# Advances in Humanoid Control and Perception

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Marijn Frederik Stollenga

under the supervision of
Prof. Jürgen Schmidhuber

May 2016

## Dissertation Committee

| | |
|---|---|
| **Prof. Marco Wiering** | University of Groningen, Groningen, The Netherlands |
| **Prof. Tobias Glasmachers** | Ruhr-Universität Bochum, Bochum, Germany |
| **Prof. Stefan Schaal** | University of Southern California, USA & Max-Planck-Institute for Intelligent Systems, Tübingen, Germany |
| **Prof. Illia Horenko** | Universitá della Svizzera Italiana, Lugano, Switzerland |
| **Prof. Olaf Schenk** | Universitá della Svizzera Italiana, Lugano, Switzerland |

Dissertation accepted on 25 May 2016

| Research Advisor | PhD Program Director |
|---|---|
| **Prof. Jürgen Schmidhuber** | **Prof. Michael Bronstein** |

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Marijn Frederik Stollenga
Lugano, 25 May 2016

*I dedicate this thesis to my beloved parents and grandparents, who have taught me to see the world with an open mind and always encouraged and supported me in every way. Without them this thesis would not have been possible.*

# Abstract

One day there will be humanoid robots among us doing our boring, time-consuming, or dangerous tasks. They might cook a delicious meal for us or do the groceries. For this to become reality, many advances need to be made to the artificial intelligence of humanoid robots. The ever-increasing available computational processing power opens new doors for such advances. In this thesis we develop novel algorithms for humanoid control and vision that harness this power. We apply these methods on an iCub humanoid upper-body with 41 degrees of freedom.

For control, we develop Natural Gradient Inverse Kinematics (NGIK), a sampling-based optimiser that applies natural evolution strategies to perform inverse kinematics. The resulting algorithm makes very few assumptions and gives much more freedom in definable constraints than its Jacobian-based counterparts. A special graph-building procedure is introduced to build Task-Relevant Roadmaps (TRM) by iteratively applying NGIK and storing the results. TRMs form searchable graphs of kinematic configurations on which a wide range of task-relevant humanoid movements can be planned. Through coordinating several instances of NGIK, a fast parallelised version of the TRM building algorithm is developed. To contrast the offline TRM algorithms, we also develop Natural Gradient Control which directly uses the optimisation pass in NGIK as an online control signal.

For vision, we develop dynamic vision algorithms that form cyclic information flows that affect their own processing. Deep Attention Selective Networks (dasNet) implement feedback in convolutional neural networks through a gating mechanism that is steered by a policy. Through this feedback, dasNet can focus on different features in the image in light of previously gathered information and improve classification, with state-of-the-art results at the time of publication. Then, we develop PyraMiD-LSTM, which processes 3D volumetric data by employing a novel convolutional Long Short-Term Memory network (C-LSTM) to compute pyramidal contexts for every voxel, and combine them to perform segmentation. This resulted in state-of-the-art performance on a segmentation benchmark.

The work on control and vision is integrated into an application on the iCub robot. A Fast-Weight PyraMiD-LSTM is developed that dynamically generates weights

for a C-LSTM layer given actions of the robot. An explorative policy using NGC generates a stream of data, which the Fast-Weight PyraMiD-LSTM has to predict. The resulting integrated system learns to model the effects of head and hand movements and their effects on future visual input. To our knowledge, this is the first effective visual prediction system on an iCub.

# Acknowledgements

Firstly, I would like express my sincere gratitude to my thesis advisor, Jürgen Schmidhuber, who has always encouraged me to pursue new ideas and gave me the academic freedom to pursue them. Without his guidance this thesis would not have been possible.

Furthermore, I would like to thank Bas Steunenbrink, Dan Ciresan and Jan Koutník for their invaluable feedback on my thesis. I would also like to thank my colleagues Leo Pape and Varun Kompella, who's collaborations greatly helped me in the beginning of my PhD, and Jonathan Masci and Wonmin Byeon with whom I had some of my most enjoyable collaborations. I also thank the rest of my colleagues, Sohrob Kazerounian, Matt Luciw, Faustino Gomez, Alan Lockett, Michael Wand, Alexander Förster, Klaus Greff, Rupesh Srivastava, Kail Frank, Jürgen Leitner and Andi Heinrich for their great collaborations and interesting discussions. And I thank our secretary Cinzia Daldini, who was always prepared to help selflessly and made my PhD studentship go smoothly.

Finally, I would like to sincerely thank my dissertation committee, Stefan Schaal, Marco Wiering, Tobias Glasmachers, Illia Horenko and Olaf Schenk, who sacrificed their time to review my work and helped me with their valuable feedback.

# Contents

# Figures

# Tables

# Notation

| | |
|---|---|
| | **General** |
| $a \in \mathbb{R}^n$ | Refers to a vector $a$ with $n$ real values. |
| $dim(x)$ | The dimensionality of space $x$, or $X$ if $x \in X$, where $x$ is a sample of space $X$ |
| $x\,y$ **or** $x \cdot y$ | Multiplication of $x$ and $y$. If $x$ and $y$ are matrices, this becomes a matrix multiplication. |
| $\lvert \cdot \rvert$ | $\ell^1$-norm of a vector. |
| $\lVert \cdot \rVert_2$ | $\ell^2$-norm of a vector. |
| $\lVert \cdot \rVert_\infty$ | $\ell^\infty$-norm of a vector. |
| $a \overset{\rho}{\leftarrow} b$ | Used for updating parameters; defined as: $a_{n+1} = \rho a_n + (1-\rho) b_n$, where $a, b \in \mathbb{R}^N$. |
| $a \leftarrow b$ | Is defined as $a \overset{1}{\leftarrow} b$ |
| $\theta$ | A set of parameters |
| $f_\theta(x)$ | A function parameterised by $\theta$, with input $x$. In principle $\theta$ is just another input and can be written as $f(\theta, x)$, but the former distinguishes training parameters and input |
| $\nabla_x f(x)$ | Gradient of function $f(x)$ towards input $x$ |
| $E(y, \hat{y})$ | A function measuring the error between the actual and desired output of a function |
| $\bullet$; $a_{0,\bullet}$ | Selects a dimension in multi-dimensional array. E.g. if $a \in \mathbb{R}^{N \times M}$ is a matrix, then $a_{0,\bullet} \in \mathbb{R}^M$ selects the first row. |
| | **Part I: Control** |
| $q \in \mathbf{Q}$ | Joint parameter vector $q$ in parameter-space $Q$ |
| $\tau \in \mathbf{T}$ | Task space vector $\tau$ in task-space $T$ |
| $\mathbf{P}$ | Kinematic configuration-space |
| $\mathbf{T}$ | Task-space |
| $\tilde{\nabla}_x f(x)$ | Natural Gradient of function $f(x)$ towards input $x$ |
| | **Part II: Vision** |
| $x$ | Generally refers to input to a function |
| $y$ | Generally used as an output of a function |
| $\hat{y}$ | Generally used as the desired output of a function |
| $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ | The $\tanh$ function. If $x$ is a vector, the function is applied element-wise. |
| $\sigma(x) = \frac{1}{1+e^{-x}}$ | The sigmoid function. If $x$ is a vector, the function is applied element-wise. |
| $softmax(x_i) = \frac{e^{x_i}}{\Sigma_j e^{x_j}}$ | The softmax function. Often used for classification as the outputs can be interpreted directly as probabilities. |
| $ReLu(x_i) = \max(x_i, 0)$ | The Rectified Linear (ReLu) function. Simply assures a value is non-negative, by returning $0$ for negative numbers. |
| $x * f$ | Convolution operation (explained in Section 7.5). |
| $a \oplus b$ | Concatenation of two vectors $a \in \mathbb{R}^{\lvert a \rvert}$ and $b \in \mathbb{R}^{\lvert b \rvert}$. The resulting vector is $a \oplus b \in \mathbb{R}^{\lvert a \rvert + \lvert b \rvert}$. |

# Chapter 1

# Introduction

**Figure 1.1. Mechanical Knight** A humanoid designed by Leonardo da Vinci; the robot can move through a mechanical contraption connected to 'tendons'.
*Photo by Erik Moller. Leonardo da Vinci. Mensch - Erfinder - Genie exhibit, Berlin 2005.*

Many have dreamed about a robot that cleans their rooms or performs other boring or dangerous task for them. For many scenarios, it is thought that this robot should be human-like in shape and size since these prospective jobs exist in environments that are designed for us. For example, a cooking robotic should be able to reach the table top and use the same utensils that we do. Such a human-like robot is called a *humanoid*.

Decades of research have gone into the development of humanoids, but in many ways the field is still at the beginning stage and many advancements are needed to reach a point where we can walk into a store and buy a new house-robot. One of the most promising developments in recent years is the amazing increase in available computational power. The result is that completely new approaches to humanoid

intelligence become possible, that were previously unthinkable. This thesis describes our advancements we made in this field that utilise this new found computational power for novel approaches to humanoid control and visual processing.

## 1.1 Humanoids

The dream of mechanical machines that perform our menial, boring or difficult tasks has existed for centuries. The Greek mythologies contain many artificial beings, such as the bronze man Talos that defended Crete. The ancient Chinese *Lie Zi* text talks about an inventor Yan Shi that made an automaton resembling a human.

More realistically, around 10-70 AD, the famous inventor Heron of Alexandria designed the first 'robot'; a mechanical table that was programmable by adding strings of different lengths. This is one of the first designs of what we would later call robots after the Czech writer *Karel Čapek* coined the term in 1921. Leonardo da Vinci designed the famous Mechanical Knight (see Figure 1.1), one of the first human inspired robots, applying his deep knowledge of human anatomy. One of his main insights was to give the mechanical robot tendons that ran through the arms and were controlled centrally. This was useful to reduce the complexity and weight of the device.

## 1.2 Intelligence

Although all these robots were inventive, they were ahead of their time; they never were able to perform complex tasks since there was no technology that could perform the needed information processing and control. The main ingredient missing was *artificial intelligence*. There simply was no technology to do computations on the scale necessary to do complex movements similar to humans, or to perform complex visual processing.

The human brain, that controls our body and performs visual processing, has about 100 billion neurons. About 20 billion of them are in the *neo-cortex*, the largest part of our brain, associated with complex information processing of which a large part is dedicated to visual processing (estimates vary between 20% and 60% depending on how general we define 'visual processing' [Belliveau et al., 1991]).

Most neurons are in the cerebellum, about 60- to 80 billion, but these neurons are much smaller and the cerebellum takes up only 10% of the total brain volume. This part of the brain is mostly concerned with movements of the body. Finally, many neurons in the spine perform their own information processing and implement many reflexes.

**Universal Computation** It might seem that this daunting complexity means we could never reproduce the intelligence needed for a humanoid in a computer. How can a computer, which can only add, multiply, compare and load and store numbers, perform functions as complex as the brain can? This problem was tackled in the early to mid 30's, after Gödel, 1931 invented the first universal formal language based on integers to prove the inherent limits to mathematics. Church, 1936 extended Gödels work and developed the equally powerful Lambda Calculus to prove the Entscheidungsproblem. These languages existed completely in mathematics and did not translate to a physical computing machine. This changed when Turing, 1936, advised by Church, proved the Entscheidungsproblem using his physically realisable, and fantastically inefficient, universal Turing Machine. Later, Post, 1936, would invent another Turing Machine equivalent language, and the first efficient computation machines were designed independently by Zuse, 1936; Zuse, 1967.

The main result of these works is the famous theory of *universal computation*, which showed that when a computational system has a certain small set of instructions, it can compute any complex function we can think of. Thus, there is no 'higher complexity' which neurons have, that cannot in principle be achieved by computers. The question remains if this *theoretical* ability can be achieved in *reality*. Nowadays, computers are incredibly fast, with a modern graphics card being able to perform trillions (1000 billion!) instructions per second. This gives hope that we start to reach the processing power that can achieve the capabilities of the human brain.

However, this power is meaningless if we don't know *how* to use it! *Which set of instructions* leads a robot to perform a complex movement, or visually recognise a person? Such sets of instructions are called *algorithms*, which are essentially recipes for the computer to follow. The objective of this thesis is to find novel algorithms that can employ this massive computational power for humanoid control and visual processing.

## 1.3 The iCub

The *humanoid* that is used throughout this thesis is the *iCub* [Tsagarakis et al., 2007; Metta et al., 2008] (see Figure 1.2). The iCub is a humanoid robot designed by the Italian Institute of Technology as part of the EU project *RobotCub* and is used in many research labs across Europe. The unique part about the iCub is that its design is open-source, both for hardware and software, which makes sure it is continuously developed and improved upon. The robot has movement capabilities similar to that of humans; it can move its head, waist, arms and legs, and even its eyes, giving it 53 degrees of freedom. In comparison, a common robot arm in a car factory has only six

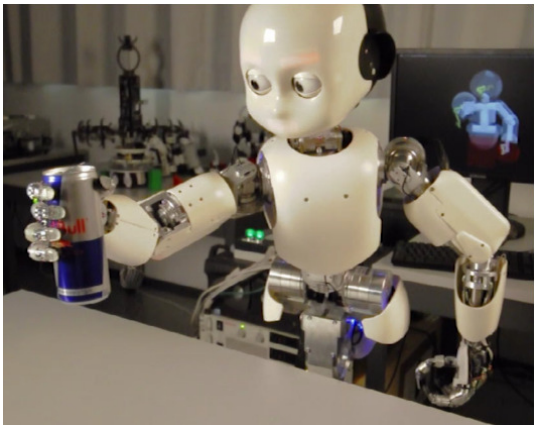


**Figure 1.2. The iCub Humanoid** The iCub is a complex humanoid robot with many degrees of freedom, designed to perform human-like motions.

degrees of freedom. The research in this thesis is done on a leg-less iCub mounted on a metal mount, giving it 41 degrees of freedom.

An interesting design choice is that the torso, arms, and fingers are controlled through metal tendons, which are connected to electric motors centred in the torso and hip. This mimics human physiology, where tendons with muscles are the main actuators, and is similar to the Mechanical Knight designed by da Vinci (see Figure 1.1). The advantage is that the weight is centred around the base of the robot, relieving the arms of heavy electric motors. The tendons also function as physical force limiters; they are designed to break under loads that would otherwise destroy the robot internally.

To give an idea of the complexity of the robot, several design diagrams are shown in Figure 1.3. The head contains 6 motors, 2 to tilt the head, 1 to rotate it, 2 to control the direction of the eyes and 1 to control the vergence (how much the eyes point towards each other). The arms have 16 joints each, 7 of which control the angle and rotation of the shoulder and elbow, and 1 turns the wrist. The other 9 control the hand, of which 2 control the direction of the thumb, 1 controls the spreading of the fingers, and the other 6 bend and flex the thumb, index and middle finger, and the last bends the 2 last fingers. Finally, the robot can bend forward and sideways, and turn around its waist, using 3 hip-joints.

**Sensors** The iCub has several sensors to obtain information from its environment. Two movable cameras placed in the head function as eyes, each capturing colour images at a resolution of $640 \times 480$ at a frequency of 30 Hz. The head also contains two microphones placed at the position of its ‘ears’ so it can hear, but these are not

used in this thesis. The joints have internal sensors to detect the rotation of the joints, giving the robot proprioseption, necessary to determine the current state of the robot and give the right control commands. Finally, the fingertips have capacitive touch sensors capable of detecting contact with conductive surfaces.

**Software** The iCub software environment relies on the YARP (Yet Another Robot Platform) framework [Metta et al., 2006]. This framework builds a unified messaging layer accessible over different computers on the same network. It provides an abstraction layer that lets services transparently open ports, and send and receive data through them under low latencies. It is used for all data communication, from sending video capture from the cameras to sending control commands to the electric motors.

The iCub itself has an internal CPU running a real-time variant of Debian Linux. The individual electric motors have their own controller boards, which perform low-level control. The control boards implement several PID-controllers, which receive desired velocities or position for the motors and output the right amount of power to them.

## 1.4 Research Questions

The main goal of this thesis is to advance the field of humanoid robotics to get closer to the dream of a humanoid that cleans our room or cooks a delicious meal for us. To this end, we will develop novel algorithms that are designed to make use of the recent massive increase in computational power.

Our work is guided by the following research questions:

**Main Research Question:**
*What innovations are needed to harness the vast amount of computational power to advance humanoid robotics?*
This question is pragmatic in nature, which is needed for an applied field like that of robotics. We don't necessarily look for the most biologically plausible approaches, and neither do we aim for the most mathematically precise. We want novel algorithms that are designed to harness the available computational power and have clear benefits that serve to advance the field. This has led to two sub-directions with corresponding questions:

**Research Question 1:**
*What innovations are needed for humanoid robotic control to harness the vast amount of computational power, and what are the benefits?*

Robotic control is a complex problem with decades of research behind it. However, many of the algorithms in the field were designed to have a low computational overhead, such as Jacobian-based approaches (see Section 2.3.1). We will introduce a novel sampling based approach that applies Natural Evolution Strategies (see Section 3.2) to get rid of many limitations that are present in the traditional approaches, at a higher computational cost. This question is addressed in *Part I* of the thesis, comprising Chapter 2 to Chapter 6.

