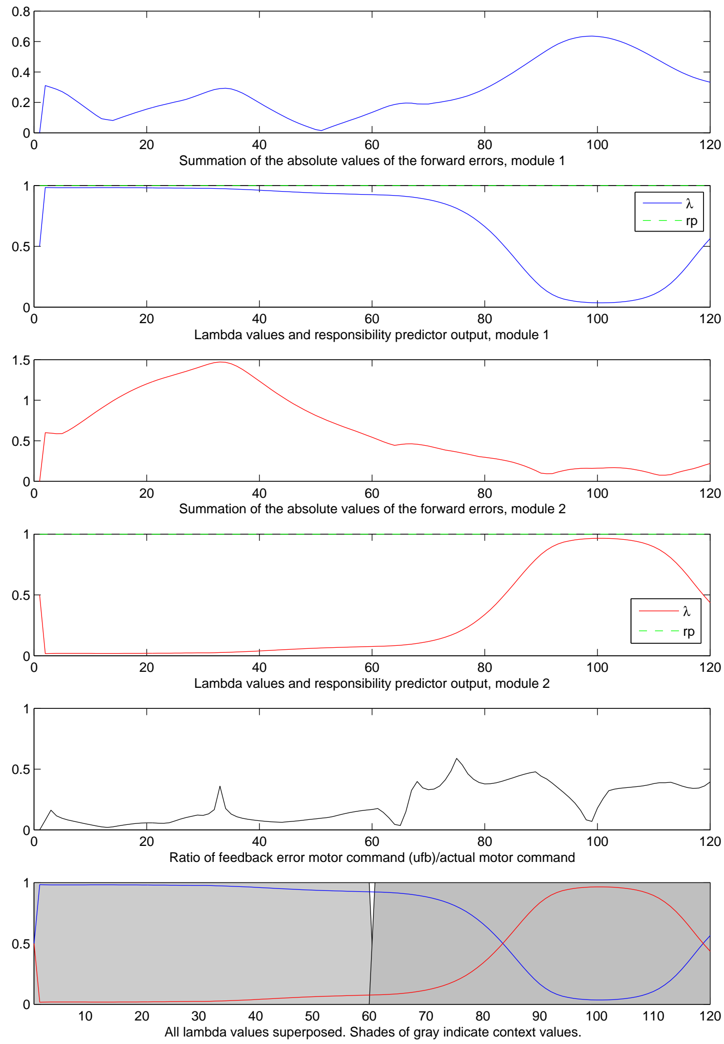


Figure 7.8: (7.1.2) Performance at epoch 1100, the two Ks. The first module is now gaining control towards the end of the motion.

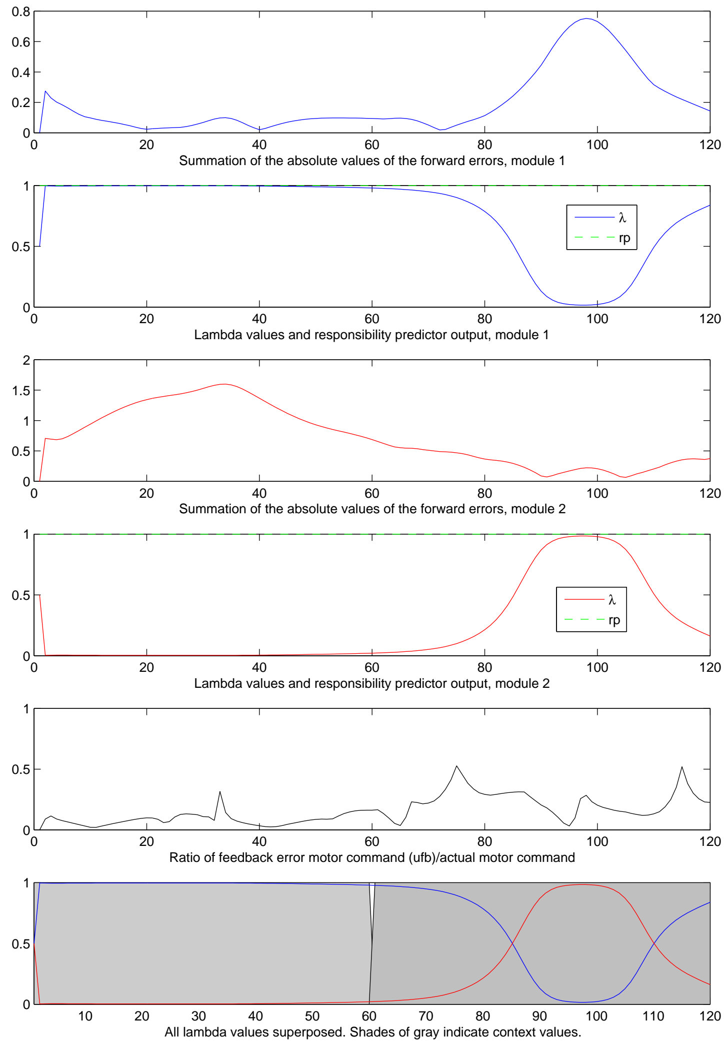


Figure 7.9: (7.1.2) Performance at epoch 1500, the two Ks. The first module has now gained even more control towards the end of the motion.

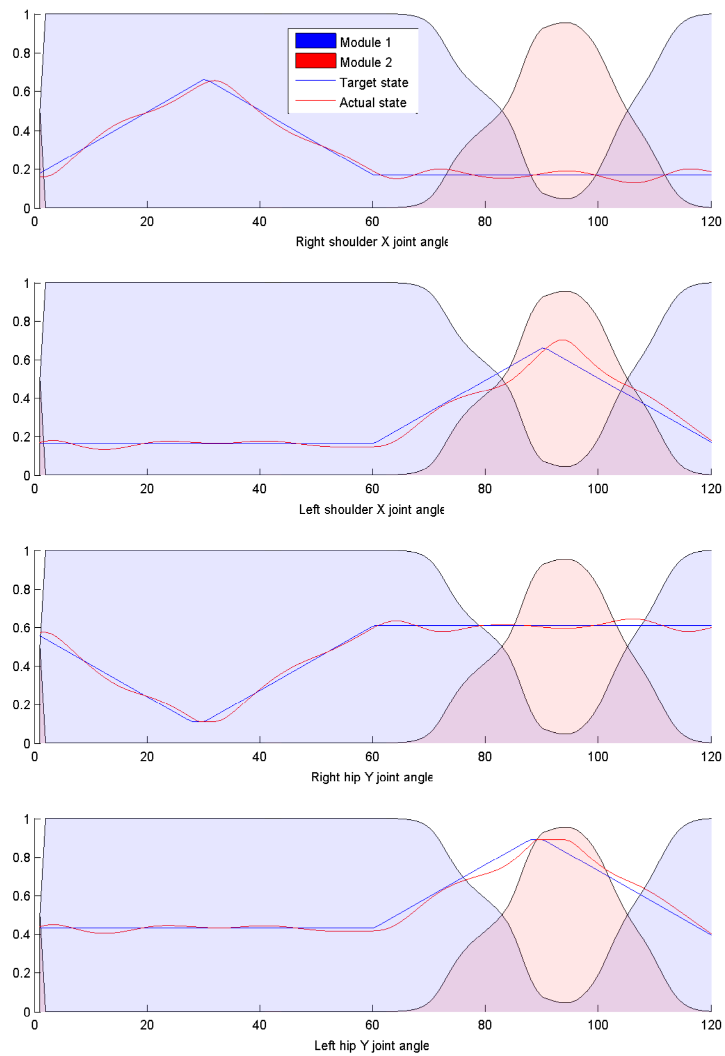


Figure 7.10: (7.1.4) The state trajectories along with controlling modules, the last epoch, the two Ks. The second module is crucial for shifting the direction of the arm.

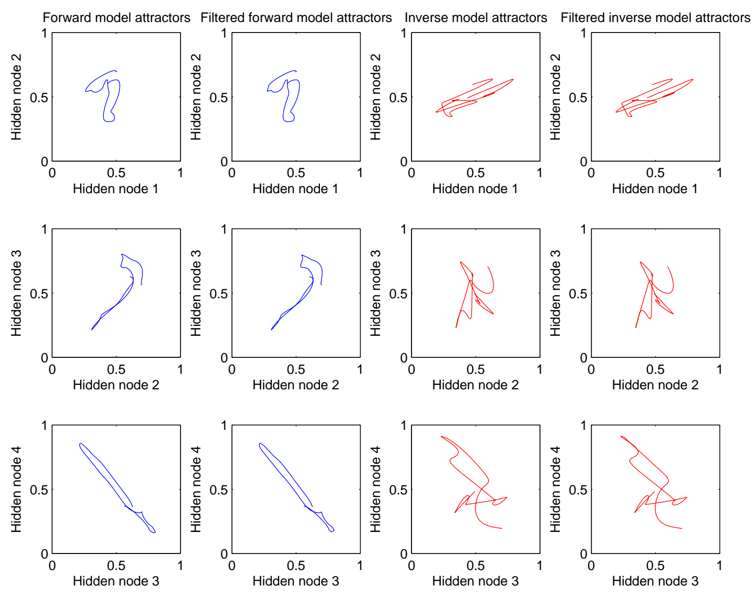


Figure 7.11: (7.1.5) Attractor plots, module 1, the two Ks. The filtered and unfiltered plots are more or less the same, which is not surprising, considering that the first module is in control most of the time. The forward model attractors seem to follow a pattern, but the inverse model attractors are more chaotic.

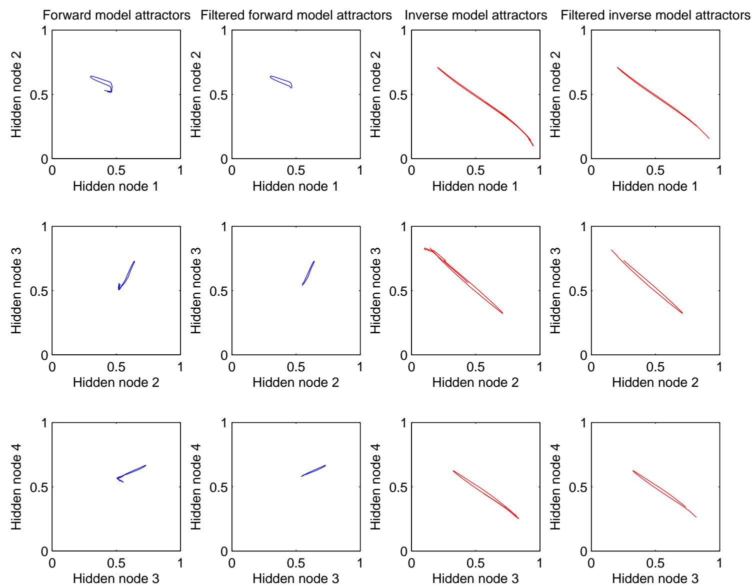


Figure 7.12: (7.1.5) Attractor plots, module 2, the two Ks. All the plots show more stable patterns, indicating that the module is very specialized for the part it controls the robot.

7.3 The cheerleader

Name of MatLab file	`breve_breve_matlab_cheerleader.m`
Name of breve file	`Tiny Dancer - cheerleader7DOF.tz`
Degrees of freedom in breve	7
Length of desired state	160
Number of paired inverse/forward models	3
Number of epochs trained	8000
Feedback error gain K	2
Output gain M	2
σ in the likelihood function	0.20
Error proportion P	5
Size of forward/inverse neural network	14-7-7
Size of responsibility predictor network	3-3-1
Learning rate δ	0.01
Active joints	Right shoulder X joint angle, Right shoulder Y joint angle, Left shoulder X joint angle, Left shoulder Y joint angle, Right hip X joint angle, Right hip Y joint angle, Left hip X joint angle

7.3.1 The learning of each of the modules

Figure 7.13 shows how two of the modules specialize early on in the training period; the same thing that was observed in the previous experiment. The first module starts out with a very low prediction error * λ, and it remains low. The module is trained specifically for its small portion of the total movement, and becomes even worse at controlling anything else. The second module is subject to the same issue, but it learns to a greater extent than the first module. As with the first module, the error curve is increasing, but it starts decreasing after epoch 2000. The error * λ remains more or less the same after epoch 2000.

It is the third module that learns most during the training run. The total error and the error * λ both decrease as training proceeds. Figure 7.14 shows that it is indeed the third module that has most of the control in the last epoch. The second module has some short spikes of about equal length in both the first and third movement, whereas the first module controls only one specific area of the third movement.

7.3.2 Switching between controlling modules

In figure 7.16 it can be observed how the third module controls most of the movement of the robot at the last epoch. The second module controls the robot in a short period of time in the first movement, and upon closer examination it is when the joint angle velocity changes sign for the right shoulder X joint angle, the left shoulder X joint angle, right hip X joint angle and to some extent also the left hip X joint angle (recall that the red line is the actual state, i.e. the performance of the robot. The peak of the desired target (the blue line)

is slightly before the actual state, however the peak of the actual state occurs when the second module has control over the robot, at about timesteps 35-40). Similarly, the first module controls the robot when the joint angle velocity of the right shoulder X and Y joint angle, left shoulder X and Y joint angle and right and left hip X joint angle change sign. Interestingly, the second module "pads" the first module in almost identically long periods of time. Even though the third module is in control most of the time, one could say that the first and second module plays important parts when changes occur in the first and third movement, as was observed for the two Ks experiment as well.

Epoch 2000 is where the learning curve of the second module starts decreasing, after having increased for the last epochs. Figure 7.17 reveals that module 3 has more control over the first movement, but that the second module is almost equally in control of the first movement. Figure 7.18 shows the control related to the joints. During the second movement, the actual state differs substantially from the target state. It can be seen as the third module learns to control the robot correctly for the second movement, but gradually surrenders control to the second module for the important changes in the first movement.