**Research Question 2:**
*What innovations are needed for computer vision algorithms to harness the vast amount of computation power, and what are the benefits?*
The main recent advances in computer vision come from neural networks, introduced in Chapter 7, and especially their convolutional variants (Section 7.5). These approaches already stretch the current hardware to their computational limits, but the hardware has also kept evolving. We will develop algorithms that push these limits further by incorporating visual feedback in several ways to improve the state of the art. This research question is addressed in *Part II* of this thesis, in Chapter 7 to Chapter 10. These algorithms are evaluated on datasets that are not related to humanoid robots.

**Research Question 3:**
*How can the algorithms emerging from Part I and Part II be integrated in an actual application on a humanoid? Part I* and *Part II* investigate control algorithms and computer vision algorithms in isolation. The question remains how the products of these two directions can come together to benefit humanoid robotics in synchrony. In the final *Part III* we show how our developments in control (Part I) and vision (Part II) are integrated and can work together to learn models of the world that can predict future observations.

## 1.5 Overview of Thesis

This thesis develops several algorithms that make use of the massive computational power of today's computers. These algorithms are divided over three parts, each comprising several chapters:

**Part I: Control** The first part concerns our advancements in *control algorithms* for a humanoid.

**Chapter 2: Control of a Humanoid** This chapter introduces the background needed for this part of the thesis, explaining how the movement of a humanoid robot is formalised.

**Chapter 3: Natural Gradient Inverse Kinematics** This chapter introduces the Natural Evolution Strategies algorithms that were developed by Wierstra et al., 2014 and Glasmachers et al., 2010a. Using these algorithms, we develop our algorithm called Natural Gradient Inverse Kinematics (NGC), which forms the basis of our approach to control. This work was published in 2013 [**M. F. Stollenga** et al., 2013b].

**Chapter 4: Task-Relevant Roadmaps** Using NGC, we develop an algorithm to build Task-Relevant Roadmaps (TRM); graphs formed by kinematic configurations such that robotic movements can be planned. This work was published in 2013 [**M. F. Stollenga** et al., 2013b] accompanied with a video [**M. F. Stollenga** et al., 2013a], which won the AAAI Video award of 2013.

**Chapter 5: Parallelising TRM** To make use of multi-core computers we developed a parallel version of TRM which was applied on a fast 64-core computer speeding op the TRM construction process. This work was published in 2014 [**M. F. Stollenga** et al., 2014].

**Chapter 6: Natural Gradient Control** The graphs of a TRM are useful, but have disadvantages. In this chapter we take a new look to the NGIK algorithm and adapt it, such that it runs as an online optimisation algorithm which constantly outputs a control signal used to control the humanoid. The resulting Natural Gradient Control (NGC) algorithm was published in 2015 [**M. F. Stollenga** et al., 2015].

**Part II: Vision** In this part we leave the control algorithms behind us and focus on how computation power can be improve *visual processing algorithms*. We develop algorithms that incorporate attention mechanisms that give them more control over *what* is processed *when*. This dynamic aspect allows for better utilisation of the accessible computation power.

**Chapter 7: Artificial Neural Networks** In this chapter we introduce the background of Artificial Neural Network (NNs) and Recurrent Neural Networks (RNN) and their variants that form the basis of our algorithms.

**Chapter 8: Deep Attention Selective Neural Networks** With NNs as a basis, we develop our own Deep Attention Selective Neural Networks (dasNet). Using an

attentional mechanism, *dasNet* controls its own visual attention which resulted in state-of-the-art performance at the time of its publication. This work was developed in joint work with Jonathan Masci and published in a shared first-author paper [**M. F. Stollenga* and J. Masci*** et al., 2014].

**Chapter 9: Pyramidal Multi-Dimensional Long Short-Term Memory Networks** Inspired by *dasNet*, we develop an advanced volumetric segmentation algorithm that applies Long Short-Term Memory networks to perform adaptive processing, resulting in state-of-the-art performance in biological volumetric segmentation. This algorithm was developed in joint work with Wonmin Byeon and was published in a shared first-author paper [**M. F. Stollenga* and W. Byeon*** et al., 2015].

**Part III: Integration** This final part serves as a proof-of-concept on how the advancements made in the first two parts can be applied to the humanoid robot.

**Chapter 10: Fast-Weight PyraMiD-LSTM** In this chapter, we integrate our work from Part 1 and Part 2 into an application on the robot. We develop a novel variant of the PyraMiD-LSTM that can predict future images from an image stream, resulting in the Fast-Weight PyraMiD-LSTM. The robot is controlled by the NGC algorithm to explore a hand movement. The result is a system that can create non-trivial predictions of future observations for several time steps into the future. This serves as an example of how the algorithms introduced in this thesis can work together.

**Chapter 11: Conclusion** In this final chapter, we review our contributions in this thesis, draw conclusions and look at possible future work.

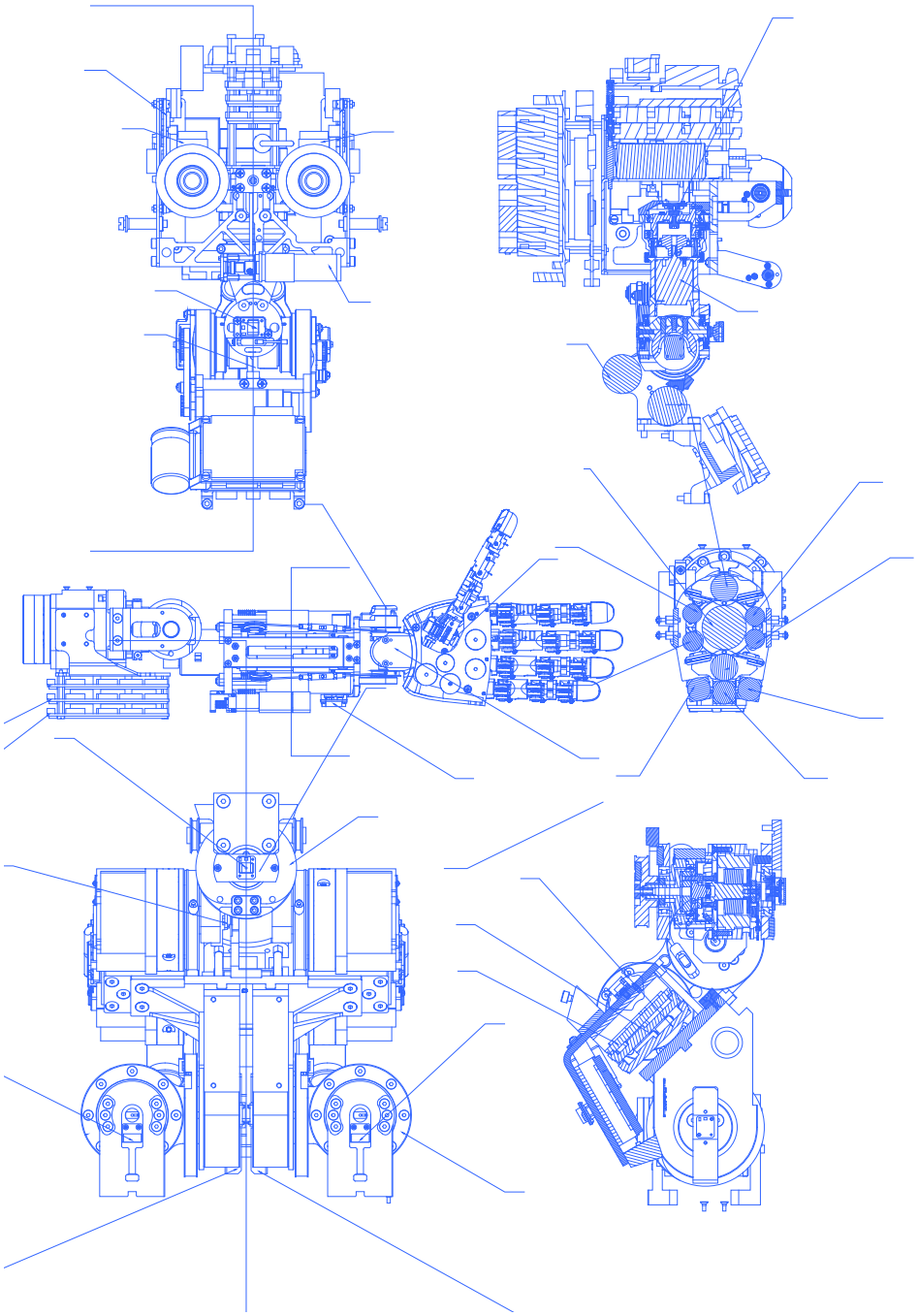


**Figure 1.3. Diagrams of the iCub** The iCub is a complex robot with many joints. These diagrams show the head, arms and torso, giving an idea of the complexity of the robot. The designs are open and available through a Subversion repository, published by the Italian Institute of Technology [*iCub Mechanical Drawings (Subversion Repository)*].

# Part I

# Control

# Chapter 2

# Control of a Humanoid

One of the main differences between a robot and a regular desktop computer, is the ability of the robot to move and interact with the environment. The first part of this thesis will involve algorithms that are designed to move the iCub humanoid. In this chapter, we will first introduce the basic formalisation of robotic control.

Moving a robot in desired ways is a hard problem, which gets harder the more complex the robot gets. For a simple two-wheeled robot, we can still manually figure out a control algorithm: turn the right wheels to turn left, turn the left wheels to turn right, and turn both wheels to move forward. However, for a humanoid like the iCub upper body with 41 degrees of freedom, such simple approaches do not work. Therefore, we need algorithms that translate a high-level description of our desired motion to lower level commands for the robot.

We will introduce the formalisms used in robot kinematics and the typical approaches used in this field. Most of these approaches are fundamentally limited due to strict formalisations of the kinematic problem that constrain the type of desired motion that can be expressed. In the following chapters, we will develop algorithms to control the humanoid that are more computationally intensive, but are much more flexible and thus try to remedy the limitations of other algorithms.

## 2.1 Kinematic Chain

A humanoid is built of *rigid bodies* connected through *joints*, controlled by electric motors, and *links*, which are rigid connections (see Figure 2.1). By turning these motors, the joint angles change and the robot moves. The state of a *joint* is described by its angle and angular velocity. In this thesis we will generally ignore the angular velocities, and let the lower level controllers take care of them.

Ignoring the angular velocity, we represent all joint angles in a single vector

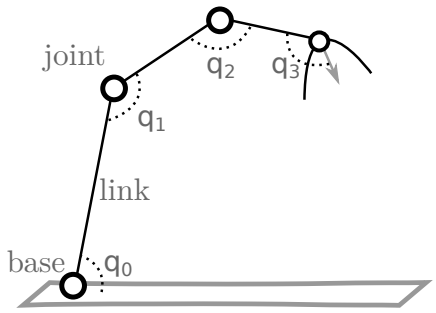



**Figure 2.1. Robot Arm** The topology of a robot is defined with a *base* and *rigid bodies* that are connected through *joints* and *links*. The robot moves through changing the angles of its joints, $\{q_0, q_1 \dots q_n\} = q$

.

$q \in \mathbf{Q} \in \mathbb{R}^J$, called the *joint parameters* of the robot. Here $\mathbf{Q}$ is the set of all possible *joint parameters*, and $J$ is the number of joints. The i'th joint is denoted by $q_i \in \mathbb{R}$, where $i$ is an index between $1$ and $J$, and $q_i$ is a real-valued number describing the angle.

To formally represent the topology of the robot, we first define the *base* of the robot, typically the hip or the torso, which forms our point of reference. We follow the skeletal structure from the base to each body part[1], forming a chain. For example, the finger connects to the hand, the hand to the arm, the arm to the torso and the torso to the hip of the robot. Each of these connections can be represented by a transformation, and these transformations can be chained together to calculate the location and orientation of each body part. The body part that is typically used for manipulation, such as a gripper and the hand of a humanoid, is called an *end effector*.

The resulting chains are represented by a tree structure called the *kinematic chain*. Given a set of *joint parameters* and the *kinematic chain* the positions and orientations of all body parts can be calculated, collectively called the *kinematic configuration*. This calculation is called *forward kinematics*.

## 2.2 Forward Kinematics

To calculate the kinematic configuration of the robot, we use transformation matrices to represent the transformations resulting from the kinematic chain. Transformations define rotations, e.g. to represent the rotation caused by a joint, and translations, e.g. to represent a link between two joints. Given a vector $v \in \mathbb{R}^3$ in Cartesian space,

[1]For most robots we use the term *rigid bodies* to refer to parts of the robot, but since humanoids are human-like we refer to them as *body parts*.

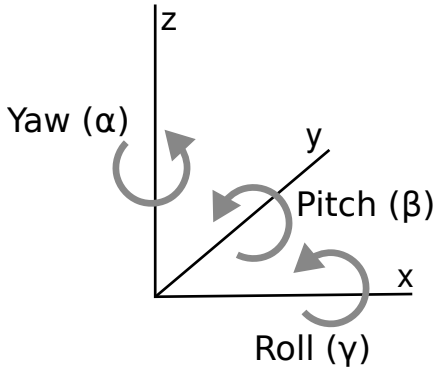



**Figure 2.2. Rotation Matrix** To define a rotation matrix, the rotations over all three axes have to be considered. Rotation of the x-axis is called the *roll*, over the y-axis is called the *pitch*, and over the z-axis is called the *yaw*.

the transformation matrix $R \in \mathbb{R}^{3\times3}$ can be used to calculate the transformed vector $v_{transformed}$ by matrix multiplication:

$$v = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.1}$$

$$v_{transformed} = R \cdot v \tag{2.2}$$

**Rotation** If we define an x-, y- and z-axis, a rotation over the x-axis is called *roll*, over the y-axis is called the *pitch* and over the z-axis is called *yaw*, as shown in Figure 2.2. The corresponding transformation matrices are:

$$R_{yaw}(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

$$R_{pitch}(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \tag{2.4}$$

$$R_{roll}(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix} \tag{2.5}$$

A general transformation matrix can be built by multiplying these matrices:[2]

$$R_{yaw,pitch,roll}(\alpha,\beta,\gamma) = R_{yaw}(\alpha) \cdot R_{pitch}(\beta) \cdot R_{roll}(\gamma) \tag{2.6}$$
$$\tag{2.7}$$

**Translations** To add translations within this framework, we represent the positions with a vector $v$ that holds the x, y and z position and a fourth value $1$:

$$v = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.8}$$

This allows us to combine translations, defined by a vector $t = \{t_x, t_y, t_z\} \in \mathbb{R}^3$, and rotations in a single matrix $4 \times 4$:

$$R(\alpha,\beta,\gamma,t) = \begin{bmatrix} & & & t_x \\ & R(\alpha,\beta,\gamma) & & t_y \\ & & & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

This final transformation matrix is parameterised by parameters $\alpha, \beta, \gamma, t$. Note that the matrix turns into an identity matrix if all parameters are $0$. All joints and links in the iCub can be represented by these transformations. For joints, the parameters depend on their angle $q_i$, whereas links are rigid and have a fixed parameterisation. In general, we write $R(q)$ to show a certain transformation matrix depends on the *joint parameters*.

Since matrix multiplications are linear, the sequence of transformations in the kinematic chain can be calculated efficiently by multiplying all matrices. For example, the position of the hand is calculated from the base position as follows:

$$v_{hand} = \tag{2.10}$$
$$R_{lowerarm \to hand}(q) \cdot R_{upperarm \to lowerarm}(q) \cdot \tag{2.11}$$
$$R_{torso \to upperarm}(q) \cdot R_{base \to torso}(q) \cdot v_{base} = \tag{2.12}$$
$$R_{base \to hand} \cdot v_{base} \tag{2.13}$$

[2]The order of rotations matters and is chosen to fit the corresponding joint. Here we show one example of ordering.

**Orientation** To calculate the orientation of a body part, for example the hand, we define a vector pointing in a certain direction with respect to the relevant body part and calculate the final orientation. E.g. we could be interested in a vector coming out of the palm of the hand:

$$v_{palm\_dir} = v_{palm\_point} - v_{hand} \tag{2.14}$$

where $v_{palm\_point}$ is calculated by adding a translation transformation:

$$v_{palm\_point} = R_{hand \to palm\_point} \cdot R_{base \to hand} \cdot v_{base} \tag{2.15}$$

To quickly calculate such directional vectors, the full transformation matrix $R_{base \to hand}$ is stored during the calculation of forward kinematics.

Both position $v$ and orientation $r$ of all body parts can now be efficiently calculated. In general, we call the set of all positions and orientations of body parts the *kinematic configuration* $p$:

$$p = \{(v_{lefthand}, r_{lefthand}), (v_{righthand}, r_{righthand}), \dots\} \in \mathscr{P} \tag{2.16}$$

To simplify notation we use function $f$, called the *forward kinematics function*, to describe the process of calculating the *kinematic configuration* $p$ from given *joint parameters* $q$ as described above:

$$f(q) = p \tag{2.17}$$

**Software** In this thesis, we apply the MoBeE framework [Frank et al., 2012] for *forward kinematics*. It takes an XML configuration file which defines the topology of the robot including all joints and performs fast forward kinematics and collision detection using the SOLID 3.5.6 [*libSolid: Collision detection library.*] collision detection library. Additionally, external objects can be added to model the environment, which are included in the collision detection. The resulting kinematic configurations can also be visualised.

## 2.3 Inverse Kinematics

As discussed, the *forward kinematics* function $f$ calculates the *kinematic configuration* $p$ given the *joint parameters* $q$.

$$f(q) = p \tag{2.18}$$

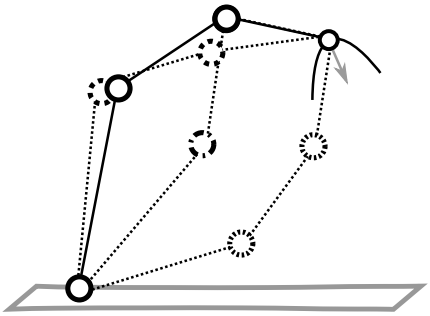


**Figure 2.3. Inverse Kinematics** IK tries to find joint parameters fitting a desired kinematic configuration. This is a hard problem since there are typically many, or even infinitely many, solutions. Finding stable methods that deal with these problems is one of the main goals of IK.

But to control the robot we need the reverse: given the desired configuration of body parts we want to find the corresponding joint parameters. This problem is called *inverse kinematics*:

$$f^{-1}(p) = q \tag{2.19}$$

where $f^{-1}$ is the inverse of the forward kinematics function, and $p$ is (a subset of) a kinematic configuration. Although $f$ is easily computed, its inverse $f^{-1}$ cannot usually be computed directly and may not even be a well-defined function. Firstly, it is not known if a solution even exists for the desired configuration $p$. And if there is, there are typically multiple (if not infinite) solutions. Figure 2.3 shows several solutions for a two-dimensional arm where the gripper is in the desired position and orientation. With only a few degrees of freedom in a two-dimensional world, there are already infinite solutions to this problem. This problem only gets bigger as we go to three-dimensional spaces and many degrees of freedom.

One solution is to constrain the problem such that there is *only one* defined solution and $f^{-1}$ can be analytically expressed [Chang, 1987; Hemami, 1987; Kauschke, 1996]. Due to the non-linearity and size of the joint parameter space, this is very difficult and is only realistic in very constrained settings.

### 2.3.1 Jacobian-based Approaches

A more popular approach is to give up on getting a global analytic expression of $f^{-1}$ and resort to local iterative optimisation. We can define a positive real-valued error function over a kinematic configuration $p$ that decreases as we get closer to our

desired configuration $p'$:

$$E(f(q), p') \geqslant 0 \tag{2.20}$$

$$q^* = \min_q E(f(q), p') \tag{2.21}$$

We can then find the optimal joint parameters $q^*$ that minimise error $E$ by calculating the gradient towards the joint parameters and use that in our optimisation:

$$\nabla_q E(f(q), p') \tag{2.22}$$

A simple way would be to iteratively follow the negative of the gradient until function $E$ does not decrease anymore:

$$q_{new} = q - \nabla_q E(f(q), p') \tag{2.23}$$

**Jacobian** To calculate the gradient of the error function $E$ towards the joint parameters, the derivative has to be calculated through all transformation matrices in the *kinematic chain*. Since these matrices have vector valued outputs there is no 'regular' gradient; instead, it is a *matrix* formed by all first-order partial derivatives for all combinations of elements in the output and input vector. This matrix is called the *Jacobian Matrix* [Wolovich and Elliott, 1984; Zohdy et al., 1989]. Control methods that use the gradient are thus said to be *Jacobian based* or *gradient based* controllers [Paul and Shimano, 1979; Angeles, 1985; Goldenberg et al., 1985; Tsai and Morgan, 1985].

However, these Jacobian approaches have several problems. Since the search space is highly nonlinear, and the gradient is only a local linearisation of this search-space, simply following the gradient can result in sub-optimal solutions. Typical examples are *local minima*, where the solver gets stuck in a local dip in the search space, or *plateaus* where the gradient is very small causing the solver to waste many iterations.

*Singularities*, areas of the search space where the gradient misbehaves, are another big problem. Such problems can occur when there is no solution to the desired kinematic configuration, e.g. when a goal is out of reach. Or when regularisations of the search space, that were added to help the solver, interact in complex ways and cause the search-space to be ill-defined.

Another example is shown in Figure 2.4, where a robotic arm is completely stretched, and the goal is to move the arm down (denoted by the red arrow). The gradient does not give a solution here because every change in joint angles results in a sideways movement, orthogonal to the desired goal. Only after the arm is slightly bent, the gradient will result in the right movement. These problems are increasingly present when the complexity of the robot increases and are especially true for a humanoid

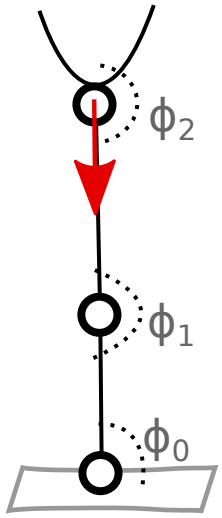



**Figure 2.4. Singularities** Singularities are points in function space where said function behaves abnormally, and are a big problem for IK. In this example, an arm extended upwards and now needs to move back down (denoted by the red arrow). However, in this stretched position the gradient around this kinematic configuration is zero, since every local movement of the joints results in horizontal movement of the gripper; not vertical. Such points are problematic for gradient based controllers.

robot.

Still, Jacobian-based approaches are still very popular. For example, Baerlocher and Boulic, 2004 applied Jacobian control to humanoids. However, setting up the constraints properly is still an involved process and the search space has to be reduced to keep the degrees of freedom under control.

## 2.4 Sampling-based Approaches

Inspired by these problems, researchers have recently turned to ignoring the gradients altogether by using *sampling based* approaches [Dutra et al., 2008; Hecker et al., 2008; Courty and Arnaud, 2008; Graça Marcos et al., 2009]. Sampling based algorithms explore the search space by evaluating the error function $E(f(q), p)$ using *only* forward kinematic calculations. By 'guessing' new parameter vectors $q$ in systematic ways, they try to find parameter vectors with an increasingly lower error.

**Metropolis-Hastings** A famous sampling algorithm is Metropolis-Hastings sampling (MH; Metropolis et al., 1953). MH is designed to generate samples from a complex distribution $p(x)$ that is hard to sample from, by using a simpler distribu-

tion (we will use a normal distribution) $r(x'|x) = \mathcal{N}(x' \mid x, \Sigma)$. Generally written $p(x) = f(x)/Z$, where $f$ can be a complex function and $Z$ is the normalisation constant such that the probabilities sum to 1: $\int_{-\infty}^{\infty} p(x) \equiv 1$. However $Z$ is unknown and might be practically impossible to calculate, thus making sampling difficult. To get around this, MH keeps track of a sample $x'$, and samples a new sample from the simple distribution which has this sample $x'$ as its first moment $x'' \sim \mathcal{N}(x', \Sigma)$. This sample is then *accepted* with a probability of

$$p_{accept}(x'' \mid x') = \min(1, \frac{p(x'') \cdot \mathcal{N}(x' \mid x'', \Sigma)}{p(x') \cdot \mathcal{N}(x'' \mid x', \Sigma)}) \tag{2.24}$$

$$= \min(1, \frac{f(x'')}{f(x')}). \tag{2.25}$$

The last step is possible since the normalisation constant $Z$ cancels out, and the normal distribution is symmetric and cancels out as well. If the sample is accepted it will be kept $x' = x''$ and otherwise discarded $x' = x'$. This process is repeated a satisfactory number of times, often determined empirically.

To use MH for optimisation, the function $f$ can be set to be $f(x) = \exp(-c(x))$, where $c(x)$ is a cost function. If we take $\Sigma = \sigma \cdot \mathbb{I}$, it is known that if we decrease $\sigma \to 0$ infinitely slowly the accepted sample will correspond with the maximum a posteriori (MAP) estimate; the sample with the highest probability. This MAP sample from $p(x) = f(x)/Z$ then coincides with the sample with minimum cost. This process is similar to Simulated Annealing which has been used for robotic control [Dutra et al., 2008]

**Sequential Importance Resampling** MH uses only one sample at a time. It seems more efficient to take several samples at each step to perform a wider search of the search space. A method that does that is Sequential Importance Resampling (SIR; Kitagawa, 1996; Gordon et al., 1993). Instead of one sample $x'$, SIR holds a set of samples $x'_i, i = 1 \cdots N$, which represent the distribution. By sequentially generating new samples and weighting them accordingly, SIR is more robust for complex distributions (for details of this algorithm see [Gordon et al., 1993]).

The sampling based approaches do not use any gradient information and avoid the problems with singularities of Jacobian based algorithms. However, their lack of gradient information typically makes them slower and thus they require more computational power. Given the advantages they give and the increase in general computational power available, this is becoming a compelling trade-off. This is seen by an increase of robotic control algorithms using sampling based approaches such as Simulated Annealing [Dutra et al., 2008], Sequential Monte Carlo (SMC) [Courty

and Arnaud, 2008], and particle filters [Hecker et al., 2008]. The latter has even been used in the computer game 'Spore' to control creatures created by the player with unpredictable arbitrary kinematic chains. This flexibility is unique to sampling-based approaches and is the basis of the control algorithms developed in this thesis.

## 2.5 Control

We make a distinction between *dynamic* control and *kinematic* control. With *dynamic* control, we refer to control algorithms that directly compute the forces for the motors of the robot, and thus take the dynamics of the robot into account. With *kinematic* control, we refer to algorithms that result in static *kinematic configurations* that are used as targets for another program that calculates the forces. This thesis concerns only *kinematic control*, allowing us to focus on the kinematic form of the movements and ignore the dynamical aspect, which would require much more detailed physical models of the robot. Extending the methods in this thesis to dynamical control is left for future work.

## 2.6 State of the Art

IK has had a long history of research employing varied methods; from direct solvers, to iterated optimisation and online learning [Paul and Shimano, 1979; Angeles, 1985; Goldenberg et al., 1985; Tsai and Morgan, 1985; Atkeson et al., 1997; Ijspeert et al., 2002]. The traditional approach to IK is to explicitly find $f^{-1}$ by constraining the problem until a *closed form* solution is obtained [Chang, 1987; Hemami, 1987; Kauschke, 1996]. This approach is very fast, but requires careful engineering and is restrictive in the constraints that can be used. Numerical approaches indirectly find $f^{-1}$ by iterative optimisation using the gradient $\nabla f$, often by calculating the pseudo-inverse or transpose of the Jacobian [Wolovich and Elliott, 1984; Zohdy et al., 1989]. Recent work applies such methods to humanoid robots [Baerlocher and Boulic, 2004] and adds an efficient way to handle prioritised hard constraints. However, the constraints have to be set up carefully and the dimensionality of the system has to be controlled to compute the inverse efficiently. Also, Jacobian methods only look at the local slope and don't take the curvature and non-linearity of $f$ into account.

Current robotic control systems are hard to set up, relying on strict formulations of kinematics, that put limits on the constraints that can be used to design movements [Hemami, 1987; Kauschke, 1996; Hsu et al., 1997; LaValle, 2006; Choset et al., 2005; Hsu et al., 2006]. To allow for more freedom in the constraints, sampling methods have been used [Dutra et al., 2008; Hecker et al., 2008; Courty and Arnaud,

2008; Graça Marcos et al., 2009]. These methods only use the forward model $f$ for evaluation, relieving constraints like continuity and smoothness on $f$, but they do not scale well to high-dimensional kinematic models.

## 2.7 Concluding Remarks

As we have seen, controlling a robot is a complex problem and an ongoing field of research. Most of the algorithms in the field solve the problem by constraining it until it can be solved analytically or using Jacobian-based approaches.

Relatively new in the field are sampling-based approaches which use more computation power but allow for a lot more freedom in defining the movements of the robot. We will continue in this direction and introduce sampling-based algorithms that allow for flexible control of humanoid robots. The higher use of computational power is offset by the flexibility the method gives. Also, we develop ways to reuse previous calculations, parallelise the algorithm and provide an online version, to mitigate the computational burden.

Chapter 3 introduces Natural Gradient Inverse Kinematics (NGIK), the basis of our approaches to IK. Chapter 4 applies NGIK to build reusable task-relevant roadmaps (TRM) to plan movements on the iCub. A parallelised algorithm to build TRMs is shown in Chapter 5. Finally, an online adaptation of NGIK, called Natural Gradient Control, is presented in Chapter 6.

# Chapter 3

# Natural Gradient Inverse Kinematic

In Chapter 2 we introduced the control problem. We notice that Jacobian based control algorithms lack flexibly definable constraints. The most flexible algorithms are sampling-based, but they require many forward kinematics evaluations and are thus computationally expensive [Dutra et al., 2008; Courty and Arnaud, 2008; Hecker et al., 2008].

In this chapter, we develop Natural Gradient Inverse Kinematics (NGIK), a sampling-based approach to inverse kinematics. NGIK makes the computational burden manageable by using the powerful sampling-based optimiser called Natural Evolution Strategies, developed by Wierstra et al., 2014 and Glasmachers et al., 2010a. In contrast to most IK methods, this algorithm performs inverse kinematics without the use of gradients or inverses of the forward kinematics function $f$. The result is a flexible algorithm applicable to a wide range of tasks and can be used on complex robots like a humanoid.

The following sections introduce the concepts used for NGIK, and develops the algorithm itself. The following three chapters extend our work on NGIK, by applying NGIK for planning, speed up the algorithm using parallelisation, and allow NGIK to be used for direct control. The result of this work was published in 2013 [**M. F. Stollenga** et al., 2013b] and parts of this chapter are derived from it.

## 3.1 Natural Gradient

The natural gradient was introduced by [Amari, 1987] as the direction of steepest descent on a curved Riemannian manifold and is a robust method for optimising difficult cost functions [Amari, 1998; Samir et al., 2012]. The approach has close

relations to the analysis of the geometry of Ricatti equations [Ammar and Martin, 1986] which has similar applications to optimisation methods [Edelman et al., 1998; Lundström and Eldén, 2002; Helmke et al., 2007].

Given is a function $f(x) \to \mathbb{R}$, where $x \in X \subset \mathbb{R}^n$ and $n$ is the dimensionality of the space $X$. The gradient $\nabla_x f(x)$ of a function $f(x)$ is a vector in the direction of steepest increase, starting from $x$, with an amplitude equal to the speed of this increase. In other words, it is the direction that maximises $\nabla_x f(x) = \max_v f(x+\epsilon v)$, where $\epsilon$ is an infinitesimal value. This assumes that $x \in X$ is a Euclidian space, where the distance between two points $x$ and $x'$ in $X$ is measured by the Euclidian distance $d(x, x') = \|x - x'\|_2$.

However, this might not be an appropriate distance metric, especially when talking about the kinematics of a robot where the search space $X$ is formed by the joint parameters. For example, we typically have correlations between our dimensions since the turning of different joints might contribute to the same movement of an end-effector. Also, different joints might be parameterised in different precisions; one could be denoted in radians while another is denoted in degrees. For the regular gradient, changing the parameterisations results in (big) changes in the gradient. However, the natural gradient is invariant to such linear transformations of the search-space!

**Distance on a curved space** Given two vectors $x \in \mathbb{R}^D$ and $x' \in \mathbb{R}^D$, of dimension $D$, let $dx = x - x'$ be a vector connecting them. As the vector $dx$ gets infinitesimally small, we can measure its distance using a linear function, describing the curvature. For an Euclidian space, the length of $dx$ is simply $|dx|^2 = dx\, dx^T$. For a curved space, however, the length of $dx$ is

$$|dx|^2 = \Sigma_{ij} g_{ij}(x)\, dx_i\, dx_j = dx\, G(x)\, dx^T \tag{3.1}$$

where $G(x)$ is a $D \times D$ positive definite matrix, expressing the scaling and correlations between dimensions within space $X$, called a *metric*. Notice that $G(x)$ depends on $x$, since curvature can change over the space. The regular Euclidian distance is a special case of this metric, when $G(x) = I$, where $I$ is the identity matrix:

$$\mathbb{I}_{ij}(x) = \begin{cases} 1, & \text{if } i \equiv j \\ 0, & \text{else.} \end{cases} \tag{3.2}$$

Given a gradient $\nabla_x f(x)$ and a metric $G(x)$, the natural gradient [Amari, 1998] is given by :

$$\tilde{\nabla}_x f(x) = G^{-1}(x) \nabla_x f(x) \tag{3.3}$$

where $\tilde{\nabla}$ denotes the natural gradient, and $G^{-1}$ is the inverse of the curvature matrix.