The multiple paired models architecture has a lot of oscillations in the number of transitions between winning modules, which can be seen in figure 7.13 in the sixth plot. Although it seems to stabilize after about 5000 epochs, there is still some changes towards the end of the training run.

After the 7000th epoch there are some oscillation in the number of transitions. The first spike in changes after epoch 7000 is at epoch 7140, which lasts until epoch 7282. Figure shows epoch 7200 and figure epoch 7400 (which is in between the transitions occuring at epoch 7379 and epoch 7445), and it is evident that these changes are very small and not substantial to the total performance of the system, showing that the system remains rather stable after epoch 5000. Figure 7.13 also shows that the summation of the absolute error is rather stable after epoch 5000.

7.3.3 Attractor plots

Figure 7.21 shows the attractor plots for the first module. The activity of the forward model (both for the entire epoch and the filtered plot) shows little variation, indicating that the module is very specialized in the output space. The inverse model attractors show patterns with more variation. The filtered patterns are close to the patterns for the whole epoch, indicating that the inverse model produces quite similar motor outputs the entire time, but the outputs are correct only for a small period of time, as can from the short period of time it is in control (figure 7.2).

The attractor plots of the second module is shown in figure 7.22. The patterns of the forward model are quite consistent, whereas the patterns of the inverse model are more erratic. The filtered plots of the inverse model are a bit more stable, but still not too coherent. This can explain why the feedback error motor command is bigger during the periods when the second module influences the total motor output (see figure 7.14).

Figure 7.23 shows the attractor plots of the dominating module, namely the third module. The forward model plots display some interesting patterns, all of them seem to travel back and forth in a boomerang-shaped pattern, with context node 2-1 and 7-6 having an extra branch, making the pattern look like

a propeller. These attractors represent the memory of the states of the total movement, since this is the module controlling the robot for most of the time. The attractors of the inverse model are more chaotic, and do not display any symmetry. Perhaps this is the reason why the other modules must “aid” the third module into getting the correct movement at certain important turning points of the movement, it is simply too complex for the third module to control the robot all the time. The filtered and unfiltered plots are almost exactly the same, but that is not so surprising, considering it is control most of the time.

7.3.4 Was the goal met?

The goal was to show that the architecture would self-organize in a more complex situation (see section 6.2.2). The network did self-organize, however the separation of the modules did not happen as intended. However, the different modules played important parts in determining the behaviour of the system. In addition, the system as a whole functioned well. Results of the experiments related to the working hypotheses:

Hypothesis 1: As in the previous experiment, the self-organization did not happen exactly as intended, although the different modules all play important parts in making the architecture function well.

Hypothesis 2: The λ values did not self-organize as intended, and in a lesser degree than the previous experiment. Again, this was not due to the context information, as the responsibility predictor had a high output during the entire epoch.

Hypothesis 3: There was no correlation between the context information and the movements, as in the previous experiment, since the responsibility predictor output was constantly high.

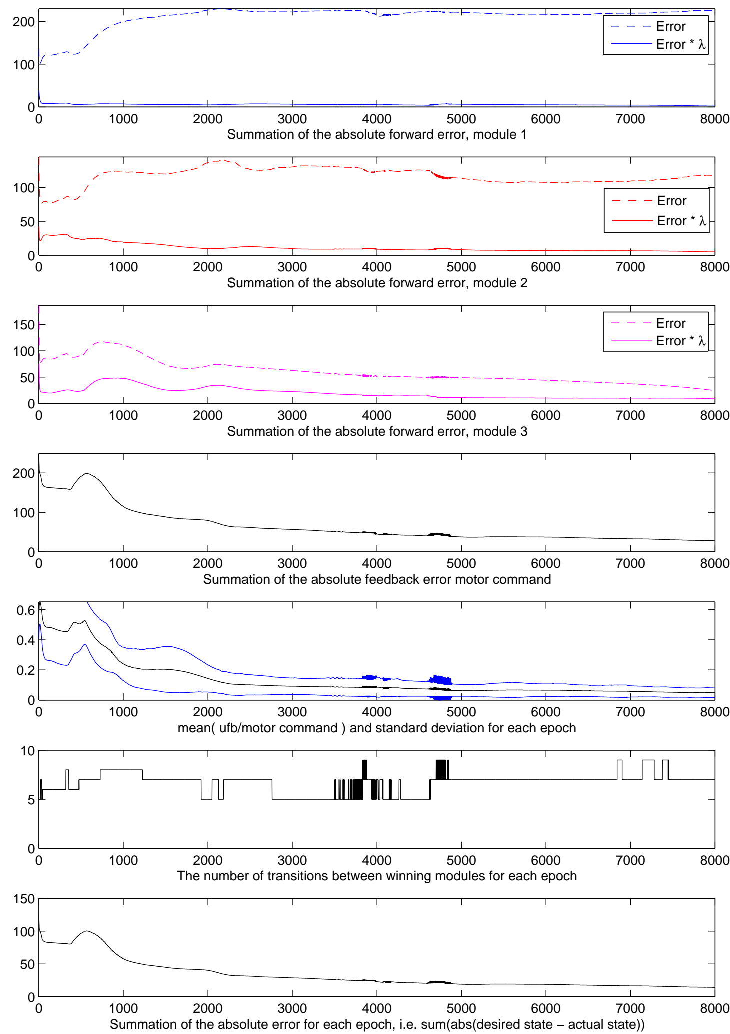


Figure 7.13: (7.1.1) The training run of the cheerleader. The x-axis shows the number of epochs. The prediction errors of the first (blue), second (red) and third (magenta) module can be seen in the three upper plots. Plots 4-7 are the same as plots 3-6 in figure 7.1. Notice how there are some oscillations in the number of transitions between the winning modules, indicating that several modules are competing for control, and that they are fairly equally well controlling the robot.

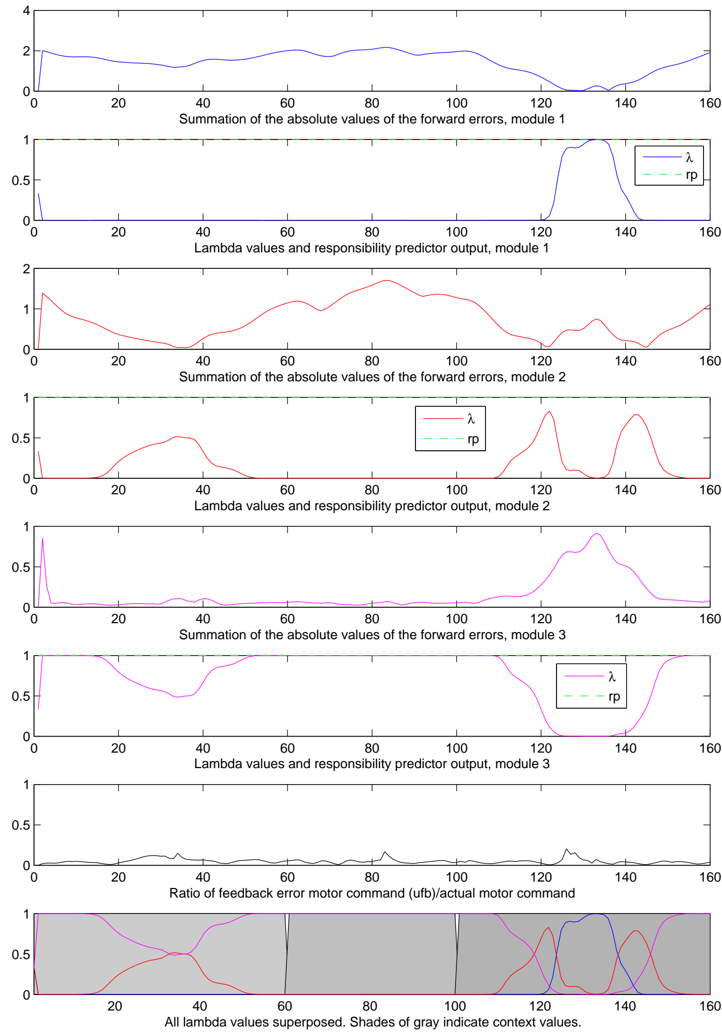


Figure 7.14: (7.1.2) The last epoch of the cheerleader. The x-axis shows the timesteps of the epoch. The plots show the same as in figure 7.2, with the addition of the performance of the third module, shown in magenta. The ratio of the feedback error motor command is low during the entire epoch, with a peak when the first module is in control.

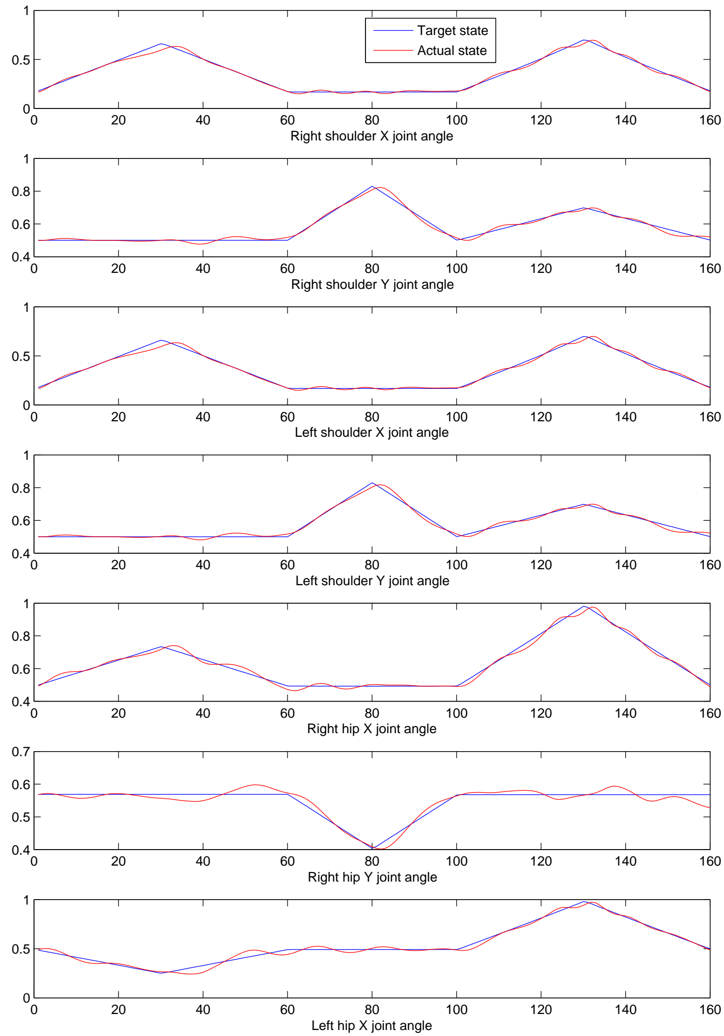


Figure 7.15: (7.1.3) The target state and the actual state of the cheerleader, at the last epoch. Each plot shows the desired and actual state for one of the degrees of freedom (see the label close to the x-axis). The x-axis shows the timesteps. The performance is very good, with little deviation from the desired trajectory.

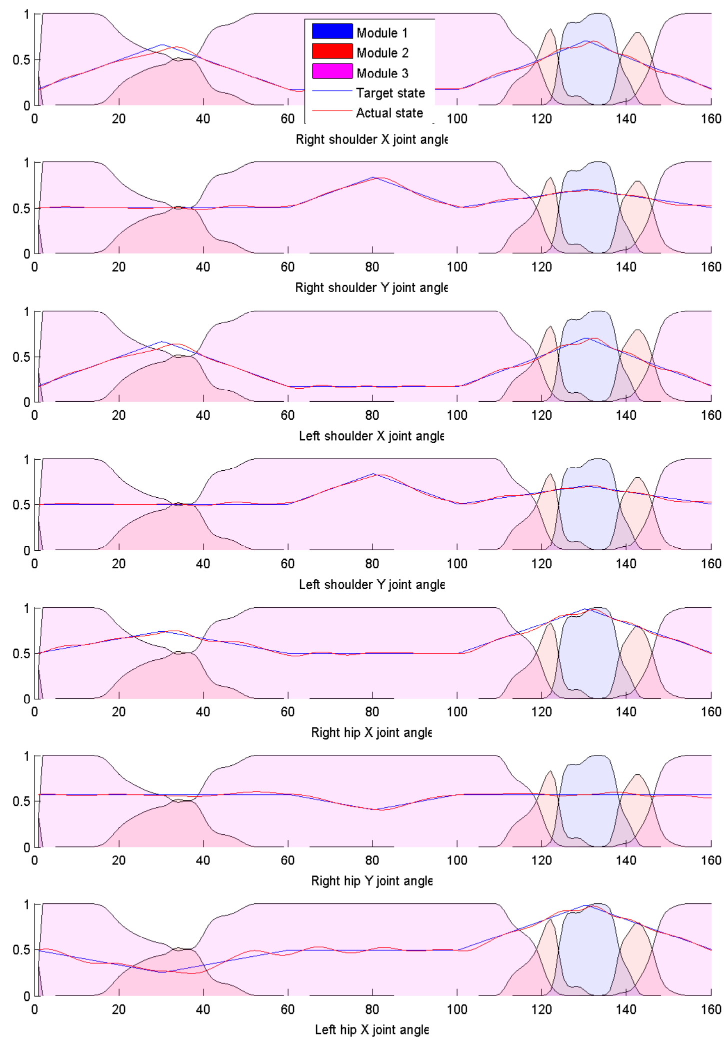


Figure 7.16: (7.1.4) The λ and target/actual trajectory of the cheerleader at the last epoch. The first and second module are important for changing the direction of the different joints during the first and last movement.