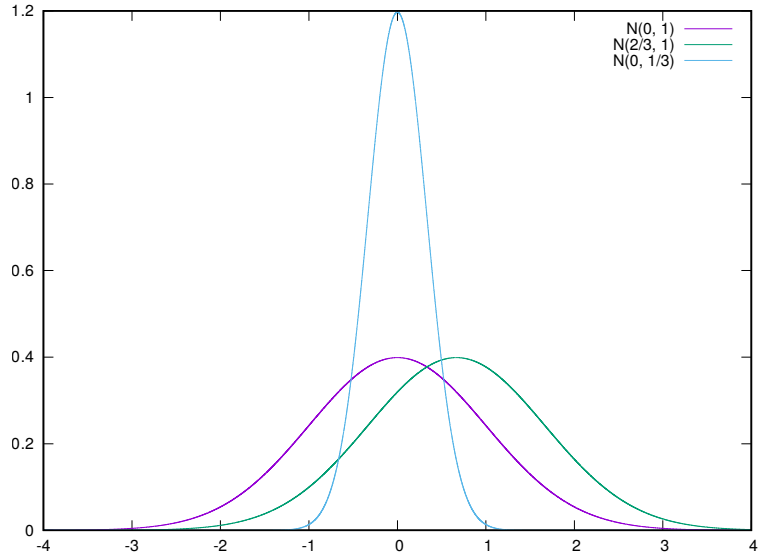



**Figure 3.1. Distances Between Distributions** Three normal distributions with different parameters are shown. One regular distribution $\mathcal{N}(0, 1)$, one thin distribution $\mathcal{N}(0, \frac{1}{3})$, and one shifted distribution $\mathcal{N}(\frac{2}{3}, 1)$. The overlap between the shifted and regular distribution is quite big while the very thin distribution has a small overlap. However, when measured with a Euclidian distance in the parameter space, the distances of these distributions to the regular distribution are equal. This motivates a better distance metric that more appropriately measures the distance between distributions.

**Distance Metric for Distributions** When is a Euclidian metric insufficient to adequately describe your parameter space? In this thesis, the natural gradient technique is used in relation to probability distributions. Say we have a one-dimensional normal distribution $\mathcal{N}(\mu, \sigma)$, where $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$. The distances in parameters space of $\mu, \sigma$ are not well described by a Euclidian metric. For example, in Figure 3.1, three distributions are shown, taken from different points in our parameter space. Distribution 1: $\mathcal{N}(0, 1)$, 2: $\mathcal{N}(0, \frac{1}{3})$, and 3: $\mathcal{N}(\frac{2}{3}, 1)$.

Using an Euclidian distance, the difference between distribution 1 and distribution 2 is as big as the difference between 1 and 3. If distribution 1 was the true distribution, would it be 'more wrong' to guess distribution 2 than distribution 3? Distribution 2 is peaked around $x = 0$ and puts almost no weight on samples outside this value, while distribution 3 has some decent overlap with distribution 1, but it's shifted. What would be a natural measure to compare distributions?

**KL divergence** A popular measure to compare two distributions $p(x)$ and $q(x)$ is the Kullback-Leibler divergence (KL-divergence;[Kullback and Leibler, 1951]), which uses an information-theoretic approach to measuring the distance between distributions. It can be explained as follows: Assume you want to transfer samples taken from a distribution $p(x)$ to a receiver. To do this, a 'code-book' is made to encode each sample $x' \sim p(x)$ in finite bits of information to send to the receiver. The optimal code-book uses fewer bits for samples that occur often, and many bits for rare cases, such that the average number of bits is minimised. However, what happens if $x$ is actually distributed according to a different distribution $q(x)$? There will be inefficiency in sending these samples! The average loss of bits per sample is the KL-divergence.

Given a distribution $p(x)$, an optimal code-book uses $-\log p(x)$ bits to encode $x$ for a given distribution $p(x)$[1]. If $x$ is very common, say it occurs 10% of the time $p(x = x') = 10\%$, then fewer bits are used, $-\log(.1) \approx 2.3$ bits. If a sample is uncommon and occurs 0.1% of the time $p(x = x'') = .1\%$, then more bits are used: $-\log(.001) \approx 6.9$ bits. This approach minimises the average bits used for a distribution $p(x)$. But when we make a code-book for distribution $q(x)$, while the true distribution is $p(x)$, we can measure the expected inefficiency in bits per sample as follows:

$$\mathrm{KL}(p|q) = \int_x p(x)[-\log q(x) - -\log p(x)] = \int_x p(x) \log \frac{p(x)}{q(x)} \tag{3.4}$$

where $p(x)$ is a true distribution, and $q(x)$ the distribution the code-book is based on. The right side, $\log \frac{p(x)}{q(x)}$, measures the number of bits wasted, while the weighted integral on the left side calculates the expectation under $p(x)$. Note that the KL-divergence is always non-negative and is an asymmetric measure. It is also undefined if distribution $q(x) = 0$ in places where $p(x) > 0$ since it results in a divide by zero.

If we use the KL divergence as our metric, the distances between the distributions in Figure 3.1 seem more natural. The difference between the thin distribution and the standard distribution is bigger $\mathrm{KL}(\mathscr{U}(0, \frac{1}{3})|\mathscr{U}(0, 1)) \approx 0.654$ than the shifted distribution with more overlap $\mathrm{KL}(\mathscr{U}(\frac{2}{3}, 1)|\mathscr{U}(0, 1)) \approx 0.222$.

Using the KL divergence as a distance metric results in the following metric $G$: If distribution $p(x)$ is a parameterised distribution $p(x \mid \theta)$, with parameters $\theta$, the metric for this statistical model turns out to be the Fisher information metric, which

[1] Since samples from the distribution are in $\mathbb{R}$ they have in principle (uncountably) infinite precision and need infinite bits to be encoded. However, we can encode the samples to an arbitrary small (finite) error $\epsilon$ with a number of bits proportional to $-\log p(x)$, and thus this measure is used.

is the only invariant metric for a statistic model [Campbell, 1985; Amari, 1987]:

$$g_{ij}(w) = E\left[\frac{\partial \log p(x \mid \theta)}{\partial \theta_i}\frac{\partial \log p(x \mid \theta)}{\partial \theta_j}\right]. \tag{3.5}$$

This metric provides a natural metric that has been shown to help many learning algorithms [Amari, 1998]. In the next section, we introduce Natural Evolution Strategies; a robust natural gradient-based optimisation method that is used throughout the thesis.

## 3.2 Natural Evolution Strategies

The natural gradient forms the basis of Natural Evolution Strategies (NES), a sampling-based optimisation method to minimise a cost function, developed by Wierstra et al., 2008; Wierstra et al., 2014. NES uses a parameterised exponential probability distribution $p(x \mid \theta)$, parameterised by $\theta$, to represent its current search state.

As of yet, NES has been applied to multivariate Gaussian distributions and the heavy-tailed Cauchy distribution [Schaul et al., 2011b]. A longer tailed distribution will have more samples that are far removed from the centre of the distribution. Although this might help when the current search is far from a solution, it also adds inefficiency and noise to our samples with many unhelpful samples far from the current centre. Thus, the family of multivariate Gaussians $\mathcal{N}(x \mid \mu, \Sigma)$ is chosen, such that $\theta = (\mu, \Sigma)$, and we rely on good initialisation to avoid ending up in bad locations in search-space.

Suppose $c(x)$ is a cost function on $x \in \mathbb{R}^n$. Then define

$$J_c(\theta) = \mathbb{E}_{p(x|\theta)}c(x) = \int c(x)\, p(x \mid \theta)\, dx \tag{3.6}$$

to be the *expected cost* under a probability distribution $\{p(x \mid \theta) \mid \theta \in \Theta\}$. Then optimising $J_c$ with respect to the parameter space is equivalent to optimising $c$, since the optimal set of parameters $(\hat{\mu}, \hat{\Sigma})$ focuses all probability weight on the optimal point $\hat{x}$, so that $\hat{\mu} = \hat{x}$, $\hat{\Sigma} \to 0$, and $J_c(\hat{\theta}) \to c(\hat{x})$.

NES tries to find the natural gradient of the cost function $J_c$ towards the distribution

parameters. First, the regular gradient is estimated, which can be rewritten as:

$$\nabla_\theta J_c(\theta) = \nabla_\theta \int c(x)p(x \mid \theta)\, dx = \tag{3.7}$$

$$\int [c(x)\nabla_\theta \log p(x \mid \theta)]\, p(x \mid \theta)\, dx = \tag{3.8}$$

$$\mathbb{E}_{p(x|\theta)}[c(x)\nabla_\theta \log p(x \mid \theta)] \tag{3.9}$$

NES estimates the gradient by sampling $K$ samples from the distribution to estimate the expected value:

$$\mathbb{E}_{p(x|\theta)}[c(x)\nabla_\theta \log p(x \mid \theta)] \approx \tag{3.10}$$

$$\frac{1}{K}\Sigma_i^K c(x)\nabla_\theta \log p(x_i \mid \theta) \text{ with } x_i \sim p(x \mid \theta) \tag{3.11}$$

The samples together are called a *population*, referring to the evolutionary analogues of this method.

Then the inverse of the Fischer information matrix $F^{-1}$ is estimated by reusing the gradients $\nabla_\theta \log p(x \mid \theta)$ to give the natural gradient (see [Wierstra et al., 2014] for more detail), resulting in the natural gradient:

$$\tilde{\nabla}_\theta J_c(\theta) = F^{-1}\nabla_\theta J_c(\theta) \tag{3.12}$$

**Exponential NES** NES was improved with exponential NES (xNES), developed by Glasmachers et al., 2010b), which uses an improved exponential parameterisation of the covariance matrix $\Sigma$. Since the covariance matrix $\Sigma$ needs to be positive definite, $\Sigma$ is represented by a Cholesky decomposition $\Sigma = AA^T$, where $A$ is further decomposed in $A = \sigma B$, where $\sigma$ is a scalar value representing the 'width' of the distribution, and $B$ a matrix representing the shape with $det(B) = 1$. Additionally, xNES performs its computations in a natural coordinate space tangent to the parameter manifold (for more details see [Glasmachers et al., 2010b]):

$$(\delta, M) \rightarrow (\mu + A\delta, A\exp(\frac{1}{2}M)) \tag{3.13}$$

where $\delta$ and $M$ for the coordinates. The resulting algorithm is shown in Algorithm 1. The algorithm iteratively updates the search distribution, until the width of the covariance matrix $\sigma$ has become smaller than a threshold value $\lambda_{stop}$ and thus the search has converged. It returns the centre of the search distribution as the result.

**Algorithm 1** XNES

Exponential NES estimates the natural gradient towards a distribution from samples from that distribution.

```
function XNES(starting distribution {μ, Σ}, learning rates η_μ, η_σ, η_B, and stopping
parameter δ_stop)
    A = CholeskyDecomposition(Σ)                          ▷ Initialise variables
    σ ← ∛|det(A)|
    B ← A/σ
    while σ > λ_stop do
        for i ∈ {1 ··· K} do                              ▷ Sample population of K samples
            z_i ← 𝒩(0, 𝕀)
            x_i ← μ + σBz_i
        end for
        sort ({z_i, x_i}) with respect to c(x_i)
        assign u_i to x_i based on sorting                ▷ Assign utilities
        G_δ ← Σ_i u_i z_i
        G_M ← Σ_i u_i (z_i z_i^T − 𝕀)
        G_σ ← tr(G_M)/d
        G_B ← G_M − G_σ 𝕀
        μ ← μ + η_μ · σB · G_δ                            ▷ Update mean
        σ ← σ exp η_σ/2 · G_σ                             ▷ Update σ
        B ← B exp η_B/2 · G_B                             ▷ Update B
    end while
    return μ                                              ▷ Return centre of distribution as optimum.
end function
```

**Fitness Shaping** The slopes of most cost functions flatten as they come closer to their goals. This can result in a undesired reduction of the norm of the gradient. To combat this, xNES applies *fitness shaping* by defining a set of utilities:

$$u_i = \max(0, 1 - 2 \cdot \frac{i}{K}) \tag{3.14}$$

The population $(x_k)_{k=1}^{K}$ is ranked according to fitness, and their fitness is replaced by $u_i$ where $i$ is the rank of $x_k$ in the population. The *utilities* $u_i$ are set to $0$ for the bottom half of the population by rank, and for the top half $u_i$ increases linearly up to $1$ for the highest ranking member. The result is that xNES is invariant under order-preserving operations on the fitness function, and thus adds robustness to the algorithm. In equations below, the populations are assumed to be ordered by their rank, so that $x_i$ is the $i^{th}$-ranked member of the population.

**Algorithm 2** SNES
Separable NES only represents the diagonal values of its search distribution, thus losing information but gaining efficiency.

**function** SNES(starting distribution $\{\mu, \Sigma\}$, learning rates $\eta_\mu, \eta_\Sigma$, and stopping parameter $\delta_{stop}$)
  **while** $\|\Sigma\|_2 > \delta_{stop}$ **do**
    **for** $k \leftarrow 1 \cdots K$ **do**
      $s_k \sim \mathcal{N}(\vec{0}, I)$
      $x_k \leftarrow \mu + \Sigma \cdot s_k$
    **end for**
    Sort $\{(s_k, x_k)\}$ with respect to $c(z_k)$
    assign $u_i$ to $x_i$ based on sorting ▷ Assign utilities
    $\nabla_\mu J \leftarrow \sum_{k=1}^{p} u_k \cdot s_k$
    $\nabla_\Sigma J \leftarrow \sum_{k=1}^{p} u_k \cdot (s_k^2 - 1)$
    $\mu \xleftarrow{\eta}_{\mu} \cdot \Sigma \cdot \nabla_\mu J$ ▷ Update mean $\mu$
    $\Sigma \xleftarrow{\exp} (\eta_\sigma / 2 \cdot \nabla_\Sigma J)$ ▷ Update Covariance $\Sigma$
  **end while**
  **return** $\mu$ ▷ Return mean $\mu$ as optimum.
**end function**

**Separable NES** xNES requires the calculation of the inverse of a matrix of size $n \times n$, where $n$ is the dimensionality of the search space (shown in Algorithm 2). Since the complexity of this operation is approximately $(O)(n^3)$ this becomes inadequate for high-dimensional spaces. For this case, there is Separable NES (SNES; [Schaul

et al., 2011a]), which uses a normal distribution with only diagonal entries in the covariance matrix. This ignores any correlations between dimensions and thus loses a lot of information, but it results in an $\mathcal{O}(n)$ inverse calculation allowing it to scale to many dimensions. SNES is used in Chapter 8 to train a complex policy.

## 3.3 Kinematics Cost Function

NES provides us with robust optimisation algorithms to minimise a function. How do we define such a function for IK? We define a cost function $c(q) \to \mathbb{R}^+$, where $q \in \mathbb{R}^J$ are the joint parameters of the robot, $J$ is the number of joints and $c$ results in a scalar cost that is to be minimised. This function is formed by a weighted sum of several cost functions:

$$c(q) = \sum_{i}^{C} w_i c_i(q) \tag{3.15}$$

where $C$ is the number of cost functions and $w_i$ is a scalar weight for the cost function $c_i$.

Because of the linear transformation invariance of NES with fitness shaping, we can design our cost function very flexibly. Where most IK methods pose strong assumptions on the cost functions used, especially with the requirement of (smooth) differentiability, NES can also incorporate discrete costs. These cost functions can range from counting the number of collisions (a function without a derivative) to the Euclidean distance of the hand towards the desired position. The cost functions used in this thesis are:

**Home Pose Cost** This function biases the search to a certain kinematic configuration. In case multiple solutions exist for a given point in task-space, the home posture allows to find a unique solution. It also increases similarity in joint parameters between nearby points in task-space.

$$c_{home} = \|q - q^{home}\|_2 \tag{3.16}$$

where $q^{home}$ is a home posture and $\| \cdot \|_2$ is the $\ell^2$-norm.

**Collision Cost** This function penalises collisions. xNES is able to estimate a gradient over collisions using the number of collisions. This number usually increases

with deeper penetration.

$$c_{collision} = |collisions| \tag{3.17}$$

**Position Cost** This function attracts a body part to a fixed position or to another body part. The target can be specified as an area or volume in task-space, rather than a point.

$$c_{position} = \|v_{bodypart} - v^*\|_2 \tag{3.18}$$

where $v^*$ is the desired position.

**Orientation Cost** This function controls the orientation of a body part. The desired orientation can be either a fixed orientation in task-space or the orientation of another body part.

$$c_{orientation} = (\vec{u}_{base}^T R_{bodypart} \vec{u}_{desired} + 1)/2 \tag{3.19}$$

where $\vec{u}_{base}$ is the unit base vector that has to point to the desired direction $\vec{u}_{desired}$ after being rotated over the transformation matrix $R_{bodypart}$ of the body part.

**Aiming Cost** This function aims a body part toward another body part or a certain position in task-space. Optionally it also keeps the end effector at a fixed distance to this point or body part.

$$c_{pointing} = (\vec{u}_{base}^T R_{bodypart} \frac{v_{target} - v_{bodypart}}{\|v_{target} - v_{bodypart}\|_2} + 1)/2 \tag{3.20}$$

**Repelling Cost** This function repels two body parts away from each other. Repulsion can optionally start after a minimum distance and is helpful for finding non-colliding postures and to increase mobility in the map.

$$c_{repel} = min(0, d - \|v_{bodypart1} - v_{bodypart2}\|_2)/d \tag{3.21}$$

## 3.4 NGIK Algorithm

The NGIK algorithm is shown in Algorithm 3. It takes a cost function, an initial distribution and a population size and returns a pose if successful, and a 'fail' if the

resulting pose was not following hard constraints. The latter can be the case if no solution can be found that does not collide with something.