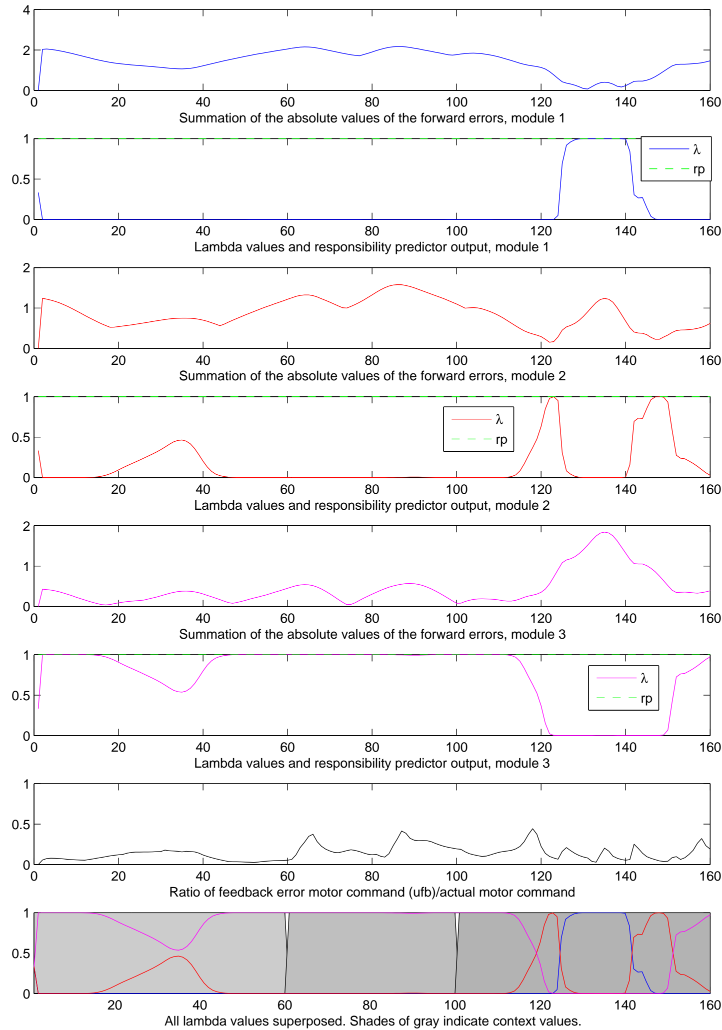


Figure 7.17: (7.1.2) The performance plot at epoch 2000, the cheerleader. This is the epoch where the second module starts decreasing its error predictions. Notice how much the error of the second module has decreased compared to the plot of the last epoch (figure 7.14). The fact that the decrease in error happens rather late in the training period, is a sign that the architecture is capable of readjusting even though the training has gone on for some time. Normally when training neural networks, the error curve decreases rapidly in the beginning and then remains low - here it the error curve is high, but the self-organizing process manages to push the module in the right direction.

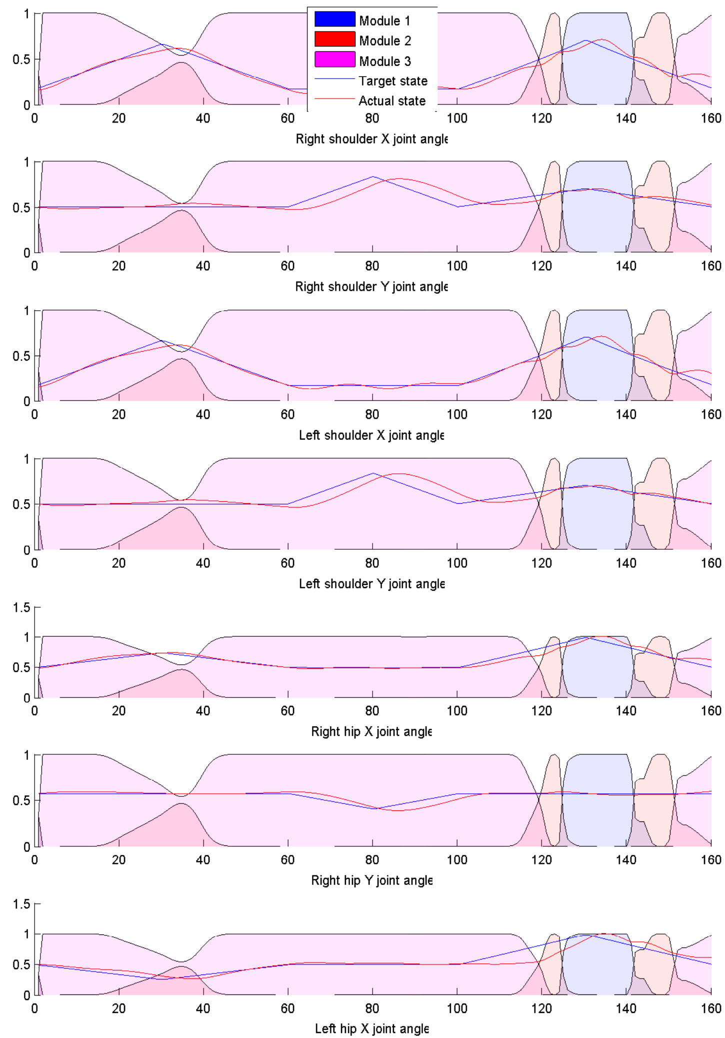


Figure 7.18: (7.1.4) Desired and actual trajectory, along with λ values, epoch 2000, the cheerleader. The third module is more in control during the first movement, than it is at the last epoch (figure 7.15).

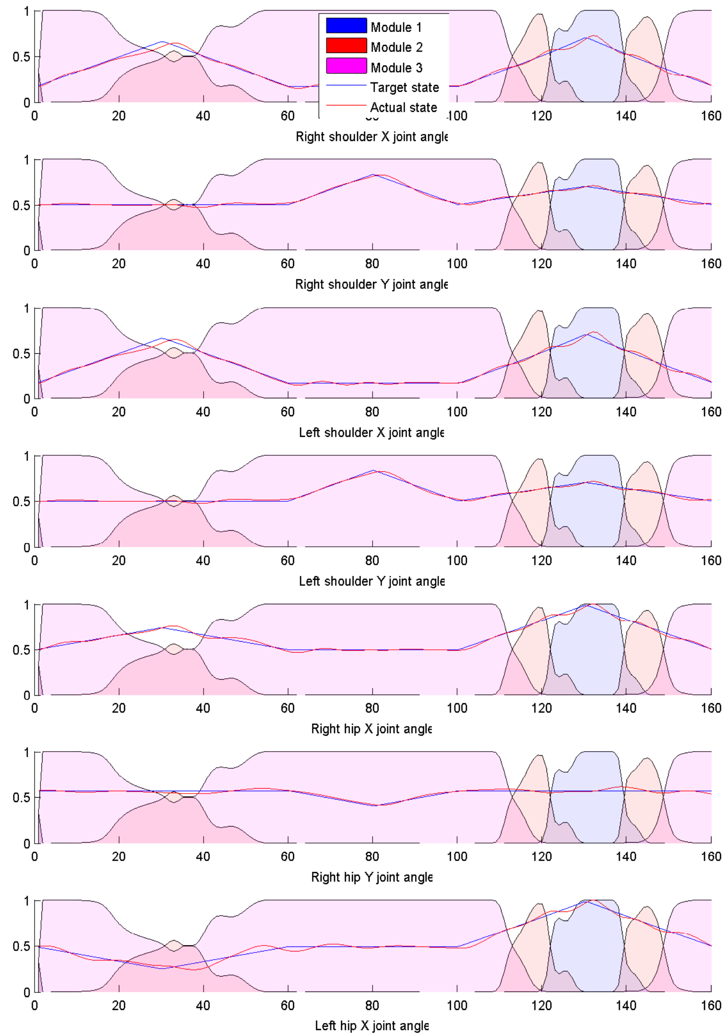


Figure 7.19: (7.1.4) Target and actual state with λ values, epoch 7200, the cheerleader. This figure (along with figure 7.20) shows that even though there are some changes in the number of transitions between the winning modules, these changes are very small and does not influence the total performance of the multiple paired models architecture to a great extent.

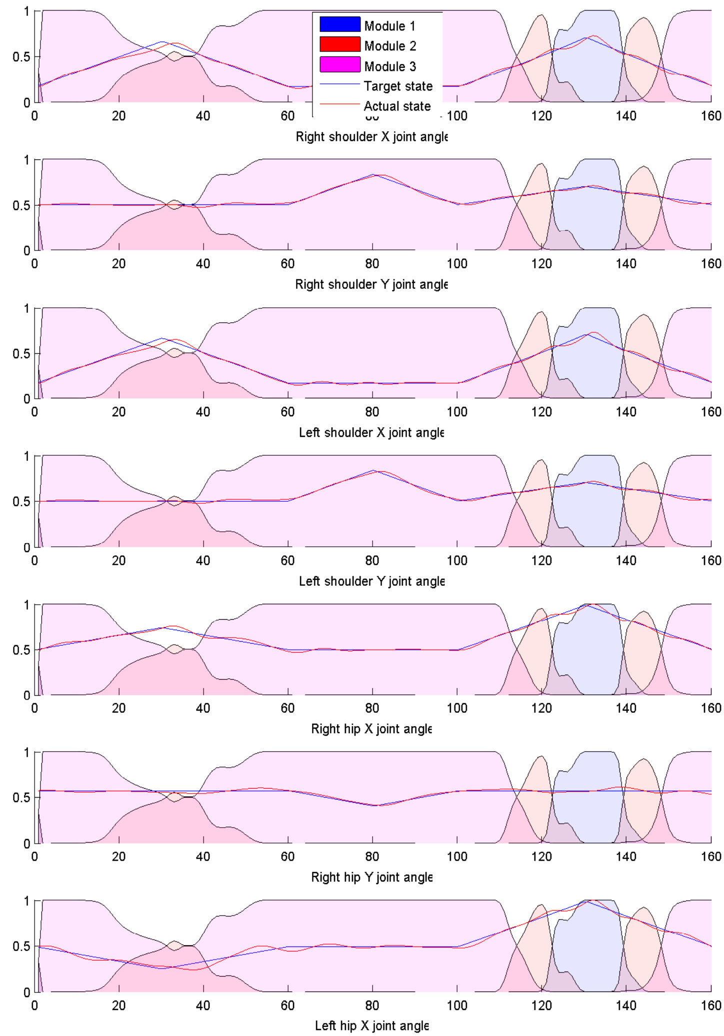


Figure 7.20: (7.1.4) Target and actual state with λ values, epoch 7400, the cheerleader. This figure and figure 7.19 show that even though the number of transitions are changing, the change is very small and not very significant, making the changes not very noticeable in the total performance of the multiple paired models architecture.

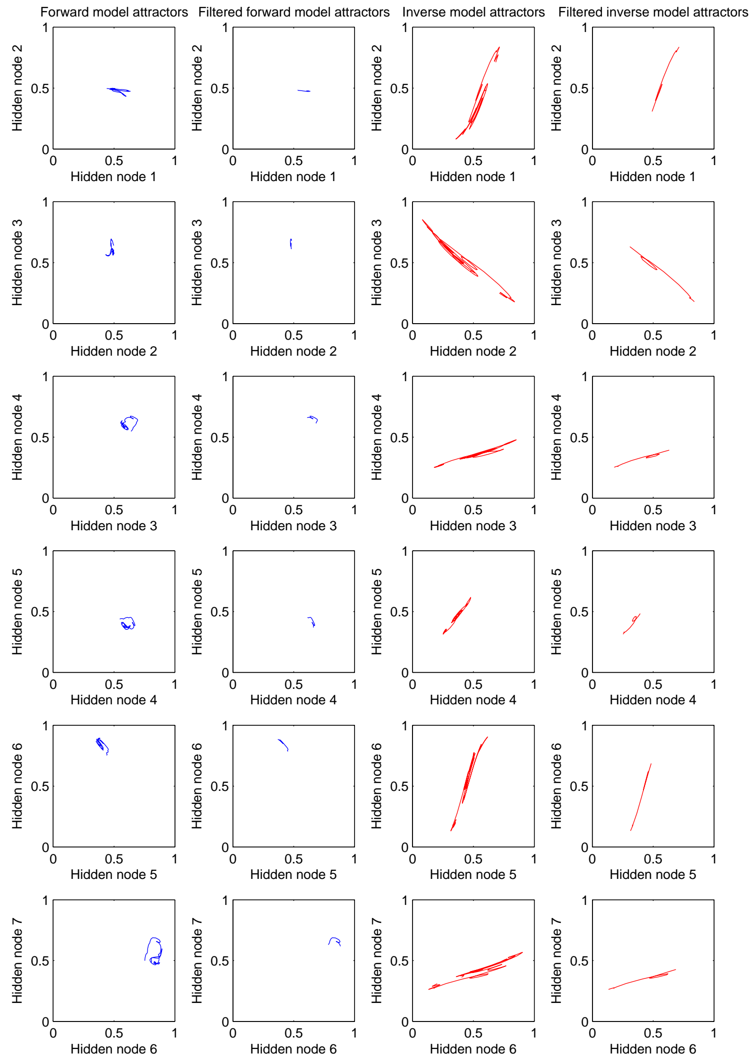


Figure 7.21: (7.1.5) Attractor plots, module 1, the cheerleader. There is little variation in the plots, indicating that the module is very specialized.