Initialisation is important for good results. Typically the mean of the distribution is set to a home pose that forms a good beginning for the search $\mu \leftarrow$ *home pose*. The covariance matrix is set such that it encompasses most of the joint configuration space $\Sigma \leftarrow \sigma \cdot \mathbb{I}$, by setting $\sigma$ to a high enough value. Setting the population to a high enough size to still allow for good results is a matter of empirical tuning. A higher $K$ results in more robust and better solutions, but comes at a computational cost. Often values of $K = 100$ and $K = 200$ seem to provide good results on the iCub, but these results depend on the complexity of the cost function $c$.

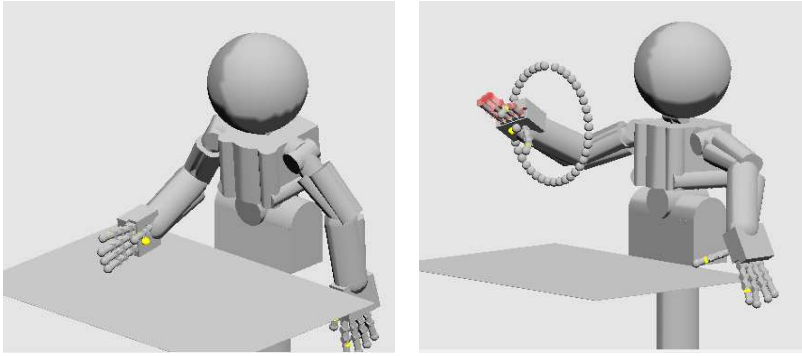
**Algorithm 3** NGIK
NGIK uses XNES to find optimal joint parameters $q$ for a given cost function $c$.

```
function NGIK(cost function c, starting distribution {μ, Σ}, population size K)
    q ← XNES(c, μ, Σ, K)
    if Check(q) then                    ▷ Check if hard constraints are satisfied.
        return q
    else
        return failed
    end if
end function
```

## 3.5 Experiments

We compared the performance of NGIK against Metropolis-Hastings (MH; [Metropolis et al., 1953]) sampling, two versions of Sequential Importance Resampling (SIR; [Kitagawa, 1996; Gordon et al., 1993]), and the simplex optimisation algorithm [Lagarias et al., 1998]. There former two were discussed in Section 2.4. The latter is an optimisation algorithm designed for convex problems and serves as an example of an unsuited optimiser.

To make the comparison fair, we use simple Gaussian proposal distributions for SIR and MC instead of tailored distributions that use information about the robot [Courty and Arnaud, 2008], as NGIK also doesn't assume extra knowledge on the task at hand. We don't compare NGIK against the related algorithm Covariance Matrix Adaptation Evolution Strategy (CMA-ES,[Hansen et al., 2003]) as it is already shown that performance differences are small [Glasmachers et al., 2010a] with xNES often beating CMA-ES on complex problems. We also don't compare to Jacobian



(1) (2)

**Figure 3.2. Two Postures** The two postures used to compare the different optimisation algorithms. The left posture is constrained to hold the left hand behind the table and the right hand above it. The right posture is a challenging grabbing posture through a loop using the thumb and index finger. The results are shown in Table 3.1

inverse/transpose based methods, as these can't handle the cost functions we use and thus simply are not applicable in this framework.[2]

SIR is the method closest to the capabilities of NGIK, as it uses a population of samples for optimisation. Every sample has a corresponding weight, which is updated as new samples are drawn from previous samples. SIR resamples when the effective weights are below a threshold, in which case a new population is sampled according to the weights, and the weights are reset. We found that using a threshold of 75% of the number of particles works well. We also compare a direct version of SIR that resamples on every iteration, which we call SIR direct (SIRD). The optimal standard deviation of the initial search distributions was determined experimentally.

The SIMPLEX optimisation algorithm is meant for convex optimisation problems and functions as an example of an algorithm that is not suited for the optimisation problem at hand.

The algorithms are used to optimise two challenging postures shown in Figure 3.2. The left posture (Figure 3.2-1) requires the left hand to be behind the table and the right hand to be in front of the robot. The right posture (Figure 3.2-2) is more difficult and requires a reaching posture through a loop.

The results are shown in Table 3.1, where we calculated the average best fitness value after 30 runs, and the corresponding standard deviation of the mean. The population size for SIR, SIRD and XNES was 300; a relatively large population size to prevent local optima. The standard deviation of the proposal distributions was

[2]In Chapter 6 an online version of NGIK is compared closely to a Jacobian method, showing that following the natural gradient from NGIK results in more efficient paths.

**Table 3.1. Comparison of NGIK** Comparison of NGIK versus different optimisation algorithms. The postures are shown in Figure 3.2.

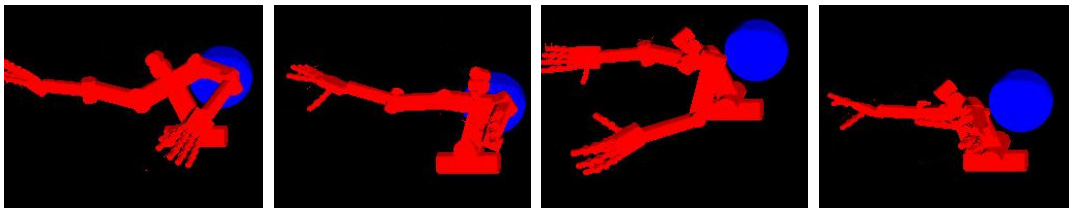
| Algorithm | Final Cost Posture (1) | Nr. Evaluations |
|---|---|---|
| NGIK | **0.1122** $\pm$ 0.0269 | 199290 $\pm$11444 |
| SIR | 0.5068 $\pm$ 0.0342 | 144920 $\pm$16172 |
| SIRD | 0.4492 $\pm$ 0.0356 | 158190 $\pm$24613 |
| MH | 13.5507 $\pm$ 5.7343 | 80432 $\pm$17200 |
| SIMPLEX | 111.113 $\pm$ 3.8616 | 420090 $\pm$0 |

| | Final Cost Posture (2) | Nr. Evaluations |
|---|---|---|
| NGIK | **0.1765** $\pm$0.0207 | 239040 $\pm$3824 |
| SIR | 1.3167 $\pm$0.0278 | 9260 $\pm$2181 |
| SIRD | 1.2835 $\pm$0.0272 | 15330 $\pm$3950 |
| MH | 1.4934 $\pm$0.0352 | 11320 $\pm$3009 |
| SIMPLEX | 1.661 $\pm$0.019 | 420090 $\pm$0 |

$\sigma = 0.005$ for MC, SIR and SIRD. NES has an initial standard deviation of $\sigma = .4$, which is higher than for the other methods as it can adjust the distribution dynamically. The other algorithms don't perform well unless the deviations are relatively small. All algorithms keep track of the best-encountered fitness and the time it was encountered. If the best fitness doesn't improve for a large number of evaluations, the algorithm is stopped. We store the number of evaluations at the time the best fitness was encountered as the running time.

NGIK outperforms the other algorithms. The SIMPLEX algorithm has the lowest performance as it assumes a convex problem and gets stuck early in the optimisation. SIR and SIRD have even performance, although their performance is significantly lower than NES, as it cannot adjust the proposal distribution. MC is similar to the SIR algorithm with a population of $1$, which accounts for its bad performance in such a high-dimensional search space. For posture (2), both SIR and MC find their best result after relatively few iterations and have trouble improving it afterwards, resulting in the low reported running times.

### 3.5.1 Pixel Counting Cost



(a) Reach (b) Reach & Minimize red (c) Reach & Maximize Blue (d) Reach & Minimize Red & Maximize Blue

**Figure 3.3. Pixel Counting Cost** The results of applying the complex 'Pixel Counting' cost function are shown. This cost counts the blue and red pixels in rendered frontal images shown here. The iCub is constrained to reach to the right, but then the 'pixel counting' cost function wants to either minimise red, maximise blue or do both. These result of minimising red pixels it a minimisation of the robots' frontal profile. The result of maximising blue pixels it minimising the obstruction of the blue ball. Such cost functions are not applicable to Jacobian-based controllers but form no problem for our framework.

One of the main advantages of NGIK is the possibility to use complex cost functions; even undifferentiable ones that cannot be used in Jacobian-based methods. To illustrate this, we create a new *'pixel counting' cost function* which works as follows: It renders an image of the iCub model from a given perspective, counts the pixels of certain colours, and then either minimises or maximises them. The cost is normalised to the range of $0$ to $1$. Cost functions of this form are intractable for a Jacobian-based controller; how would one define a differentiable cost function from the number of counted pixels to the joint parameters?

In the experiment, a blue ball is placed behind the iCub (in the model), and a subset of the iCub is drawn in red. The images are rendered at $400 \times 300$ pixels and are taken from the front of the iCub. The iCub starts with three costs: 1) a *Position* cost to reach to the right with the right hand, 2) a *Home Pose* cost to guide the search, and 3) a *Collision* cost to avoid collisions. Then the *Pixel Counting* cost is added. Then Natural Gradient Control (NGC), an online NGIK based method explained in Chapter 6, was run until the cost did not improve for several iterations.

We tried four different configurations:

1. **Normal:** Just the three basic cost functions are used.

2. **Minimise Profile:** A view cost is added that minimises the amount of red, i.e. the iCub wants to minimise its frontal profile. This could be useful to move through corridors or minimise the chance of being seen.

3. **Maximise Ball:** A view cost is added to maximise the amount of blue, i.e. to show as much of the blue ball as possible. This is useful when a humanoid has to perform a task while not obscuring an object from a user's perspective (such as the TV in the introductory example).

4. **Combined:** The second and third cost functions are both added, minimising the frontal profile, while maximising the view of the ball.

The results of this experiment are shown in Figure 3.3. Figure 3.3-a shows the reach behaviour as a result of optimising the cost functions. The iCub simply tries to reach to the right. The ball is clearly occluded by the body. In Figure 3.3-b a Pixel Counting cost is added to minimise the amount of red. Since the iCub is drawn in red, this results in the iCub minimising its profile by keeping its left hand pointed towards the viewer, minimising its surface. No effort was taken to avoid occluding the blue ball. In Figure 3.3-c a Pixel Counting constraint was added to maximise the amount of blue, i.e. to avoid occluding the blue ball. The result is that the iCub turns to the right so it can avoid occlusion while still reaching to the right. No effort was taken to minimise its profile. Finally, in Figure 3.3-d both view constraints are added together, minimising frontal profile and occlusion of the ball. Again the iCub points its left hand towards the camera but also turns to the right to avoid occlusion.

This cost function exemplifies a practical application of the flexibility that our method allows that would not be possible with a Jacobian based approach.

## 3.6 Concluding Remarks

In this chapter, we developed NGIK, a sampling-based approach to inverse kinematics. Although the algorithm is more computationally expensive than its Jacobian-based variants, it is much more flexible and allows for the use of creative cost functions. This serves as an example where increased computational power makes completely new approaches possible.

The following chapters we develop NGIK further. In Chapter 4, NGIK is used to build navigable graphs of parameter configurations related to specific tasks. This algorithm is improved with a parallel version introduced in Chapter 5. Finally in Chapter 6 we develop an online version of NGIK that runs continuously and directly outputs control signals that adapt to changes in its cost function.

# Chapter 4

# Task-Relevant Roadmaps

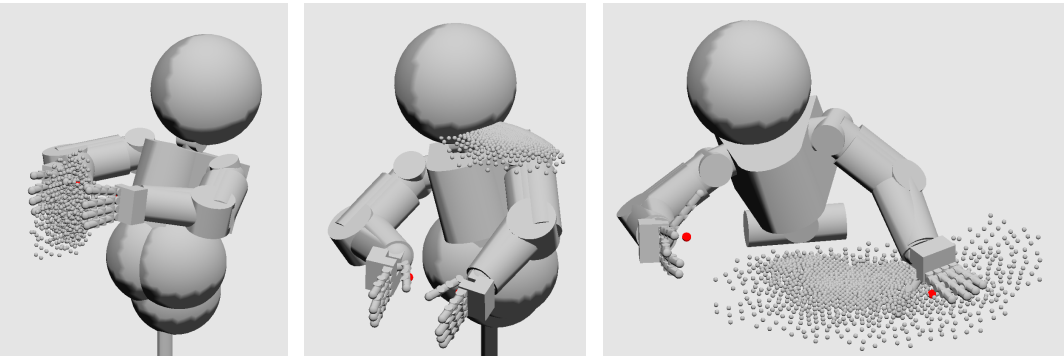


**Figure 4.1. Task-Relevant Roadmaps** Three TRMs are shown. The dots show the task positions corresponding to the parameter vectors in the map. In the left image, the task function is the mean position of the left and right hand, resulting in a lifting behaviour. In the middle image, the task function uses the neck position, resulting in an exploration map. The right image shows a reaching behaviour where the task space is formed by the position of the left hand. The spread of the dots shows TRMs naturally expand the reachable task space.

In the previous chapter, we have described NGIK, which achieves much greater flexibility than Jacobian-based algorithms by using a sampling-based approach, but with a higher but manageable computational cost. NGIK provides a general method for optimising a robot's kinematic configuration by minimising a freely definable cost function. However, finding *one* pose is not enough if you want to make movements, instead, we need *sets of poses* that form a path through the joint parameter space that can be followed.

As such, we want to build a graph of optimised poses, such that we can plan movements towards our goal (such as shown in Figure 4.1). Such graphs are often used in robotic planning [Kauschke, 1996; Hsu et al., 1997; LaValle, 2006; Hsu et al.,

2006]. Graphs also allow efficient reuse of previous computations.

In this chapter, we develop the Task-Relevant Roadmap (TRM; **M. F. Stollenga** et al., 2013b) that applies NGIK in novel ways to build searchable graphs of joint parameters that can be used to generate movements. The general idea is to iteratively optimise kinematic configurations with NGIK to build a set of parameter configurations that lie on a *task manifold*. We will first explain what is meant with a task manifold, then explain how we use NGIK to create a Task-Relevant Roadmap, and finally show the results of this approach.

The result of this work was published in 2013 [**M. F. Stollenga** et al., 2013b] and parts of this chapter are derived from it.

## 4.1 The Task Manifold

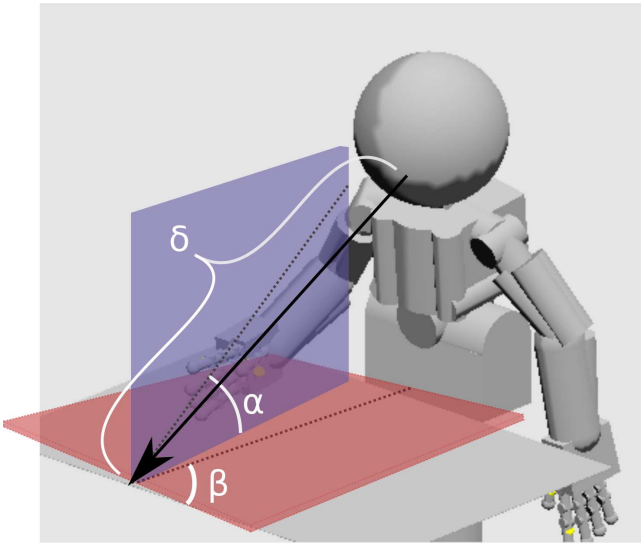



**Figure 4.2. Task-space** This example shows a task space designed to inspect an object from different angles and distances. The task space is formed with the variables $\{\alpha, \beta, \delta\} = t \in \mathbf{T}$, where $\delta$ is the distance to the point, and $\alpha$ and $\beta$ are angles that the head makes with respect to the object.

For a given task, we define the task manifold to be a *set* of relevant joint parameters and corresponding kinematic configurations that are useful for this task. For example, when reaching for an object in 3D space, this set contains kinematic configurations corresponding to *reaches* within this space, which also take into account other constraints such as not colliding with objects and making sure the robot's eyes are looking at the reach position. This set forms a connected subset of joint parameters, denoted

$\mathbf{T}$, within the full space of possible joint parameters $\mathbf{T} \subset \mathbf{Q}$, and thus forms a *task manifold* (see Figure 4.2).

To build these manifolds we need two things:

- Define a Kinematic Cost Function, as described in Chapter 3 that defines our constraints; such as preventing collisions, keeping a hand in a certain orientation of keeping the eyes focused on an object.
- Define our desired degrees of freedom, which will define our orientation over the manifold.