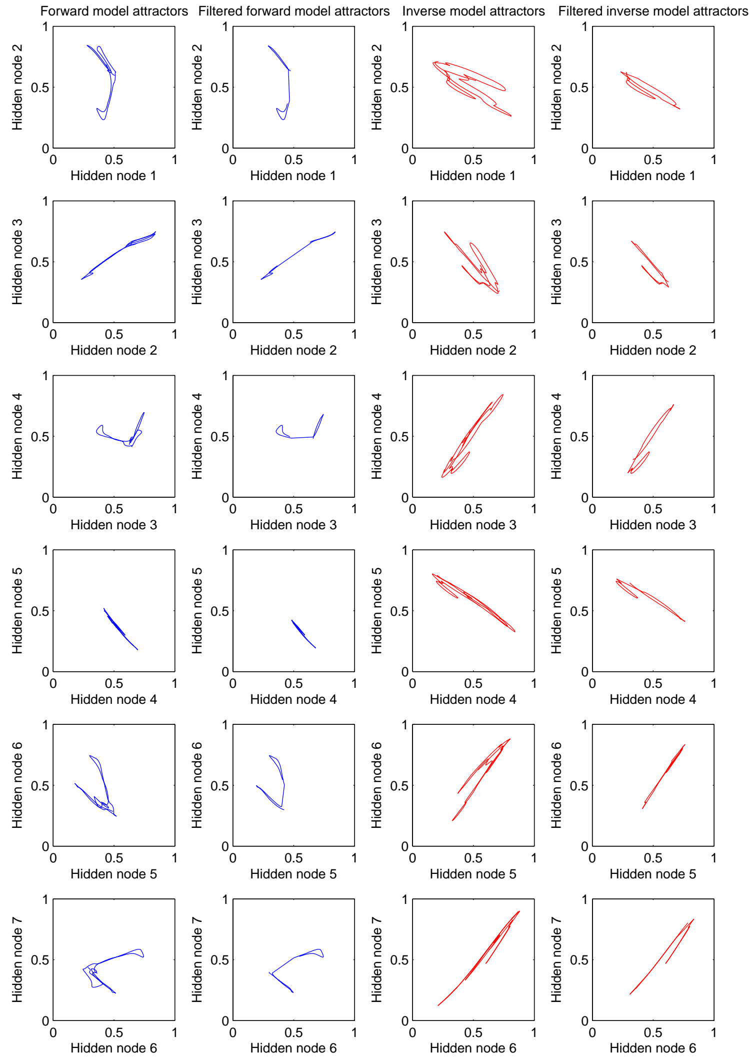


Figure 7.22: (7.1.5) Attractor plots, module 2, the cheerleader. The forward model plots are more consistent than the inverse model plots, which are a bit more fluctuating. This could explain why the feedback error motor command is bigger when this module is having control over the output.

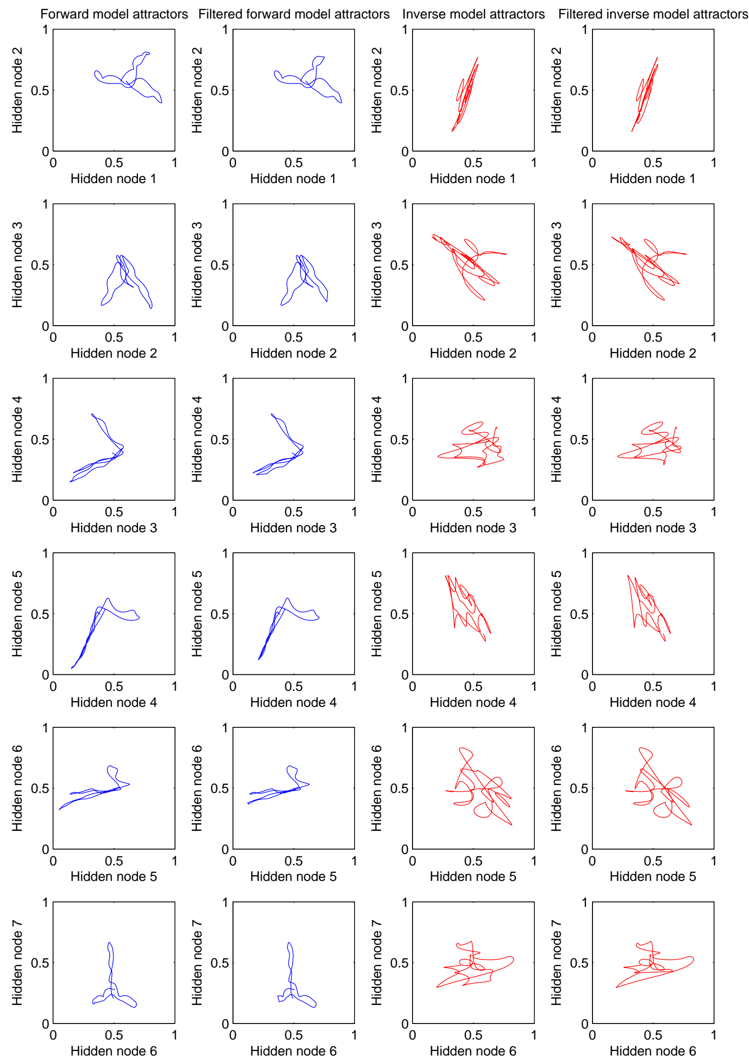


Figure 7.23: (7.1.5) Attractor plots, module 3, the cheerleader. The forward models display interesting patterns, reminiscent of a boomerang, showing a memory that seems to be quite stable.

7.4 YMCA

Name of MatLab file	`human_breve_matlab_YMCA4DOF.m`
Name of breve file	`Tiny Dancer - YMCA4DOF.tz`
Degrees of freedom in breve	4
Length of desired state	144
Number of paired inverse/forward models	4
Number of epochs trained	12000
Feedback error gain K	3
Output gain M	3
σ in the likelihood function	0.10
Error proportion P	8
Size of forward/inverse neural network	8-4-4
Size of responsibility predictor network	4-4-1
Learning rate δ	0.01
Active joints	Right shoulder X joint angle, Right elbow X joint angle, Left shoulder X joint angle, Left elbow X joint angle

7.4.1 The learning of each of the modules

Figure 7.24 shows that every module has a decreasing error curve, not just a low error * λ curve. Especially interesting is the curve of the third module, where the error curve starts to decrease after epoch 5000. This indicates that the system is capable of retuning itself even after it has been trained for a long time. The performance of the system as a whole does not improve or worsen remarkably during this period, so clearly the first module manages to learn more without affecting the total performance.

The experiment has four modules, making the competition harder between each of them. In addition, the target state is gathered from tracking data of me dancing. The movement has more noise and is more complex than the movements in the previous experiments, which were hard-coded by me. Figure 7.25 shows how the different modules are more equal in terms of time controlling the behaviour of the robot. The second module is the one dominating, but the first and fourth are not far behind, and they are active for almost the same amount of time as each other. The third module is also active in the first part of the movement. For the YMCA movement, all modules play an important part in controlling the robot.

7.4.2 Switching between controlling modules

After some initial changing between which module is the winning module, the number of transitions is fairly stable after epoch 1000, with some minor fluctuations around epoch 5500 and 9500 (figure 7.24). At epoch 2000 and 9000 there is the same amount of transitions, but figure 7.27 shows that the separation of control between the winning modules is quite different from epoch 9000 (figure 7.28). Although they share some similarities (such as a very dominant second module and a control period of the fourth module that is almost the

same), there are some noteable differences. Epoch 9000 has more clearly defined winning modules than at epoch 2000, i.e. one module is close to having the entire control during a movement. In epoch 2000, control is more shared between modules, such as the period at timesteps 25 - 35, where the second and first module share responsibility of controlling the robot. Another similarity is that towards the end of the movement, there is no clear winning module, but responsibility is shared between the winning modules.

It is also often the case that there are only two modules that share the control at a certain timestep. If we again look to epoch 2000 and epoch 9000, this can be seen in figures 7.29 and 7.30. The ideal would be that one module was in control during a certain period of time, but the figures show how the modules participate in controlling the robot, and that the responsibility is rather often shared between two modules.

In terms of performance, epoch 2000 is not that bad compared to epoch 9000 (which as a sudden increase in error, as can be seen in figure 7.24). The total performance plot of epoch 2000 is shown in figure 7.31, epoch 9000 in figure 7.32. The latter figure (epoch 9000) shows a bad error in the movement of both the left and right elbow at about the 70th timestep, not present at epoch 2000. This demonstrates how the multiple paired models structure tries out new solutions, but occasionally there is a mismatch between the actual state and the desired state, and worse performance is the result.

7.4.3 Attractor plots

The attractor plots of the first module can be seen in figure 7.33. The first and third plots of the forward model are somewhat regular, indicating a stable memory. The second plot is more irregular. The attractor plots of the inverse models are also quite irregular. This probably explains why the first module only controls the robot for some short spikes of time (although it does have a rather high responsibility signal towards the end of the movement).

Figure 7.34 shows the attractor plots of the second module. These attractor plots are the most chaotic of all the modules, which is intriguing, given that the second module dominates the robot's motion for long periods of time. The same was observed in the cheerleader experiment; the module that was in control most of the time had the most chaotic attractors. Perhaps this is an indication of the complexity of storing big chunks of the movement to be imitated, whereas storing simple changes requires less memory and therefore yields more simple attractors. The second module is in control for two periods of time, where the latter is very long as well. By looking at figure 7.25, it can be seen how the second module controls the robot during the entire third movement (the "C") and half of the fourth movement (the "A"). It could be the attractors are complex since they represent these two different movements, who are by themselves fairly complex. There is a substantial difference between the filtered and unfiltered attractors, indicating that the second module is not very specialized, but actually wants to try to control other parts of the movement as well. If the multiple paired models structure was trained for more epochs, perhaps the second module would have managed to increase its period of control.

The third module has the most stable attractors of all the modules (figure 7.35). Both the filtered and unfiltered plots are almost identical, suggesting that the module is specialized for a certain part of movement. Indeed, figure 7.14

shows that the third module is most active during the beginning and end of the total movement. The beginning of the “Y” and the ending of the “A” are very similar, where the robot is basically lowering or raising the arms in unison.

Figure 7.36 depicts the attractor plots of the fourth module. The inverse model attractors are very different from that of the other movements, indicating that the module knows a specific type of movement. Figures 7.25 and 7.26 show that the fourth module is most active during the end of the first movement (the “Y”) and during the entire second movement (the “M”). The “M” is very different from the other letters, since the elbow joint angles vary a lot. In the other movements, it is the shoulder joints that are responsible for creating the corresponding letter. That could be the reason why the inverse model attractors are so peculiar for the fourth module.

7.4.4 Was the goal met?

The goal of the experiment was to demonstrate the architecture’s capability of dealing with imitation of a human being, see section 6.3.2. The system performed well at the imitative act as a whole, but the separation of responsibilities between the modules did not happen as intended (as has been demonstrated in the previous experiments as well). Nevertheless, it is clear that each module does play an important part of the total control (as was displayed in the previous experiments as well). The results related to the hypotheses:

Hypothesis 1: In this experiment, the self-organization was the most complex compared to the previous experiments. The analysis shows that all the modules play important parts in controlling the robot, but the self-organization did not happen as intended (i.e. according to the context information).

Hypothesis 2: As in the previous two experiments, the context information did not help the self-organization. Any detection of movement boundaries was done without the use of the context information. This could be why there are more small periods of control between the modules; the complex movements can be separated in more ways due to their complexity.

Hypothesis 3: As in the previous experiments, the relationship between the context information and the movements were not discovered, due to the responsibility predictor only outputting high values during an epoch.

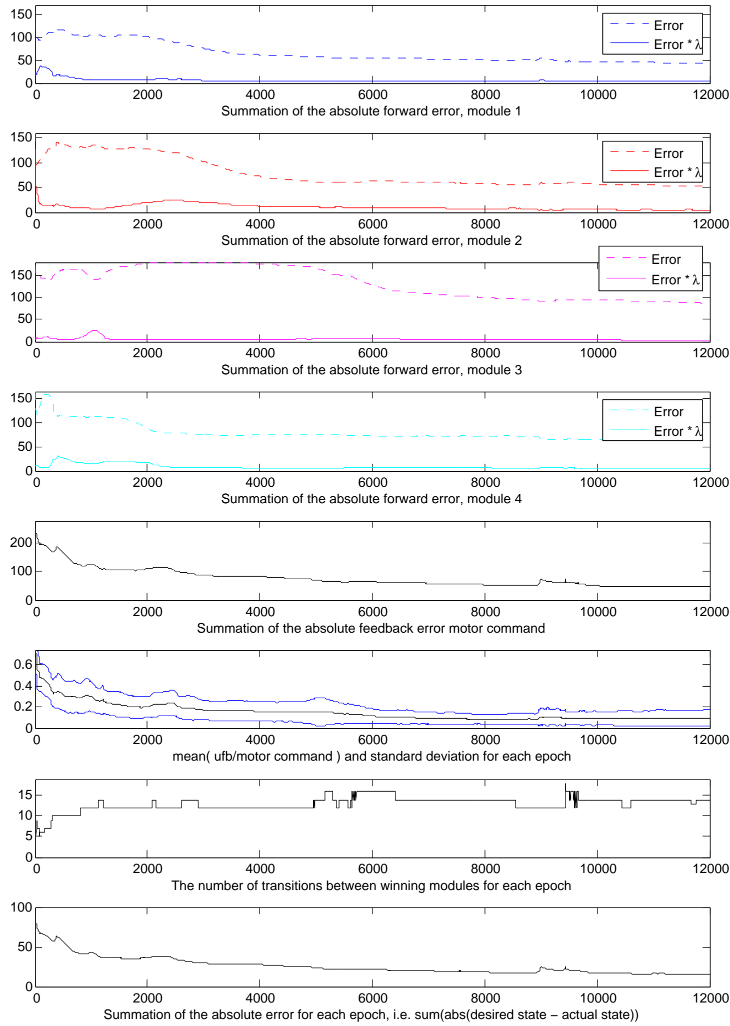


Figure 7.24: (7.1.1) The entire training run, the YMCA. The x-axis shows the number of epochs. The prediction errors of the first (blue), second (red), third (magenta) and fourth (green) module can be seen in the four upper plots. Plots 5-8 are the same as plots 3-6 in figure 7.1. All the error plots are steadily decreasing throughout the training period.