The former defines the constraints and thus reduces the size of the manifold while the latter defines in which direction the manifold grows. To define our degrees of freedom, we introduce the *Task Function*:

$$g(q) \rightarrow \tau; \tau \in \mathbb{R}^G \tag{4.1}$$

which maps joint parameters $q$ to a $G$-dimensional vector which defines our degrees of freedom. For example, it could map to the 3D position of the hand, or to the angle with respect to an object. This freely definable function will form the degrees of freedom with which we control the robot and perform a certain task. Examples of task functions are:

**Position Task Function** The 3D position of a body part, or a subset thereof, can function as map coordinates:

$$g_{position} = v \tag{4.2}$$

**Rotation Task Function** The rotation of a body part is a natural task-space to increase manipulation of a hand. The angle of a directional vector $\vec{u}_{bodypart}$ and a reference vector $\vec{u}_{reference}$ can be calculated:

$$g_{rotation} = \cos^{-1}(\vec{u}_{reference} \cdot \vec{u}_{bodypart}) \tag{4.3}$$

**Distance Task Function** The distance between a body part $v_1$ and another body part or object $v_2$ can, for example, be used to control the distance of the head towards an object it is focusing on.

$$g_{distance} = \|v_1 - v_2\|_2 \tag{4.4}$$

## 4.2 Task-Relevant Roadmaps Algorithm

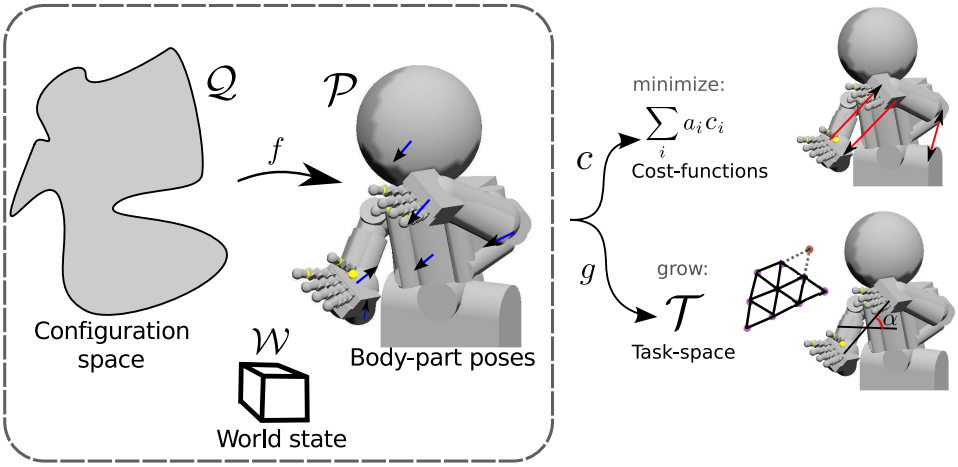



**Figure 4.3. Building Task-Relevant Roadmaps** A function $f$ computes a pose $p \in \mathbf{P}$ from a configuration $q \in \mathbf{Q}$ (left). Several cost functions $c_i$ are defined that are to be minimised and define the desired behaviour. The minimisation is repeated, finding increasingly expanding configurations that cover the desired *task-space* $\mathbf{T}$. The resulting configurations are stored in a map, covering the task-space while minimising the cost functions, thus creating a Task-Relevant Roadmap.

Here we introduce the TRM building algorithm, shown in Figure 4.3. Given a Kinematic Cost Function $c$ and a Task Function $g$ we can set up the TRM building process. We first define the map that will contain the points in the map

$$\{(q_0, \tau_0), (q_1, \tau_1) \cdots (q_N, \tau_N)\} = \mathbf{M} \tag{4.5}$$

where $q_i$ is a set of configuration parameters and $\tau_i = g(q_i)$ the corresponding task vector. The general approach is to repeatedly run NGIK on the defined constraints which minimise the Kinematic Cost Function. However, this would result in the same kinematic configuration every time. Instead, we introduce an extra *Map Building Cost Function*, which makes sure every iteration of NGIK results in new configurations that expand the task manifold.

**Iterative Construction** We define the *Map Building Cost Function* as follows:

$$c_{map} = \sum_{\{q',\tau'\} \in \mathrm{NN}(m,\tau,\mathbf{M})} \underbrace{|d - \|\tau - \tau'\|_2|}_{construction} + \underbrace{\lambda \|q - q'\|_2}_{smoothness} \tag{4.6}$$

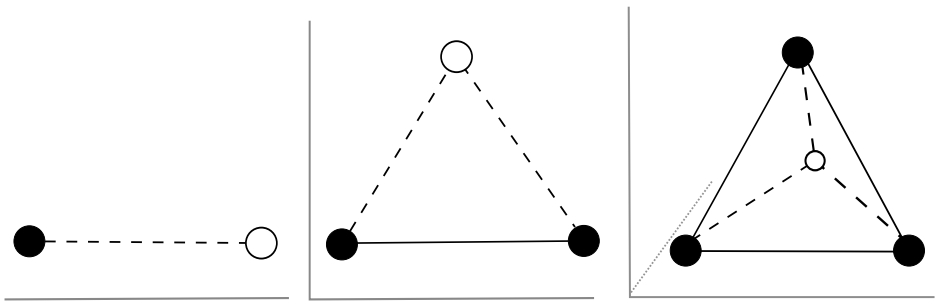


**Figure 4.4. Map Building Cost** The map building cost function is minimised when a predefined distance $d$ to $n$ nearest graph nodes is achieved. The number of nearest neighbours corresponds to the dimensionality of the task-space as shown in the figures.

where $\mathbf{M}$ is the current set of parameter configurations and their corresponding task vector, $NN(m, \tau, \mathbf{M})$ is the set of $m$ nearest neighbours in map $\mathbf{M}$ to task vector $\tau$, and $d$ is a distance constant and $\lambda$ is a weight for the smoothing term.

The *construction* term is minimised when the task vector $\tau$ corresponding to parameter vector $q$ keeps a distance $d$ to the $m$ nearest solutions that are already in map $\mathbf{M}$. The effect of this function is shown in Figure 4.4; the NGIK optimiser will prefer solutions that are close to previous solutions, but differ by distance $d$ in the task-space. If $m$ is chosen to be the number of dimensions of the task-space, the new solutions will form a simplex with previous solutions, and result in stable extensions of the map $\mathbf{M}$. The *smoothing* term acts as a regulariser to keep the parameter vectors close to previous solutions by adding the Euclidian distance between the current and nearest parameter vectors. This keeps consecutive solutions close in parameter vector space such that the resulting movements are smoother.

Whenever a solution is found, it is double checked to see if is not too close to previous solutions. If not, it is stored in the set $\mathbf{M}$, and NGIK is run again. Repeating this process results in a set of solutions $\mathbf{M}$ that span the task-space $T$. Once the process fails $\delta$ times in a row, the process is stopped, where $\delta$ is set by the user.

**Constructing and Planning in a Graph** To allow for path planning in the map, we need to define connections between solutions. We use a simple connection strategy, where each point in the map is connected to $n$ nearest neighbours. The Task-Relevant Roadmap is then formed by adding the edges to the solutions, forming a graph $\mathbf{G} = \langle \mathbf{M}, \mathbf{E} \rangle$ where $\mathbf{E}$ are the edges.

This process can be used to create a variety of maps, shown in Figure 4.6. To plan movements over these maps, a distance function between two solutions needs to be defined. A simple approach is to use the task distance between two solutions

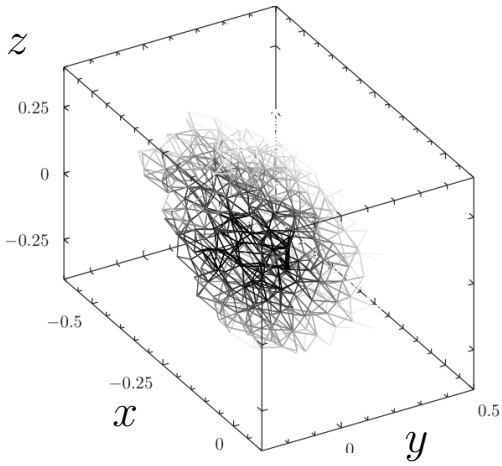



**Figure 4.5. Planning in a TRM** The plot shows one for the TRMs, where each solution is plotted in its task-space position. Each point in the graph represents a pose in which the right hand of the robot is at location $(x, y, z)$. A graph is formed by connecting the points to their nearest neighbours in task-space. By running Dijkstra's algorithm from a goal point, we can find the path distances and shortest paths towards this point. The darker an edge is drawn in this picture, the closer it is to the goal. The result is a graph of which the actionable policy can be extracted at any point in the graph, by following the edge with the lowest distance to the goal.

$d(i, j) = \|\tau_i - \tau_j\|_2$. In that case Dijkstra's algorithm [Dijkstra, 1959] can calculate the distance from every node towards a desired goal, as shown in Figure 4.5. Another approach is to sample each edge at several positions and evaluate the cost function there. This is more computationally intensive but has the benefit of detecting collisions when the environment changes or when a connection between two solutions cuts a corner. In that case applying A* search [Hart et al., 1968] is more efficient to minimise the number of evaluations. This latter approach is used in this work. The algorithm is shown in action in our 2013 AAAI best-student video award winning movie: `http://youtu.be/N6x2e1Zf_yg` [**M. F. Stollenga** et al., 2013a].

## 4.3 TRM Algorithm

The algorithm is described in Figure 4.3. It takes a cost function $c$ and a task function $g$ and builds a TRM $\mathbf{G}$. The map building cost $c_{map}$ is added by the algorithm and does not need to be provided. Note that $c_{map}$ depends on what is in the current map $\mathbf{M}$ and thus changes when something is added to the map.

It is important to set the initial distribution parameters properly. Typically, the mean $\mu$ is set to a home pose that functions as a good starting position for the search.

Then the covariance matrix is set to an identity matrix multiplied by a constant: $\Sigma = \sigma \cdot \mathbb{I}$. By setting $\sigma$ high enough so that it encompasses most of the reachable joint-space around $\mu$ we ensure that even difficult poses can be found. It is also important to set the population size $K$ high enough to get solid optimisation results, but low enough to still finish in a manageable time. Finding a good trade-off between computation time and quality of results for $K$ is done by empirical evaluation.

**Algorithm 4 The TRM building algorithm** The TRM building algorithm repeatedly runs NGIK to find optimised joint parameters to build a map of poses. The map building cost function ensures the map grows. Finally the points are connected and a TRM is returned.

```
function TRM(cost-function c, task function g, population size K, initial distribution parameters μ, Σ, stopping parameter λ_stop, connection parameter λ_connect)
    M ← ∅                                          ▷ Initialise an empty map
    c ← c + c_map        ▷ Add the map build cost function, which depends on M
    fails ← 0
    while fails < λ_stop do
        q ← NGIK(c, μ, Σ, K)
        if Failed(q) then
            fails ← fails + 1                      ▷ If failed, increment counter
            Continue
        else
            fails ← 0
        end if
        M ← M ⊕ {q, g(q)}                          ▷ Add point to map
    end while
    E ← ConnectNearest(M, λ_connect)   ▷ Create edges to connect the map
    G ← {M, E}
    return G
end function
```

## 4.4 Experiments

**Performance and Build Time** The time it takes to build a map depends mainly on two factors: the desired density of postures and the dimensionality of the task-space. The number of postures needed to cover a map scales approximately proportional to $(\frac{l}{d})^n$, where $l$ is the width of the task-space, $d$ the distance between points, and $n$ the dimensionality. For most maps a distance of a few centimetres between postures is enough (if the task-space is defined in Cartesian space). In practice it takes a few

minutes to build a 2D map and tens of minutes for a 3D map. For higher dimensionality the search-time becomes prohibitive because of exponential scaling. We stress that once created, the maps can be re-used in several tasks and building time is not an issue anymore.

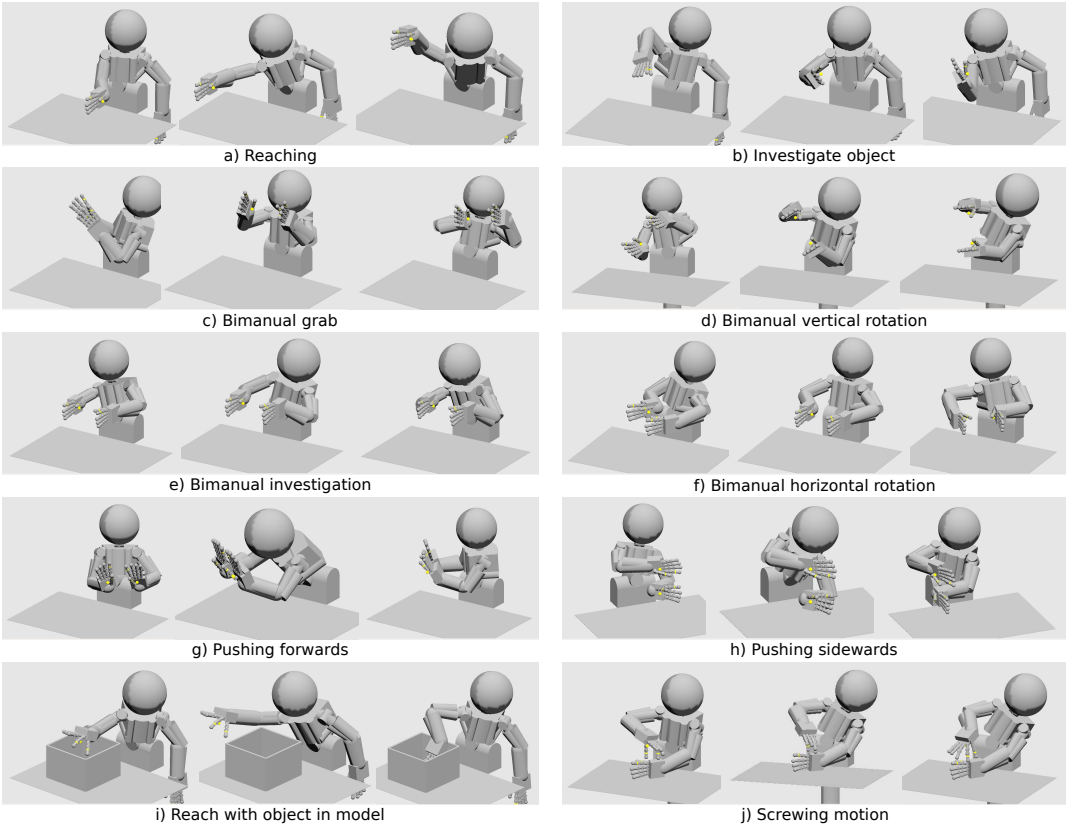



**Figure 4.6. Task-Relevant Roadmaps** Several Task-Relevant Roadmaps that can perform different tasks.

Figure 4.6 shows the result of several maps that were built using TRM for a variety of behaviours. Maps **a**-**j** in Figure 4.6 show several examples of TRMs that are tailored to certain tasks (a video-demonstration is shown at: `http://youtu.be/N6x2e1Zf_yg`). All maps use the collision cost function and home-posture cost function to create collision free postures that look natural.

**Basic Manipulation** Map–**a** was built for a grasping task. As a task-space, we use the position of the left hand. The hand is free to move, but its orientation is kept upright by an orientation cost function. Finally, a pointing constraint is added from the head to the hand to keep the eyes fixed on the moving hand, which would be

required for a typical grasping task. The table is added to the world model as a static object. Map–**b** restricts the position of the right hand, but allows it to rotate freely. By using the angles of the hand relative to the head, the TRM allows the humanoid to investigate an object in its hand from different angles.

**Bi-manual Manipulation** We created several maps that can manipulate a box. Because the box is big, we use bi-manual manipulation. The left hand is constrained to point to the right hand, and vice versa, using pointing cost functions. Map–**c** uses the point between the left and right hand as the task-space, resulting in bi-manual reaches. Map–**e** fixes the position of the hands, and uses the position of the head as a task-space, creating a TRM that can investigate the box between the hands.

If we use the angle between a virtual 'rod', between the left and right hand, and the z- or y-axis as a task-space, we get vertical and horizontal rotating motions respectively (maps **d** and **f**). By constraining the hands to be parallel and fix their relative position, we can get pushing motions. We can create forwards and sideways pushing motions by controlling their orientation, shown in maps **g** and **h**.

**Obstacles** To avoid obstacles we can either use our planning algorithm on a general TRM to plan around it, or put the object in the model while building the TRM. The latter approach is shown in map–**i**, where a box is added, while using the same constraints as the simple reach TRM of map–**a**. This allows us to find difficult solutions that cannot easily be found or specified without putting an object in the model.

To show the complexity of movements that TRMs can express, we also build a map to perform a unscrewing movement. By putting pointing cost functions on the thumb and index finger we can create a grasping posture. By fixing the point between the two fingers and setting the angle of one of the fingers as the task-space we can build a TRM that creates a unscrewing motion, shown in map–**j**.

If we visualise the task positions corresponding to a TRM we see that they naturally expand the reachable task-space, as Figure 4.1 shows.

## 4.5 Concluding Remarks

We developed the TRM algorithm which uses NGIK to build task-relevant searchable graphs. called TRMs. These TRMs can be used for a variety of movements as has been shown in this chapter and in a demonstration movie [**M. F. Stollenga** et al., 2013a]. TRMs have also been found use in other research [Kompella et al., 2015].

From a computational perspective, TRMs allow us to efficiently reuse previous computations, mitigating the increased computational burden posed by the NGIK

optimiser. The building of a TRM can be done offline before the task is done, and thus the build time is less important. However, we cannot build a TRM for every occasion; in fact, every small change to the environment can mean that a TRM is not valid anymore and needs to be rebuilt quickly.

To increase the speed of TRM-building, we will develop a parallel version in the next chapter. In Chapter 6 we will develop a version of NGIK that foregoes building TRMs altogether and functions as a control algorithm in itself.