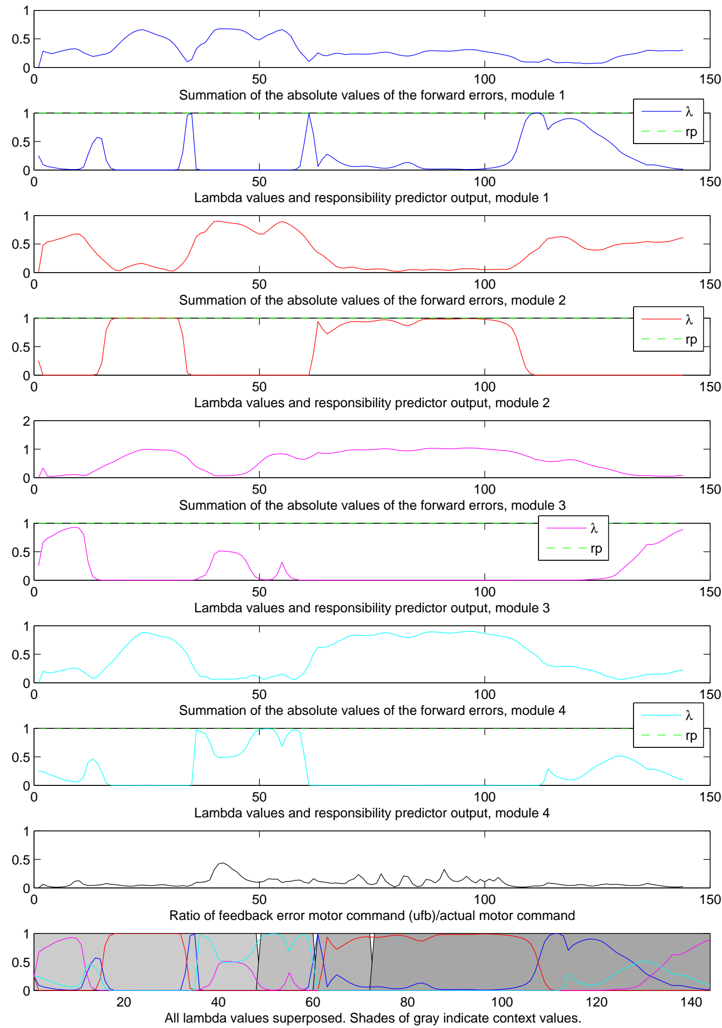


Figure 7.25: (7.1.2) Performance of the last epoch, the YMCA. The x-axis shows the timesteps. The plots show the same as in figure 7.14, with the addition of the third (magenta) and fourth (green) module. Notice how the λ plots superposed are a lot more complex than for the other experiments, i.e. there are a lot more transitions between the winning modules.

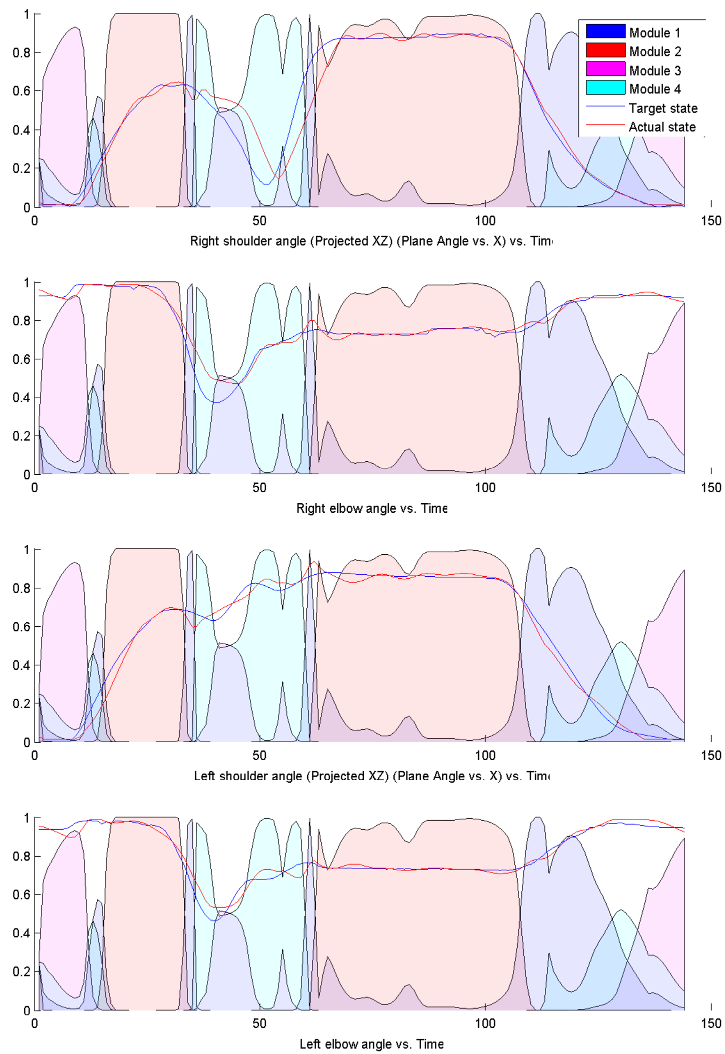


Figure 7.26: (7.1.4) The λ and target/actual trajectory of the YMCA motion at the last epoch. The performance is quite good, even though there are a lot of changes regarding which module is in control.

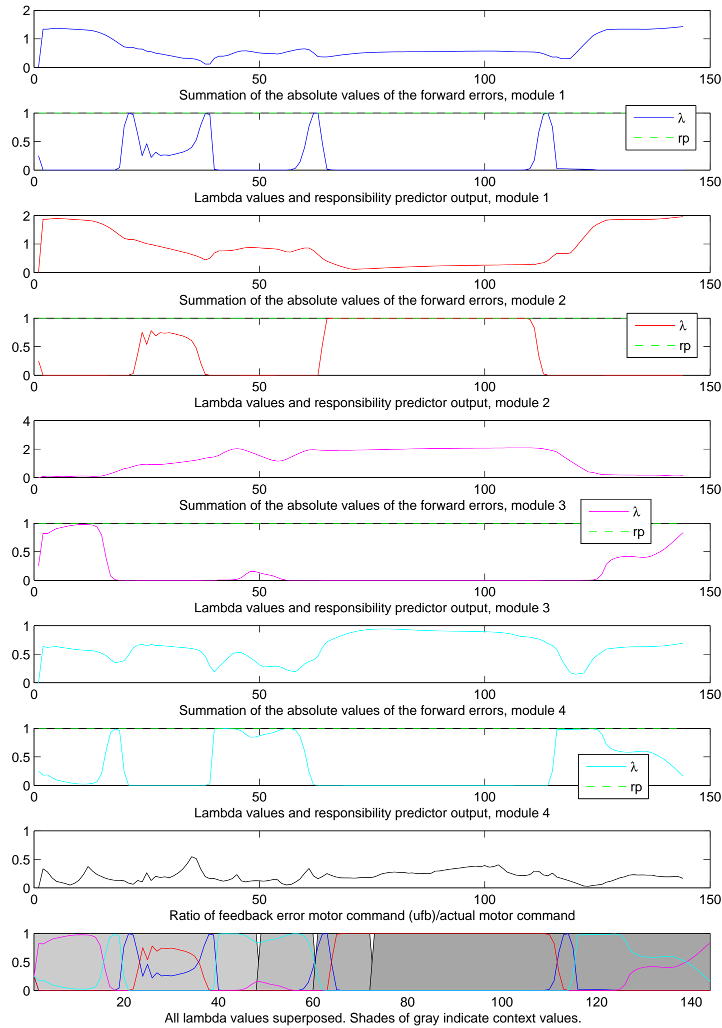


Figure 7.27: (7.1.2) Performance of epoch 2000, the YMCA. The performance is quite good, even at such an early stage of the training period. However, the ratio feedback motor error commands to the total motor command is quite high, indicating that the multiple paired models architecture did indeed need some help to meet the target states.

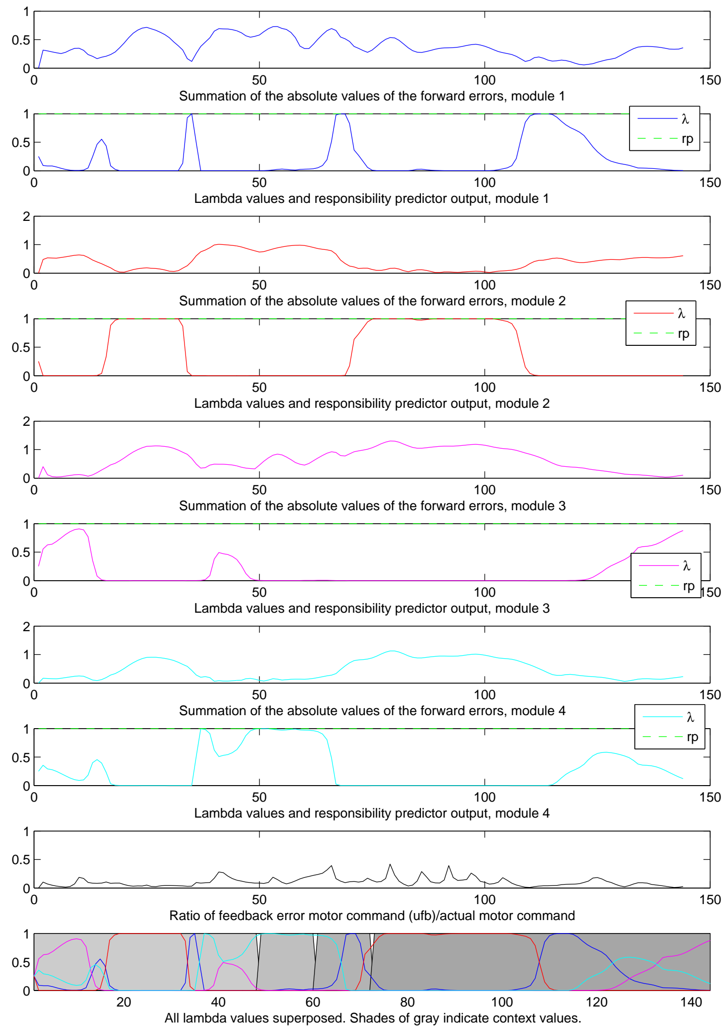


Figure 7.28: (7.1.2) Performance of epoch 9000, the YMCA. The separation of control is quite different from that in epoch 2000 (figure 7.27), which shows the dynamic nature of the multiple paired models architecture.

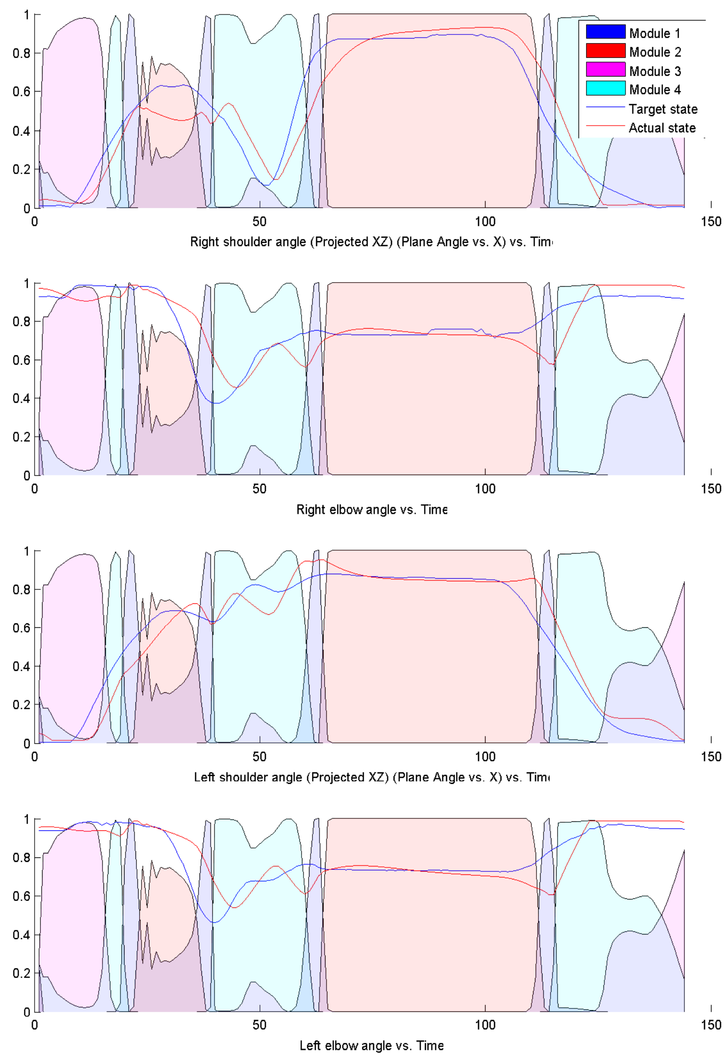


Figure 7.29: (7.1.4) Desired and actual state with λ plots, epoch 2000, the YMCA. The plots show that the modules participate in controlling the robot, i.e. there are timesteps where two modules play an almost equal part in controlling the robot.