# Chapter 5

# Parallelising TRM

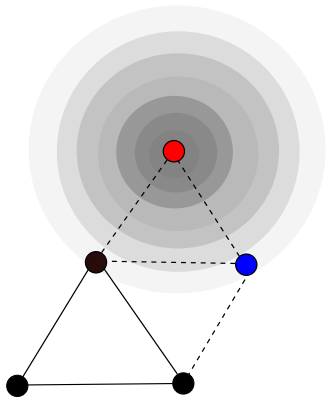


**Figure 5.1. Repulsion Cost** Adding repulsion allows different instances of NGIK to communicate such that they do not find the same solution and waste computation power. The repulsion is implemented by adding a cost function, shown with dark concentric circles around the *red* NGIK instance, and is added to the samples from the *blue* NGIK instance.

The TRM building procedure results in useful maps that can be used for planning paths on task manifolds. However, the building process is not fast enough to be run online and can take minutes for relatively simple tasks. Thus, we want to speed up the process by parallelising the algorithm over several computer cores.

Instead of searching a single solution at a time using NGIK, we want to run many instances of NGIK simultaneously. However, simply running many instances of NGIK will not work, since they will end up finding the same solution, wasting a lot of computation time. Instead the instances need to coordinate the search by communicating with each other. In this chapter, we introduce communication through simple repulsion and show it results in great speedups on computer clusters with up to 32 cores. The work in this chapter was published in 2014 [**M. F. Stollenga** et al.,

**Algorithm 5 SelectProportional** Selects an existing point in a map that will be used to initialise a new NGIK search. The points are taken from the edge of the map, and fail counters ensure that bad starting points are avoided.

```
1:  function SelectProportional(M)
2:      if Empty(map) then
3:          return q_home
4:      end if
5:      for p ∈ M do
6:          scores ← scores ∪ { countN(M, p) + fails[p]}
7:      end for
8:      selection ← ∅
9:      for p ∈ M do
10:         if scores[p] = min_p scores then
11:             selection ← selection ∪ {p}
12:         end if
13:     end for
14:     return SelectRandom(selection)
15: end function
```

2014] and parts of that publication appear in this chapter.

## 5.1 Communication through Repulsion

Parallelising TRM is not a simple matter of distributing the TRM building algorithm (see Algorithm 4) over multiple cores. The independent NGIK instances must be coordinated in order to ensure that they do not duplicate work or interfere with each other in building the map. Therefore, we introduce a *Repulsion Cost Function* for each NGIK instance $i$:

$$c^i_{repel}(q) = \sum_{i \neq j} \max[0, d - \|g(q) - g(\mu_j)\|_2] \tag{5.1}$$

where $d$ is a constant determining the repel distance and $\mu_j$ is the centre of the search distribution of the $j$th NGIK instance. The sum iterates over all other NGIK instances and adds a cost if the sample $q$ comes too close to them in task-space. In effect, this cost pushes the distributions of two NGIK instances away from each other, such that they move into uncharted parts of task-space rather than search in the same region.

Figure 5.1 illustrates two NGIK instances and their distributions simultaneously searching for a new configuration. The red dot and blue dot represent the centres of

distributions from two NGIK instances. The circular shapes represent the *repulsion cost* from the red instance, which is added to the cost of samples from the blue instance. This keeps the blue instance away from red's 'territory'. A similar force works vice versa.

**Stopping Criterion** The Converged?() function in Algorithm 6 is used to decide when an individual NES should stop searching and return its result. It works by maintaining two moving averages, one averaging the lowest costs of the last $20$ populations, and one averaging the last $40$. If the average of the last $20$ iterations is higher than the average of the last $40$, the search is considered to have stagnated, and is terminated. The parameters were determined experimentally to lead to a good trade-off between quality of results and speed.

**Proportional Selection** Since we are incrementally growing a set of solutions, it would be wasteful to restart each instance from the same starting position. Instead, we use the previously found solutions as starting positions for our NGIK instances. However, which points are good starting points? Points at the edge of the solution set are good candidates, but some might be at the edge of reachable space and cannot be extended. Therefore we apply a proportional selection procedure (see Algorithm 5), which selects the points with the minimum score, where the score for solution $p$ is counted as:

$$score(p) = CountN(\mathbf{M}, p) + fails[p] \tag{5.2}$$

where $CountN(\mathbf{M}, p)$ counts the number of neighbours of point $p$ and $fails[p]$ counts the number of failed search attempts starting from this point. From the solutions $p$ that score the lowest, one is selected at random. The result is that points near the edge are preferred since they have fewer neighbours, but if many searches fail they will not get selected anymore.

Algorithm 6 shows Parallelised TRM in pseudo-code. The most important differences from the non-parallel version (Algorithm 4) are lines *4*, where the repulsion constraint is added, *6-10*, where the separate NGIK instances are initialised, and *11-12*, where the NGIK instances are updated in parallel.

## 5.2 Experiments

Parallelisation was implemented using the Threading Building Blocks library, which is responsible for spawning threads and assigning them to each core [Reinders,

**Algorithm 6 Parallel TRM** The parallelised version of Algorithm 4 uses a special selection procedure and a repelling cost function to co-ordinate the search between several NGIK instances.

```
1:  function Parallel-TRM(h, k, P)
2:      i ← 0                                          ▷ initialize an empty map
3:      M ← ∅                                          ▷ initialize an empty set of NES algorithms
4:      c* ← c + α_map c_map                           ▷ add the map-build cost function
5:      c* ← c* + α_repel c_repel                      ▷ add repulsion cost function
6:      while size(S) < min(P, size(M)) do
7:          q_start ← SelectProportional(M)            ▷ start a new NES
8:          Initialize(s, q_start)                     ▷ add it to already active set
9:      end while
10:     for s ∈ S do                                   ▷ This for loop runs in parallel
11:         Update(s)                                  ▷ this is the most computationally intensive step
12:     end for
13:     for s ∈ S do
14:         if Converged(s) then
15:             S ← S \ {s}
16:             q′ ← GetMean(s)
17:             if Check(q′, t′) then
18:                 M ← M ∪ {(q′, t′)}
19:             else
20:                 i ← i + 1
21:             end if
22:         end if
23:     end for
24: end function
```

2007]. The repulsion cost functions rely on fast nearest neighbours searches, thus, we use the Approximate Nearest Neighbours library, which uses KD-trees for efficient search [*libANN: CA library for approximate nearest neighbor searching.*].

The method was evaluated on a reaching task, where the task-space is formed by the $\tau = (x, y, z)$ coordinates at the centre of the right-hand palm of the iCub. A collision cost prevents the iCub from hitting itself, and a homepose cost guides the robot into more natural postures. Two more cost function are added to keep the left hand oriented straight and close to a certain position, to keep it out of the way of the right hand (see Chapter 3 for more details). The distance parameter $d$ was set such that the points in the constructed map are spaced roughly 2.5cm from each other in task-space.

The population size for every NES, $\lambda$, was set to $30$ and the covariance matrices

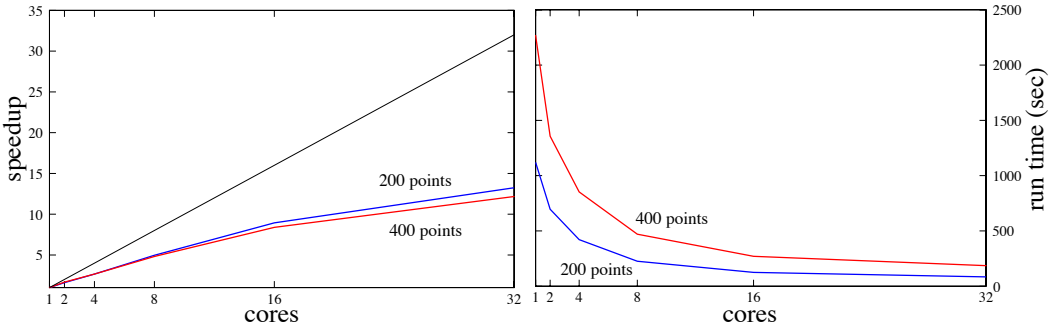



**Figure 5.2. Parallel Speedup** The curves show the speedup achieved for each number of cores when building a map with 200 points (upper curve), and 400 points (lower curve).

were initialised with values of $0.03$ on the diagonal. In other words, NES initially searches using a standard deviation of $0.03$ for every joint. For the very first point of the map, different values are used as there is no other point to start from, and thus the search needs to be more thorough. We use a population of $150$ and a standard deviation of $0.15$ in this case.

A set of $10$ simulations was run for each of six levels of parallelisation: $1, 2, 4, 8, 16$, and $32$ cores. All simulations were run until the map contained $400$ points, on a $64$-core Dell PowerEdge C6145.[1]

**Discussion** Figure 5.2 shows the average performance gain afforded by Parallel TRM over 'serial' TRM . The graph on the left plots the speedup for each number of cores for reaching the first 200 and 400 points. The black diagonal line represents the ideal, where speedup equals the number of cores. The graph on the right shows the performance in terms of the amount of real time required to reach 200 and 400 points, for a given level of parallelisation. Parallelisation at 32 cores reduces the time to generate a 400-point graph from 2267 seconds ($\approx$ 37 minutes) to 186 seconds. While this is a far cry from what would be required to allow for planning in even very slowly changing dynamic environments, the speedup of $12\times$ is significant. Moreover, the difference between the speedup for 200 and 400 was not found to be statistically significant ($\rho = 0.05$), which indicates that the rather modest parallel efficiency is not due to the increasing size of the map. Instead, the hardware architecture seems to suffer from a memory hierarchy that is not well suited to high-throughput parallel access [*First results for swift on a 64-core amd opteron 6376.*] Future experiments will migrate the system to new hardware with substantially larger L3 caches.

[1] with 4 AMD Opteron 6376 16-core CPUs running at 2.3 GHz, and 128 GB of RAM ($16 \times 8$ GB modules).

## 5.3 Concluding Remarks

The regular TRM building algorithm suffers from long building times. This is mitigated by using coordinated instances of NGIK that run in parallel. The results are significantly shorter building times which would allow TRMs to be built quickly in response to changes in the environment or task at hand. Big processors with many cores are becoming more easily available, and nowadays even entry computer processors quickly have 8 cores available, which can be used fully by this approach. With ever increasing computer power, we believe TRMs can be built fast enough in the future to allow for online and responsive control.

# Chapter 6

# Natural Gradient Control

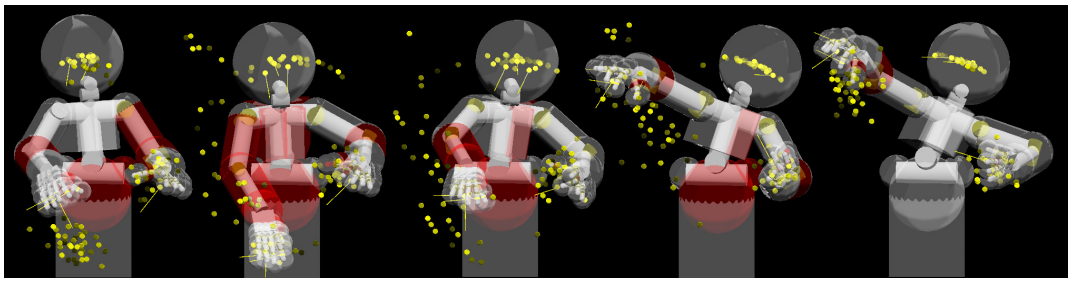


**Figure 6.1. Natural Gradient Control** Screenshots show from left to right how NGC transitions from a settled state to a change in fitness function and then settles down again. The dots show the task vectors corresponding to samples taken by the NGC algorithm. NGC can quickly adapt its covariance matrix to sample in the direction of best fitness, as seen by the changing sampling cloud.

TRMs are convenient but take time to build (see Chapter 4). This can be problematic when the environment, or even the task itself, changes, and a TRM needs to be rebuilt. Even though the building process can be sped up by parallelising the algorithm (see Chapter 5), there is a need for an optimisation algorithm that works online and directly adapts to changes in the environment.

The solution is an online control algorithm, called Natural Gradient Control (NGC). NGC is based on the insight that every iteration of the update step of NGIK can be used to form a control command. Thus, instead of running NGIK for many steps until a cost function is optimised, it can be run *one* iteration and receive a control command. This approach can run fast enough to function as a controller.

In order to prevent full convergence and keep sensitivity to changing goals, several adaptations to NGIK were made that control the covariance matrix of the distribution. NGC avoids the map-building process altogether and can thus react quickly to

environmental changes. Controlling a robot using NGC is as simple as changing the (parameters of) cost functions.

In this chapter we will first explain how NGIK can be adapted for online control to create NGC, and then compare NGC to similar algorithms to show its advantages. This work was published in 2015 [**M. F. Stollenga** et al., 2015], and parts of that publication appear in this chapter.

## 6.1 Adapting NGIK for Control

NGIK searches for solutions to the IK problem. The search is started from either a random position or a standard pose, and results in a solution in a (local) minimum by repeatedly estimating the natural gradient toward parameters of a search distribution and updating them accordingly (see Chapter 3). It thus tries to answer the question: 'What is the best kinematic configuration to minimise these cost functions?' However, for control we don't look for a global minimum; we want to find the best *direction of movement* from the *current configuration* that minimises the cost function.

So the following changes are made:

- The centre of NGIK's search distribution $\mu$ is always set to the current configuration parameters of the robot, obtained through its proprioseption sensors.
- Only a single NGIK iteration is made, estimating the natural gradients of the distribution parameters $\tilde{\nabla}_{\mu,\Sigma}(f(q \sim \mathcal{N}(\mu, \Sigma)))$.
- Instead of updating $\mu$ iteratively as in NGIK, the natural gradient towards $\mu$ functions as the control signal: $u = -\tilde{\nabla}_{\mu}$.
- The covariance matrix $\Sigma$ is updated as in NGIK, but additional normalisation steps are added to keep the search distribution responsive to changes, as explained below.

Essentially, NGC interprets the natural gradient computed by NGIK as a control signal. The mean $\mu$ is thus not directly updated as in NGIK, but indirectly updated *through the dynamics of the robot*.

The gradient for the mean is used to update the attractor $\hat{q}$ of the controller:

$$\hat{q} = q_t + \lambda \tilde{\nabla}_{\mu} J_c(\theta) \tag{6.1}$$

where $\lambda$ is determined by a line search, explained below.

NGIK is designed to reduce the magnitude of the covariance matrix as it reaches the (local) minimum. However, NGC is used as a low-level or mid-level controller,

for which the task goals are continuous and time-varying. Thus, we must make sure the covariance matrix does not shrink so it stays responsive to changes and does not suffer from 'tunnel vision'.

Therefore we regularise the covariance matrix $\Sigma$ by adjusting the matrix $B$ used by xNES (see Section 3.2) with the following procedure:

1. Update $B$ with the natural gradient term from NGIK.

2. Increase the diagonal values of $B$ from Algorithm 1 by adding a regularisation coefficient $\alpha$, to prevent $B$ from becoming too thin:

$$B \leftarrow B + \alpha\,\mathbb{I}. \tag{6.2}$$

3. Regularise $B$ to have a unit Frobenius norm:

$$B \leftarrow B/\|B\|_2. \tag{6.3}$$

For small values of $\alpha$, $B$ still adapts to the shape of the cost function of the local search space. However, $B$ is prevented from over-representing certain directions, and thus stays responsive for changes. When the robot pose gets close to the optimum for the current objective, Eqn. (6.2) slowly shapes $B$ back into an identity matrix, which corresponds to the initial state for $B$. This property ensures that NGC is ready to respond to new objectives. In addition, the covariance magnitude $\sigma$ is kept at a constant value. This simple adjustment reduces the influence of local minima and also allows the algorithm to quickly respond to changes in the task objective.

NGC is shown in Algorithm 7.

**Line Search** The algorithm relies on a line search in the direction of the natural gradient to find an appropriate parameter vector $q^*$ that functions as an attractor. Let $v = \tilde{\nabla}_\mu J_c(\theta)$ be the natural gradient at time $t$. Fix a maximum step size in parameter vector space $\gamma$, and let $\eta$ be a fixed decay factor ($\eta = 0.833$, determined experimentally). Then for $1 \leq i \leq I$:

$$q(i) = q - \gamma\,\eta^i\,\frac{J_t}{||J_t||} \tag{6.4}$$

find the least $i^*$ such that the cost $c(q(i^*))$ is less than $c(q(i^* + 1))$. Here $I$ is a maximum number of search steps, set to $I = 20$ in our experiments, at which point the discount is small $\eta^I \approx 0.026$. Then the line search outputs the attractor point $q(k^*)$ and the attractor is set (see Eqn. (6.1)).

**Algorithm 7** The NGC algorithm, derived from the original xNES algorithm in [Glasmachers et al., 2010b]. By using an infinite loop and regularisation on $B$ the algorithm continually optimises and is responsive to changes in task objectives. The $current\_pose()$ and $set\_attractor()$ functions provide the link to the robot.