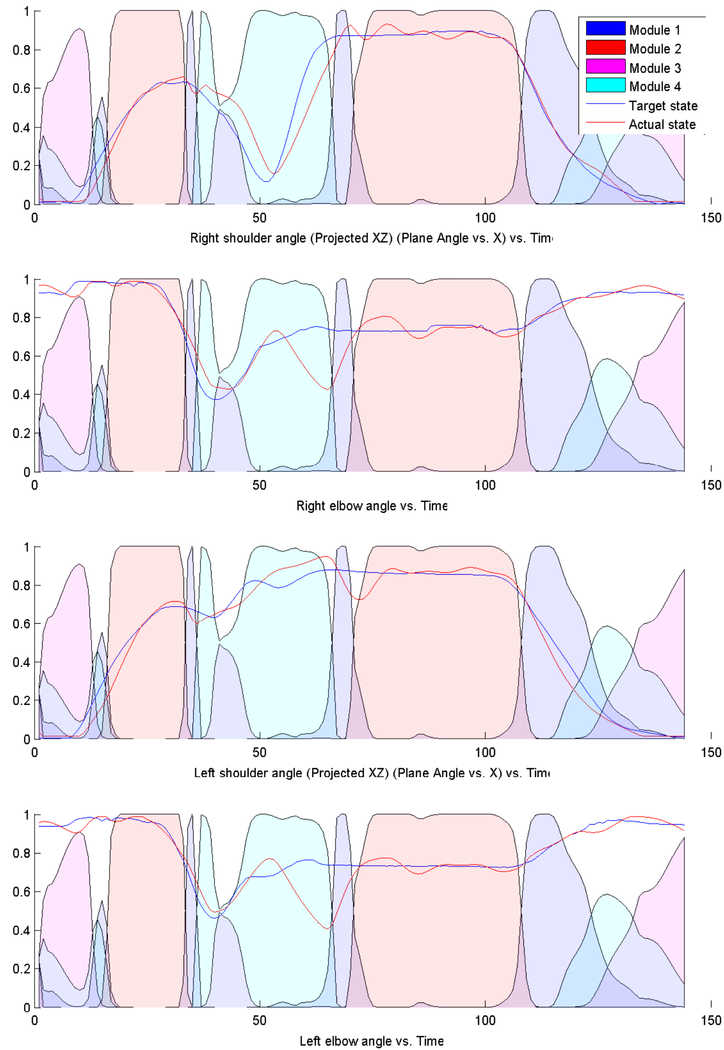


Figure 7.30: (7.1.4) Desired and actual state with λ plots, epoch 9000, the YMCA. The plots show that the modules participate in controlling the robot (as was also shown for epoch 2000, see figure 7.29), i.e. there are timesteps where two modules play an almost equal part in controlling the robot. Notice the difference between this plot and the plot for epoch 2000. This shows that the multiple paired models architecture is dynamic and changes over time.

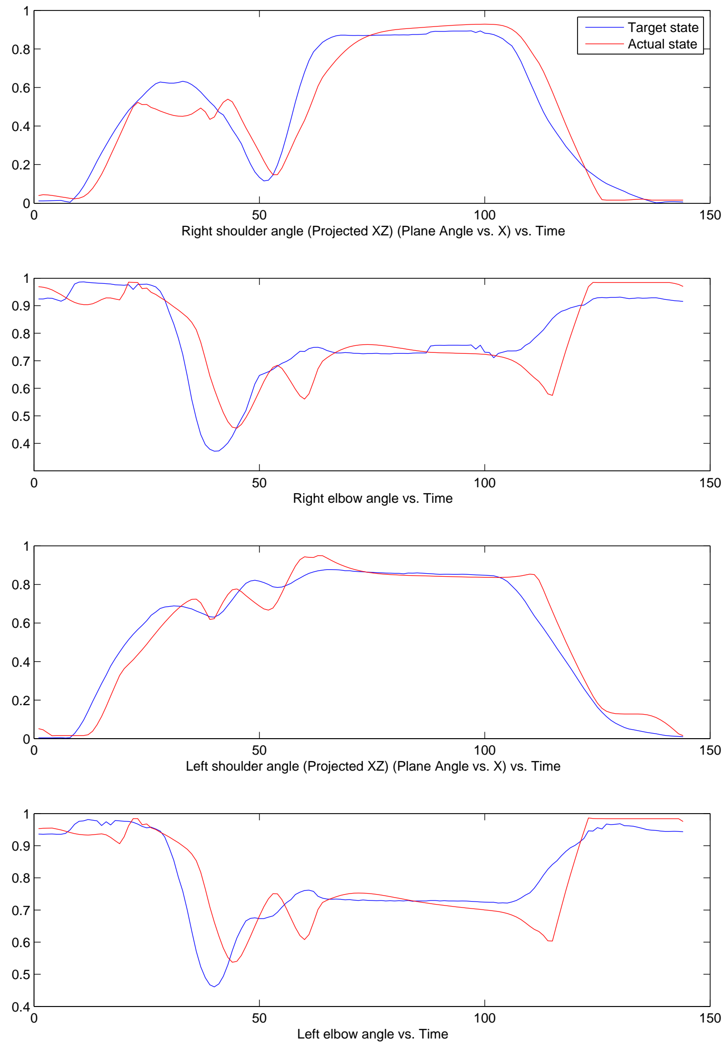


Figure 7.31: (7.1.3) Desired and actual trajectory, epoch 2000, the YMCA. The performance is quite good, even this early on in the training period. Compare it to epoch 9000 (figure 7.32), which suddenly has a huge error around timestep 65, but the *rest* of the motion is quite close to the desired state. This shows that the multiple paired models architecture adapts quickly, but still tries out different solutions as the training progresses.

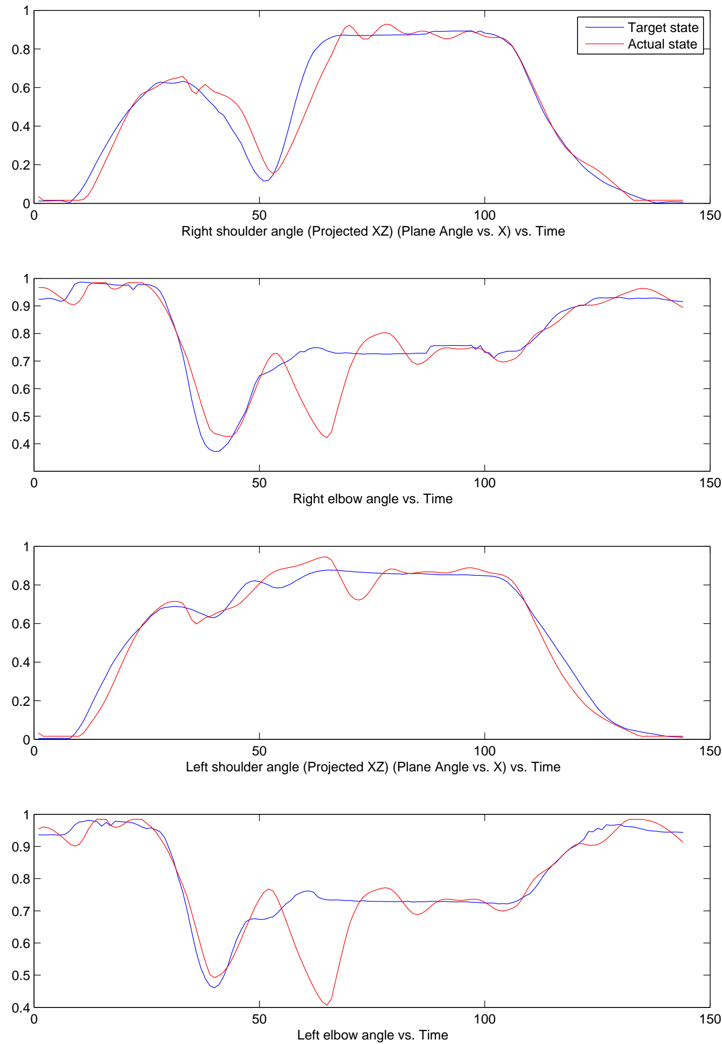


Figure 7.32: (7.1.3) Desired and actual trajectory, epoch 9000, the YMCA. At epoch 2000 (figure 7.31) the performance was already quite good. Here, a huge error has occured around timestep 65. However, since the rest of the motion is quite close to the desired trajectory, this shows that the architecture tries to find different solutions, but sometimes misses.

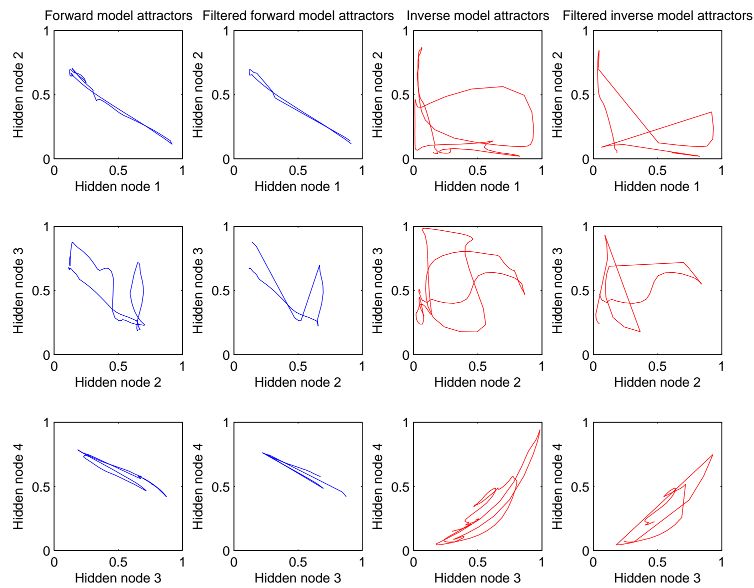


Figure 7.33: (7.1.5) Attractor plots, module 1, the YMCA. There are more stable patterns for the forward model than the inverse, which might explain whey the module controls the robot only for a short period of time.

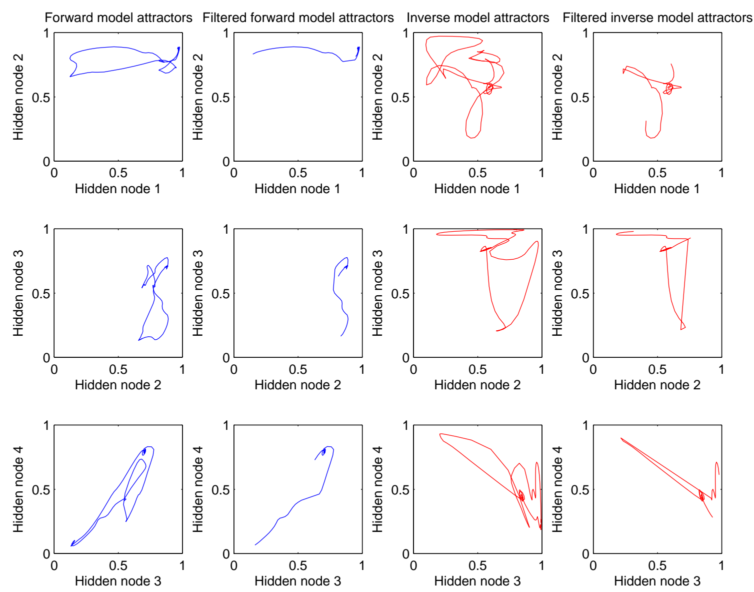


Figure 7.34: (7.1.5) Attractor plots, module 2, the YMCA. The second module has the most chaotic attractor plots. It dominates the motion of the robot for long periods of time; the complexity of the attractor plots might represent the complexity of the motion it controls.

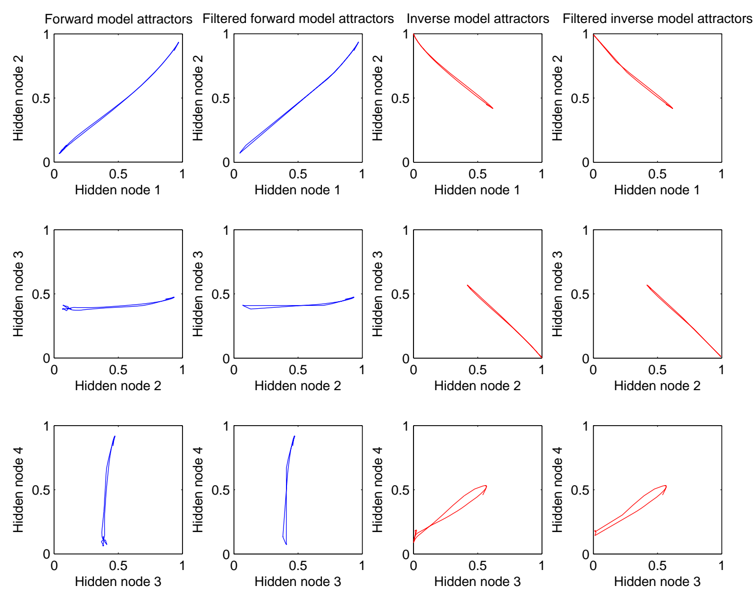


Figure 7.35: (7.1.5) Attractor plots, module 3, the YMCA. The third module has the most stable attractors compared to the other modules. This might explain why it is in control in both the beginning of the “Y” and the ending of the “A”; these movements are very similar (although in the opposite direction).

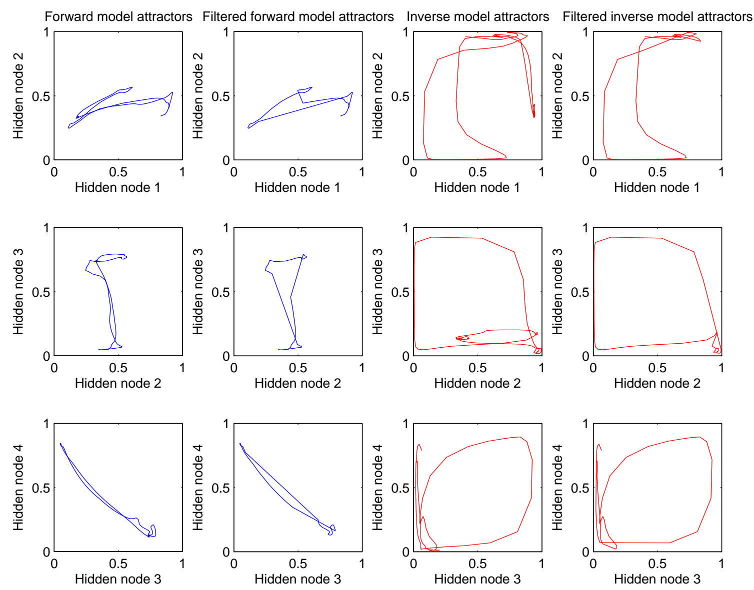


Figure 7.36: (7.1.5) Attractor plots, module 4, the YMCA. The inverse model attractors are particular for this module, indicating that the inverse model knows a specific type of movement. Indeed, it is in control during the "M" movement, which is different from all the other movements, since it requires moving the elbow joints.

Chapter 8

Conclusion

The most important lesson I have learned when designing a self-organizing system is that it will most likely *not* self-organize the way I intended it to self-organize. In fact, it might seem a bit contradictory to design a self-organizing system and imposing certain intentions and restrictions to it. After all, it is supposed to self-organize, and the designer should make as few choices as possible, since the choices made will most likely inhibit the emerging solution. However, when doing an experiment it is required that some limitations and choices are made, given the limited resources available. Ideally, I would like the number of modules to evolve, i.e. not having to define beforehand the number of modules in the architecture. A genetic algorithm would be very well suited for this, where each individual would hold parameters such as number of modules, learning rate, size of the networks and so on. Each individual would then be trained on the imitation task at hand, and the results compared, offspring created and another generation could be run. However, given that the training period of the YMCA experiment took 11 hours to complete on my iMac G5, it is clear that a genetic algorithm solution is not feasible with the current computing resources available, since a genetic algorithm is typically run for thousands of generations.

Instead, I have to make choices at the design stage of an experiment. I have made the motions that are to be imitated, and I have explicitly thought of a way to separate the movements into discrete modules. The idea of the context information was that a certain movement would be made discrete, and that a module would understand that when one of the input values of the context information was high, there is something unique with the corresponding movement. A module would then learn this specific movement. The number of modules therefore always corresponded to the number of discrete movements. The results have shown that it is never the case that a module has learned a specific movement as intended. Instead, the λ values show that the modules are in control during periods of time where they overlap between two movements (i.e. the first module is in control during both the first and second movement, as was shown for the two Ks), or that they are in control for very short periods of time.

However, even though the responsibility signals of each of the modules do not correspond directly to the context information I provided (in fact, the context information does not contribute to determining the responsibility signal at all, which will be discussed shortly), the self-organization was not completely

without correlation to the different movements. The plots show that the different modules are often responsible for making a switch in the behaviour, such as changing the direction of the movement of an arm. For the first two experiments, the results showed that there was one major dominating controlling module, whereas the other modules take control from time to time to change the behaviour, such as changing the direction of the arm. The results showed that the different modules would not exactly self-organize as intended, but they would share responsibility when it came to controlling the robot. As discussed earlier, most of the changes of the module in charge happened at periods of time where crucial changes needed to be made, such as changing the direction of the arm. This shows that the multiple paired models architecture is indeed capable of detecting when important *changes* in the desired state occur. The multiple paired models architecture self-organizes so that it will find these changes itself, even without the context information.

The context values were used because the were part of the original design by Wolpert. However, when looking at the plots of the λ values and the output of the responsibility predictor, it is clear that the context information actually does not help the multiple paired models architecture at all. All the plots show that the output of the responsibility predictor (that was thought of as the *prior probability* that a given module was suitable to control the robot) remained high for the entire epoch. In other words, it might just as well not have been there. Although the proper training was performed as written in [62], the responsibility predictor did not perform as intended at all. This is without doubt also one of the reasons why the responsibility signals of the modules do not match the context information provided by me. Recall from the system architecture in figure 4.1 that the context information is only given to the responsibility predictor; it is the responsibility predictor that must convey the information received from the context signals.

An easy solution would be to train the responsibility before training the rest of the multiple paired models architecture, but that lessens the degree of self-organization of the multiple paired models architecture. Another solution must be found for making the responsibility predictor work as intended[1]. Currently, it does not play a part in determining the responsibility signal of a given module. But even though the context information is not used within the system, it manages to self-organize and also discover changes in the desired trajectory which it uses to allocate modules to specific movements, based on the predictive capabilities of the forward model.

The overall performance of the system as a whole is very good for each experiment. The plots of desired trajectory versus actual trajectory show that the multiple paired models structure has found a good solution to the behaviour it was supposed to imitate. The use of the context information did not work as intended, but still the multiple paired models structure managed to coordinate

[1] An evident question is why I have not tried to implement a solution in this Master's thesis. This was discovered during the stage of analyzing the data gathered from breve, after all the simulations had been run. When I did the experiments, I did of course plot the performance of the network in order to tune the parameters. However, it did not occur to me to plot the output of the responsibility predictor, which I was certain would be equal to the λ signal. When I discovered the malfunctioning of the responsibility predictor, I realized I could not spend more time trying to find a solution; I simply had to stop programming in order to finish the Master's thesis. Since I will be continuing at the Forskerskole, I will have the opportunity to work more on this problem.

several modules so that they work seamlessly.

So how do the results compare to the working hypotheses (see section 1.3)?

Hypothesis 1: The multiple paired models architecture did self-organize the control of different movements to different modules, but the modules did not control one movement exclusively in any of the experiments, which was the intention.

Hypothesis 2: As the results have shown, the context information did not help the multiple paired models architecture to self-organize. The detection of movement boundaries had to be done by the architecture without the context information. Nevertheless, the architecture seems to have detected important changes in the movements, such as changing the direction of an arm. The inverse/forward coupling seem to have overcome, to some extent, the problem of the malfunctioning responsibility predictor.

Hypothesis 3: The responsibility predictor did not function as intended (as has been mentioned several times), so the multiple paired models architecture did not discover any relationship between the context information and the movements.

The response to the research question posed in section 1.2 will be a positive answer, however there is still work to do to make architecture behave as intended. The hypotheses stated in section 1.3 were not confirmed, since the architecture did not function exactly as intended. However, it is far from being a failure. The overall results were quite good, and the process has been very educational. However, a solution to make the responsibility predictor behave as intended would make better grounds to examine how well the architecture works. Why this desire to make the responsibility predictor work as intended, when the overall system performance itself was quite satisfactory? Besides the obvious (i.e. it was supposed to work like that) there is a link to the mirror neurons discussed in section 2.2. The responsibility predictor activity can be seen as the mirror neuron activity that have been found in monkeys and humans. The mirror neurons coded for a specific action to be done, but on a higher level, i.e. mirror neuron activity did not directly produce motor commands, instead they might code for the abstraction of a motor program. Now look at how the responsibility predictor is intended to function. Based on some context information, it is supposed to output the suitability of the module to control the robot. Since each module represents a certain movement (at least ideally, although it did not work like that in my implementation), the responsibility predictor can be viewed as coding for this movement at a higher level, like the mirror neurons do. The mirror neurons do not code the motor command directly, neither does the responsibility predictor. If we see the mirror neuron activity as a gating mechanism for a certain behaviour (which is a natural consequence if we think that it codes for a motor command, only at a higher level), the link between mirror neuron activity and responsibility predictor activity becomes even stronger.

Having both the benefits of having a multiple paired models architecture (see section 4.1.1) and the mirror neuron activity seem like a good starting point for building an architecture that could be used for imitation learning. I am currently in the first part of the Forskerskole at IDI, and will continue with my PhD after the completion of the Master's thesis. The work I have done so far will make a good starting point for future work.

Chapter 9

Future work

As work with the system has progressed, ideas for future work have appeared. At some point I had to stop implementing and running simulations in order to finish and write my Master's thesis. Since I am part of the Forskerskole at IDI and will continue doing my PhD after completion of my Master's thesis, I will have the opportunity to pursue these ideas.

- Learning a complete forward model of the breve simulator, and use the forward model instead of the actual simulator when training the inverse and forward models. This would mean that the imitator would know how its own body behaves given a specific motor command, but still it would not know how to imitate certain movements. This would allow for a significant speed in training, since the multiple paired models architecture would not need to communicate with the breve simulator during training.

- Training different modules at specific motor primitives, and then putting them into the architecture. By training each of the modules on one movement sequence would enable me to look at how the architecture will work when trying to organize the different movement primitives. This is essentially the same as in Demiris' work, where the movement primitives are already specified, it is the coordination that is investigated. The two approaches can then be compared to see which is a) more computationally effective and b) which works best.

- Making the responsibility signals more detailed, i.e. they could be governing one specific output motor neuron. The forward model could predict the next state, and the evaluation of the prediction could be based on the accuracy of each output neuron of the forward model. If some neurons were very correct in predicting the next state, the corresponding motor neurons (i.e. based on indices only) could be given a bigger portion of the responsibility signal. Without having implemented such a scheme, it seems to me like this approach would allow superposition of specific motor skills from different modules. At the current implementation, the module knows a specific movement (not entirely equal to the boundaries I have set, but still a movement that will generate motor commands that will lead put the robot closer to the desired state). If the multiple paired models architecture was to imitate an action that was a composite of different

movements (i.e. moving your right arm and your left foot, with one module for each) this approach might work to split the responsibility between the two modules.

A variation on this could be assign special motor capabilities to specific modules. Say one module controls only the left arm, and another module controls the right leg. The responsibility signals could then gate based on the different motor capabilities, allowing for superposition of movements. The different output regions of each module could be seen as a mask covering some part of the motor output space. I.e. if there are 16 output neurons, the module controlling the left arm will control the first four neurons, the right arm will be neurons 5-8 and so on.

- The different parameters of the system could be determined using a genetic algorithm instead of trying to fine-tune them by hand. It is a classical optimization problem, with an easy fitness function (e.g. low ratio between the feedback motor command to the actual motor command ratio and target state and current state being the same), however the simulator used for the current experiments are not very suited for using genetic algorithms. Since one run will take typically several hours, running a population of one hundred individuals for thousands of generations is clearly not feasible. However, if it was possible to use another simulator that was faster this might be a good approach.

- As discussed in section 8, the current implementation of training the responsibility predictor does not work. Haruno hints that they are trained beforehand in [27], where he says that “A responsibility predictor estimates the responsibility before movement onset using sensory contextual cues y_t and is trained to approximate the final responsibility estimate.” This hints at the possibility that the responsibility predictor is trained beforehand, but as I mentioned earlier, this would reduce the degree of self-organization of the system. Another solution could be to train the responsibility predictor more intensively (i.e. five times when the other networks are trained once). Perhaps the context sensor input could be coded differently, to make it easier for the responsibility predictor to realize the relationship it should learn. Nevertheless, finding a way to make the multiple paired models architecture realize the relationship between the context information and the discrete movements will be what I first focus on after the completion of this thesis.

- Learning several motions, and investigate whether the multiple paired models architecture will overcome problems related to catastrophic forgetting [1, 2, 3]. In addition, investigate how reshuffling the movements would affect the multiple paired models, both during training and during recognition of a motion.

- Investigating the performance of the network when the number of modules is increased and decreased. See if the added modules will lead to totally redundant modules, or if they will all contribute in some manner. Investigate to see what will happen if there are fewer modules than intended; will they be able to self-organize and perform as good as the multiple paired models architecture with the intended number of modules?

- Using the Pro Reflex system to do imitation experiments with human test subjects in real-time. It is possible to write plug-ins to the Pro Reflex software. The plug-in could then communicate with MatLab, which would hold the multiple paired models architecture. MatLab would communicate with breve, as done in the current implementation. The humanoid robot in the breve simulator could then be displayed via a video projector to the test subject. The test subject (equipped with fluorescent balls for easy tracking by the Pro Reflex system) could then dance, and subsequently feed the desired state to the multiple paired models architecture, which would control the breve simulator.

  A lot of different interesting scenarios could be investigated. One could be to teach the multiple paired models architecture a repertoire of dance movements, and then make the test subjects dance without knowing the repertoire (as Ito and Tani did with arm movements in [32]). The situation also allows studying the joint attention in the human-robot interaction, as studied in [31, 59].

  Another interesting scenario would be to use the interaction to teach the robot new movements. If a faster on-line learning scheme was devised, this could be a fun demonstration of the capabilities of the multiple paired models architecture. If the learning would separate itself into different modules, a vast array of movements could (in theory) be learned.

- Experimenting with different neural network architectures and also the dimension of the neural networks. For this thesis, the same neural network architecture (the recurrent neural network) was used. The dimensions of the networks were different according to the degrees of freedom, but they still followed the same formula; i.e. that the number of nodes in the hidden layer equaled the number of nodes in the output layer. It would be interesting to try out different number of nodes in the hidden layer, as well as trying out feedforward networks and perhaps even the counter-propagation network[1]. In addition, different training regimes could be used, i.e. *batch learning* where the updating of the weights is done *after* one pass through the epoch.

[1] The counter-propagation network [28, 29] is a simple winner-take-all network that works as a look-up table. It is trained very fast compared to feedforward networks, but lack some of the generalization capabilities of feedforward networks.

Bibliography

[1] Bernard Ans and Stéphane Rousset. Neural networks with a self-refreshing memory: knowledge transfer in sequential learning tasks without catastrophic forgetting. *Connection Science*, 12(1):1–19, 2000.

[2] Bernard Ans, Stéphane Rousset, Robert M. French, and Serban Musca. Preventing catastrophic interference in multiple-sequence learning using coupled reverberating networks. In *Proceedings of the 24th Annual Meeting of the Cognitive Science Society*, pages 71–76, 2002.

[3] Bernard Ans, Stéphane Rousset, Robert M. French, and Serban Musca. Self-refreshing memory in artificial neural networks: learning temporal structures without catastrophic forgetting. *Connection Science*, 16(2):71–99, June 2004.

[4] Michael Arbib. *Imitation in animals and artifacts*, chapter The Mirror System, Imitation, and the Evolution of Language, pages 229–280. MIT Press, Cambridge, 2002.

[5] Aude Billard and Maja J. Matarić. Learning human arm movements by imitation: evaluation of a biologically inspired connectionist architecture. *Robotics and Autonomous Systems*, 941:1–16, 2001.

[6] Cynthia Breazeal and Brian Scassellati. Robots that imitate humans. *Trends in Cogntive Sciences*, 6(11):481–487, 2002.

[7] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE journal of Robotics and Automation*, 2(1):14–23, 1986.

[8] Angelo Cangelosi and Thomas Riga. An embodied model for sensorimotor grounding and grounding transfer: Experiments with epigenetic robots. *Cognitive Science*, (in press).

[9] Angelo Cangelosi, Thomas Riga, Barbara Giolito, and Davide Marocco. Language emergence and grounding in sensorimotor agents and robots. In *First International Workshop on Emergence and Evolution of Linguistic Communication*, pages 3–8, Kanazawa, Japan, 2004.

[10] Thierry Chaminade, Andrew N. Meltzoff, and Jean Decety. Does the end justify the means? A PET exploration of the mechanisms involved in human imitation. *NeuroImage*, 15:318–328, 2002.

[11] Mirella Dapretto, Mari S. Davies, Jennifer H. Pfeifer, Ashley A. Scott, Marian Sigman, Susan Y. Bookheimer, and Marco Iacoboni. Understanding emotions in others: mirror neuron dysfunction in children with autism spectrum disorders. *Nature Neuroscience*, 9:28–30, 2005.

[12] Anthony Dearden and Yiannis Demiris. Learning forward models for robots. In *Proceedings of IJCAI*, pages 1440–1445, 2005.

[13] Yiannis Demiris and Anthony Dearden. From motor babbling to hierarchical learning by imitation: a robot developmental pathway. In *EPIROB*, pages 31–37, 2005.

[14] Yiannis Demiris and Gillian Hayes. *Imitation in animals and artifacts*, chapter Imitation as a dual-route process featuring predictive and learning components: a biologically-plausible computational model, pages 327–361. MIT Press, Cambridge, 2002.

[15] Yiannis Demiris and Bassam Khadhouri. Hierarchical attentive multiple models for execution and recognition of actions. *Robotics and Autonomous Systems*, (to appear).

[16] Yiannis Demiris and Gavin Simmons. Perceiving the unusual: temporal properties of hierarchical motor representations for action perception. *Neural Networks*, 2006.

[17] Michel Desmurget and Claude Prablanc. Postural Control of Three-Dimensional Prehension Movements. *J Neurophysiol*, 77(1):452–464, 1997.

[18] K. Doya. What are the computations of the cerebellum, the basal ganglia and the cerebral cortex? *Neural Networks*, 12:961–974, 1999.

[19] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[20] Andrew H. Fagg and Michael A. Arbib. Modeling parietal-premotor interactions in primate control of grasping. *Neural Networks*, 11:1277–1303, 1998.

[21] Vittorio Gallese and Alvin Goldman. Mirror neurons and the simulation theory of mind-reading. *Trends in Cognitive Sciences*, 2(12), 1998.

[22] P. Gaussier, S. Moga, J. P. Banquet, and M. Quoy. From perception-action loops to imitation processes: A bottom-up approach of learning by imitation. *Applied Artificial Intelligence*, 1(7):701–727, 1998.

[23] Michael S. Gazzaniga, Richard B. Ivry, and George R. Mangun. *Cognitive neuroscience: the biology of the mind*. Norton, New York, c2002.

[24] S.T. Grafton, M. A. Arbib, L. Fadiga, and G. Rizzolatti. Localization of grasp representations in humans by positron emission tomography. 2. observation compared with imagination. *Experimental Brain Research*, 112(1):103–111, November 1996.

[25] Stevan Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42:335–346, 1990.

[26] Christopher M. Harris and Daniel M. Wolpert. Signal-dependent noise determines motor planning. *Nature*, 394:780–784, 1998.

[27] Masahiko Haruno, Daniel M. Wolpert, and Mitsuo Kawato. MOSAIC model for sensorimotor learning and control. *Neural Comp.*, 13(10):2201–2220, 2001.

[28] R. Hecht-Nielsen. Counterpropagation networks. *Applied Optics*, 26:4979–4984, 1987.

[29] Robert Hecht-Nielsen. Applications of counterpropagation networks. *Neural Networks*, 1:131–139, 1988.

[30] John H. Holland. *Adaptation in Neural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[31] Masatio Ito and Jun Tani. Joint attention between a humanoid robot and users in imitation game. In *Proc. 3rd. Int. Conf. on Development and Learning (ICDL '04)*, 2004.

[32] Masato Ito and Jun Tani. On-line imitative interaction with a humanoid robot using a dynamic recurrent neural network model of a mirror system. *Adaptive Behavior*, 12(2):93–115, 2004.

[33] Robert A. Jacobs, Micheal I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3:79–87, 1991.

[34] Dan-Anders Jihrenhed, Germund Hesslow, and Tom Ziemke. Exploring internal simulation of perception in mobile robots. In Arras, Baerveldt, Balkenius, Burgard, and Siegwart, editors, *2001 Fourth European Workshop on Advanced Mobile Robotics*, volume 86, pages 107–113, Lund, Sweden, 2001. Lund University Cognitive Studies.

[35] Michael I. Jordan and Robert A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6:181–214, 1994.

[36] Michael I. Jordan and David E. Rumelhart. Forward models: Supervised learning with a distal teacher. *Cognitive Science*, 16:307–354, 1992.

[37] Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell. *Principles of neural science*. McGraw-Hill, New York, 2000.

[38] Mitsuo Kawato. Feedback-error-learning neural network for supervised motor learning. In R. Eckmiller, editor, *Advanced neural computers*, pages 365–372, 1990.

[39] Mitsuo Kawato. Internal models for motor control and trajectory planning. *Current Opinion in Neurobiology*, 9:718–727, 1999.

[40] Rodolfo R. Llinás. *I of the vortex: from neurons to self.* MIT Press, Cambridge, Mass., 2001.

[41] Maja J. Matarić. Getting humanoids to move and imitate. *IEEE Intelligent Systems*, pages 18–24, July 2000.

[42] Maja J. Matarić. *Imitation in animals and artifacts*, chapter Sensory-Motor Primitives as a Basis for Learning by Imitation: Linking Perception to Action and Biology to Robotics, pages 392–422. MIT Press, Cambridge, 2002.

[43] Kishan Mehrotra, Chilukuri K. Mohan, and Sanjay Ranka. *Elements of artificial neural networks*. MIT Press, Cambridge, Mass., 1997.

[44] Andrew N. Meltzoff and M. Keith Moore. Imitation of facial and manual gestures by human neonates. *Science*, 198:75–78, October 1977.

[45] Andrew N. Meltzoff and M. Keith Moore. Imitation in newborn infants: Exploring the range of gestures and the underlying mechanisms. *Developmental Psychology*, 25:954–962, 1989.

[46] Andrew N. Meltzoff and M. Keith Moore. Explaining facial imitation: A theoretical model. *Early Development and Parenting*, 6:179–192, 1997.

[47] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[48] H. Miyamoto, M. Kawato, T. Setoyama, and R. Suzuki. Feedback-error-learning neural network for trajectory control of a robotic manipulator. *Neural Networks*, 1:251–265, 1988.

[49] Chrystopher L. Nehaniv and Kerstin Dautenhahn. *Imitation in Animals and Artifacts*, chapter The Correspondence Problem, pages 41–63. MIT Press, Cambridge, 2002.

[50] Aidan O'Dwyer. *Handbook of PI and PID controller tuning rules*. Imperial College Press, London, c2006.

[51] Jean Piaget. *Play, dreams and imitation in childhood*. W. W. Norton, New York, 1962.

[52] V. S. Ramachandran. Mirror neurons and imitation learning as the driving force behind "the great leap forward" in human evolution. Online essay, http://www.edge.org/documents/archive/edge69.html, June 1 2000.

[53] G. Rizzolatti, L. Fadiga, M. Matelli, V. Bettinardi, E. Paulesu, D. Perani, and F. Fazio. Localization of grasp representations in humans by PET: 1. observation versus execution. *Experimental Brain Research*, 111(2):246–252, September 1996.

[54] Giacomo Rizzolatti, Luciano Fadiga, Vittorio Gallese, and Leonardo Fogassi. Premotor cortex and the recognition of motor actions. *Cognitive Brain Research*, 3:131–141, 1996.

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

[56] Stefan Schaal, Auke Ijspeert, and Aude Billard. Computational approaches to motor learning by imitation. *Philosophical Transactions of the Royal Society of London: Series B, Biological Sciences*, 358(1431):537–547, 2003.

[57] John Searle. Minds, brains, and programs. *Behavioral and Brain Sciences*, 3(3):417–457, 1980.

[58] Gavin Simmons and Yiannis Demiris. Optimal robot arm control using the minimum variance model. *Journal of Robotic Systems*, 22(11):677–690, November 2005.

[59] Jun Tani and Masato Ito. Interacting with NeuroCognitive robots: A dynamical systems view. In *Proc. 2nd Int. Workshop on Man-Machine Symbiotic Systems*, pages 123–134, 2004.

[60] Jun Tani, Masato Ito, and Yuuya Sugita. Self-organization of distributedly represented multiple behavior schemata in a mirror system: Reviews of robot experiments using RNNPB. *Neural Networks*, 17:1273–1289, 2004.

[61] Paul J. Werbos. Backpropagation through time: what it does and how to do it. In *Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990.

[62] Daniel M. Wolpert, Kenji Doya, and Mitsuo Kawato. A unifying computational framework for motor control and social interaction. *Philosophical Transactions: Biological Sciences*, 358(1431):593–602, 2003.

[63] Daniel M. Wolpert and Mitsuo Kawato. Multiple paired forward and inverse models for motor control. *Neural Networks*, 11:1317–1329, 1998.

[64] Daniel M. Wolpert, R. Chris Miall, and Mitsuo Kawato. Internal models in the cerebellum. *Trends in Cognitive Sciences*, 2(9), 1998.

[65] Tom Ziemke, Dan-Anders Jihrenhed, and Germund Hesslow. Internal simulation of perception: a minimal neuro-robotic model. (in press).

Glossary

Behaviour

See *inverse model*.

Controller

See *inverse model*.

Correspondence problem

The problem of mapping visually perceived coordinates onto one's own motor capabilities.

Desired state

The state that the multiple paired models architecture should be in, in the next timestep. Also called *target* state.

Elton

The name of the humanoid robot.

Feedback error

The difference between the desired state at time t and the actual state at time $t+1$ constitutes a signal that is used to pull the system as a whole in the correct direction. The state of the environment corresponds directly to the joint angles of the simulator. The difference between the target joint angles and the actual joint angles becomes the joint angle velocity applied to the same joint angle, multiplied with a constant K that is fine-tuned for each experiment.

Forward model

It is easier to say what a forward model *does* than what it *is*. The forward model predicts the next state of the environment (i.e. it is a predictor), given the current state of the environment and the forces (i.e. motor commands) acting on the environment. It can be implemented in various ways, in this thesis it is implemented using neural networks.

Inverse model

As with the forward model, it is easier to say what an inverse model *does* than what it *is*. The inverse model produces the motor commands

needed to achieve a goal, when given the current state of the environment and the target state. Often called a *behaviour* or *controller*. It can be implemented in many ways, in this thesis it is implemented using recurrent neural networks.

Mirror neurons

An area of the brain where there is similar activity both when observing and executing the same action.

Module

An abstract entity holding the forward/inverse models. In figure 4.1, it can be seen as the box holding the inverse/forward models, the responsibility predictor and the likelihood function.

Motion

The total trajectory that constitutes one epoch. The *motion* is made up of several *movements*, i.e. the entire imitative act. It is I who have defined the boundaries between the movements.

Movement

A segment of the total *motion* that is to be imitated. For instance, the "Y" of the "YMCA" motion is a *movement*, "YMCA" is a *motion*. The movements correspond directly to the context values, i.e. they are separated at the same timesteps. The boundaries of the movements are defined by me.

Multiple paired models

An architecture consisting of several *modules*. Each module consist of a forward and inverse *model*. Strictly speaking, it should be called *multiple modules*, but I do not want to add further confusion by deviating from the name used in the literature.

Pattern

An input/output pair of a neural network. Normally, a *pattern* refers to one input/output pair, but in the work of Ito and Tani [32] a pattern is a *sequence* of patterns.

Performance error

The error of the system as a whole compared to the desired state, i.e. the difference between the desired state and actual state.

Prediction error

The error of the forward model. The error is $\hat{x}_{t+1} - x_{t+1}$, i.e. the difference between what the forward model predicted at timestep t compared to the actual state at timestep $t + 1$.

Sequence

A collection of patterns where the *order* of the patterns is crucial; the order of the patterns cannot be altered.

State

A set of variables that defines the environment. In my implementation this implies the following: at timestep t, the variables describe all the joint angles of the environment.

Target state

See *desired state*.

Trajectory

A sequence of coordinates or joint angles that constitutes a *motion*.

Appendix A

Attachments

A.1 Source code

The source code for MatLab and the breve simulator is attached to the thesis. All the source files (both the `*.m` and `*.tz`) have been well documented, although no specific commenting or coding standard has been followed. Notably, for all the MatLab functions written, the inputs and outputs along with a description have been specified in the beginning of the file, so by typing

```
> help function_name
```

this description will be written to the MatLab prompt.

A.2 Videos

Videos are attached that show the motions that are trained upon by the multiple paired models architecture. All the videos show the demonstrator side-to-side with the imitator, allowing for easy comparison of the performance of the imitator. There are videos for each of the experiments, i.e. *Ks.mov*, *cheerleader.mov* and *YMCA.mov*. In addition, the YMCA motion can be seen along with the Pro Reflex tracking, this can be seen in the video *YMCA with Pro Reflex tracking.mov*. If you cannot see the videos by simply clicking on the *.mov*-files, download the `QuickTime Player` from the following URL: http://www.apple.com/quicktime/download/