**function** NGC(dimension $dim$, population size $K$, cost function $c$, density $\sigma$, inflation $\alpha$, step $\kappa$, decay $\eta$)
  $B \leftarrow I^{dim \times dim}$
  $u \leftarrow utilities(dim)$
  **while** true **do**
    $\mu \leftarrow current\_pose()$ ▷ Get Current Robot Pose
    **for** $i \leftarrow 1$ to $K$ **do** ▷ Sample New Population
      $z_i \leftarrow \mathcal{N}(0, I)$
      $q_i \leftarrow \mu + \sigma B \cdot z_i$
    **end for**
    sort $\{(z_i, q_i)\}$ w.r.t. $c(q_i)$ ▷ Fitness Evaluations
    $\nabla_\mu J_c(\theta) \leftarrow \sum_{i=1}^{K} u_i \, z_i$
    $\nabla_S J_c(\theta) \leftarrow \sum_{i=1}^{K} u_i \, (z_i z_i^T - I)$
    $\nabla_\sigma J_c(\theta) \leftarrow tr(\nabla_S J_c(\theta))/dim$
    $\nabla_B J_c(\theta) \leftarrow \nabla_S J_c(\theta) - (\nabla_\sigma J_c(\theta)) \; I$
    $\tilde{\nabla}_\mu J_c(\theta) \leftarrow B \cdot \nabla_\mu J_c(\theta)$ ▷ Estimate Gradient
    $B \leftarrow B \cdot \exp(\nabla_S J_c(\theta))$
    $B \leftarrow B + \alpha \, I$ ▷ Inflate $B$
    $B \leftarrow B / \|B\|_2$ ▷ Regularise $B$
    $q^* \leftarrow line\_search(q, \tilde{\nabla}_\mu J_c(\theta), \kappa, \eta)$
    $set\_attractor(q^*)$
  **end while**
**end function**

## 6.2 Control Cost Function

Controlling the robot through NGC is done by changing the cost function. The cost function $c'(q)$ is built by combining several costs as described in Chapter 3, and then a *task function* $g(q)$ is defined as in Chapter 4. We then add a *control cost function* based on the task function that allows us to move through task-space:

$$c(q) = c'(q) + \lambda \|g(q) - \tau^*\|_2 \tag{6.5}$$

where $\lambda$ is a constant controlling the strength of the control cost, and $\tau^*$ is the desired position in task-space. The result is a controllable cost function that is controlled by changing the variable $\tau^*$. This function can directly be used in NGC.

## 6.3 Experiments

To investigate the behaviour of NGC two experiments were performed on a model of the iCub:

1. **Dynamics:** This experiment investigates the behaviour of NGC and compares it to Jacobian-based control determined using finite differences. The goal is to show that NGC can provide gradient signals that are more well-behaved than Jacobian methods, and the covariance of NGC's search distribution adapts to the cost functions.

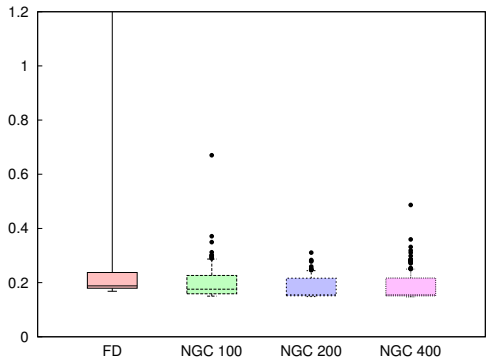
2. **Complex Vision-Based Cost:** Section 3.5.1 shows the experiments using NGC with a Pixel Counting cost function that generates an image of the robot, and then minimises or maximises the occurrence of certain pixel values. Such functions are highly non-linear and not easily differentiable with respect to the kinematic chain of the robot. This experiment shows how NGC is applicable to domains that are out of reach for Jacobian-based control.

**Dynamics** A cost function was constructed with the following goals: The iCub should place the palm of its right hand into a grasping position at a given location in real-world coordinates, with the fingers pointing downwards and with the eyes of the robot pointing towards the hand. This configuration is intended to reflect the task described in the introduction where the robot makes a movement in order to grasp a chess piece, while looking at it. The iCub is allowed to use its full $41$ degrees of freedom to solve the task. The cost function has the following components:

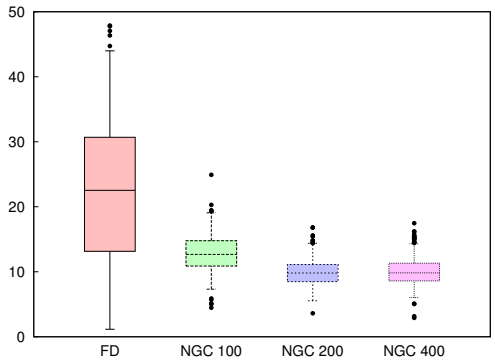
- **Right Hand Position:** The goal of the experiment is to move the right hand to a particular real-world location. The cost for this purpose is the $\ell^2$-norm between the current and desired position ($v_{righthand}$ and $v^*_{righthand}$ respectively) of the right palm of the hand: $c_{position} = \|v_{righthand} - v^*_{righthand}\|_2$. The right-hand objective may be controlled over time by altering $v^*_{righthand}$, thus changing the total cost function.

- **Left Hand Position:** The left-hand cost $c_{lefthand}$ is the same as for the right-hand, but the left-hand position remains fixed at an initial position in these experiments.

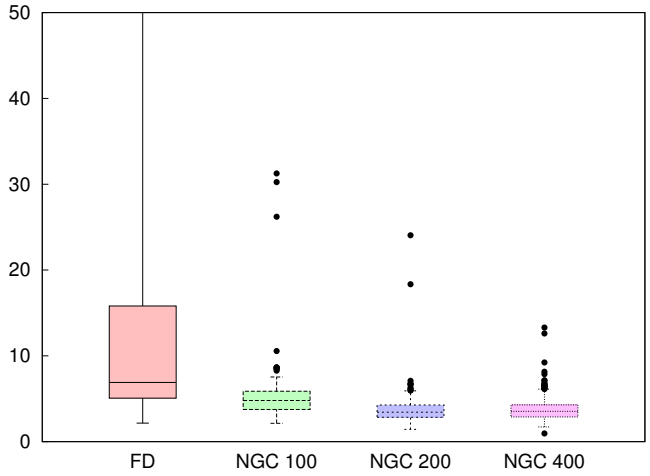
- **Looking:** Both the left and right eye should be aimed at the right index finger to enable perception. The cost function $c_{eye}$ uses the dot product of the normalised vector coming out of the eye $u_{eye}$ and the normalised vector pointing from the eye to the finger: $u_{eye\_diff} = \frac{w_{rindex} - v_{eye}}{\|v_{rightindex} - w_{eye}\|_2}$. Separate cost functions are used




**(a)** Final Cost




**(b)** Path Length




**(c)** Integrated Path Cost

**Figure 6.2. NGC Results** Experiments comparing NGC with different population sizes to FD on a grasping task from 200 random starting poses. The box plots include 95% of the data-points, the rest are drawn as outliers with dots (if in range). (a) NGC moves the robot into configurations with lower final costs with greater consistency than FD. In 40 of the 200 cases (20%) FD ends up in undesirable local minima which cannot be escaped and ends up in collisions, resulting in high final and integrated costs. (b) Path lengths show little difference between $K = 400$ and $K = 200$ for NGC. $K = 100$ performs noticeably worse and takes longer paths. FD results in longer paths in most cases, while a few times the path is remarkably short. Failure in the average case is due to an early local minimum, stopping the algorithm prematurely. (c) The cost integrated over the path shows the efficiency of the algorithms. Again $K = 400$ and $K = 200$ perform very similarly, barring a few outliers. $K = 100$ shows noticeably higher costs, and FD performs consistently worse. Again a few samples for FD have a very low cost, due to early stopping.

for the left and right eye due to the details of the robot platform. The eyes are coordinated by NGC automatically.

- **Point Fingers Down:** To prepare for a grasp, a cost function is developed that rewards the robot when the index finger, middle finger, and thumb are all pointed downward. Separate cost functions are calculated for each finger and for the thumb by taking the dot product between the vector coming out of the fingers and a vector pointing down the z-axis.

- **Home pose:** In order to discourage the robot from moving into strange positions, the $\ell^2$-norm between the current pose $q$ and a stable homing pose $q_0$ is controlled using $c_{home} = \|q_t - q_0\|_2$. This cost acts as a regularisation term.

- **Collision:** Collisions are detrimental to the iCub and thus are heavily discouraged by multiplying the number of collisions with a big factor $c_{col} = 1000 \cdot |collisions|$. The factor 1000 ensures a collision will always supercede other constraints.

NGC was compared to a Jacobian-based method where the Jacobian is estimated using finite differences (FD): The Jacobian of a cost $c$ at a pose $q$ is estimated by making small perturbations to $q$ in orthogonal directions and treating the normalised differences in $c$ as the gradient (perturbations of $10^{-3}$ degrees per joint were used). Calculating the Jacobian using FD is accurate if the function is smooth over the small perturbations, which is the case for our cost functions.[1]

To compare the gradients of NGC and Jacobian-based methods, we applied a simple simulation environment. When FD and NGC are run on the real robot, they yield a new attractor point $\hat{q}$ at each time step. In the model-based experiments, rather than simulate the dynamical system, a step of fixed length is taken towards the attractor at each time step. This scenario would correspond to the robot always running at the same speed through joint parameter space, without momentum or gravity. Although this scenario may seem unrealistic, it also avoids complicated interactions with the environment, which may bias the results. Our focus here is to compare the quality of the control signals, without the complications of a full dynamical system. We also run NGC on the real iCub in experiments below.

A set of $200$ starting positions was generated randomly, and FD and NGC were used to minimise the cost function in each case. NGC used several population sizes $K$: 100, 200, and 400. The optimiser was stopped when the function does not decrease for 10 iterations.

[1]Calculating the Jacobian using FD is a common trick, e.g. MATLAB's solvers default to FD estimation if no Jacobian is given for the provided objective function [*Nonlinear Equations with Finite-Difference Jacobian - MATLAB*].

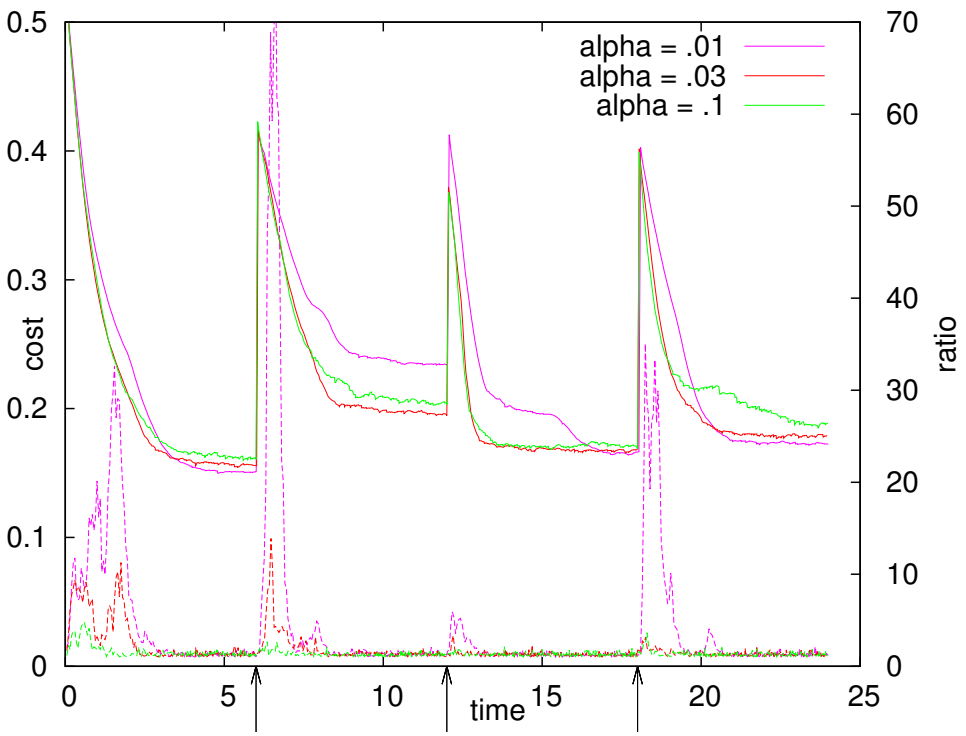



**Figure 6.3. Responsiveness of NGC** Plot shows the responsiveness of NGC to changes in the task objective, indicated by arrows. The left scale shows the change in cost over time. When the objective changes, the cost spikes and is then reduced over time. The right scale shows the ratio of the first to the second eigenvalue of the covariance matrix. This ratio increases as NGC responds to the new objective, then stabilises to one when a local minimum is reached. Values are graphed for different regularisation parameters $\alpha$. Lower $\alpha$ results in faster adaptation, but can lead to erratic behaviour when it adapts too much.

The results are shown in Figure 6.2 in box plots, which include 95% of the samples. The 5% outliers are drawn as dots if in range. Figure 6.2a shows the final cost achieved by the algorithms, which measures how well each method was able to solve the task. NGC slightly outperforms FD on average in terms of the final cost, with little difference between the different population sizes. However, a big spike in the box plot shows the FD algorithm often ends up in local minima which it cannot escape. Because FD does not sample around its searching position like NGC, it cannot see ahead and gets pushed into collisions, adding a high cost. In fact 40 of the 200 experiments (20%) resulted in a collision. It is hard for FD to get out of such a position by simple gradient descent in the cost function.

Figure 6.2b shows the length of the path taken, and Figure 6.2c the cost integrated

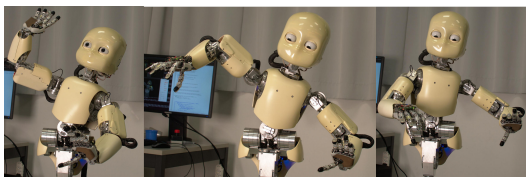
over the path (see Figure description for details). It shows that NGC outperforms FD in most cases, with much more stable performance. Increasing the population of NGC from $100$ to $200$ results in a significant increase in performance, while increasing to $400$ only increases performance slightly. The experiments in Figure 6.2 and the demonstration in Figure 6.4 show that NGC can be used as a flexible and robust low- or mid-level controller. The primary benefit of NGC is its versatility with respect to complex task constraints. This benefit comes with the computational burden of sampling hundreds of robot poses and computing their cost. Nonetheless, the quality of the control signal obtained justifies the computational overhead in many tasks, and the flexibility of NGC can enable coarse-grained planning algorithms.

**Effects of Regularisation** To understand the effect of regularisation, the same cost function was optimised, while periodically changing the goal for the right hand to a position far removed from the previous position. Figure 6.3 shows the responses of NGC to these changes. Arrows point out when the goal is changed. The graphs in the upper part show how the cost decreases over time. When the goal changes, the cost immediately increases and is then gradually reduced by NGC. The bottom graphs show the ratio between the first and second eigenvalues of the covariance matrix. When the goal changes, this ratio increases quickly, adjusting its distribution to the primary direction of steepest descent. As the goal is reached, the ratio settles as the covariance matrix stabilises to the identity matrix. Different regularisation values $\alpha = .01, .03, .1$ were tested; a higher value means a stronger pull of the covariance matrix towards an identity matrix. The best responses are given by setting $\alpha = .03$. A lower $\alpha$ leads to erratic behaviour due to big changes in the covariance matrix, and higher $\alpha$ reduces responsiveness.

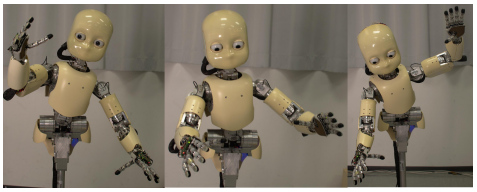
Figure 6.1 shows screenshots of the optimiser for $\alpha = .3$ and $K = 400$ when a goal position changes. At the left panel, the search distribution starts in a settled-down state, then notices the change and quickly adapts by sampling in the direction of the new goal. Finally, it settles down again upon reaching the goal in the right panel. The experiments demonstrate that NGC is able to respond flexibly to a complex cost function where the Jacobian provides an inadequate control signal.

**Demonstration on the iCub** In order to verify that the experimental setting properly reflects real-world behaviour, NGC was used to control the real robot. Figure 6.4a shows the starting and ending positions for NGC in this task. From a position far away from the goal, NGC controls the robot to a pose that looks at the hand and holds the fingers downwards.

To show the flexibility of NGC the left and right hand are controlled separately. In



**(a)** NGC can control the iCub to perform a reach behaviour. The corresponding cost function encourages the fingers of the right hand to point down, and the eyes to look at the right index finger, to possibly allow for perception-based control.



**(b) NGC Demonstration** NGC can control both arms independently by dynamically changing the goal position. This cost function states that the iCub should look at its right finger while controlling its 41-degrees of freedom and avoiding self-collisions.

**Figure 6.4. Demonstrations** Demonstrations on the real iCub

this case, the requirement to point the fingers down is removed to allow for elaborate movements, but the gaze of the robot is still aimed at the hand. Figure 6.4b shows that the robot can freely control its 41 degrees of freedom to enable these movements. These results show the feasibility of complex cost functions that are easily incorporated in NGC but very hard to combine with Jacobian-based inverse kinematics approaches.

## 6.4 Discussion

The experiments demonstrate that NGC is a viable low- to mid-level control method that can incorporate complex, non-differentiable task constraints for high degree-of-freedom robotics. In this section, the advantages and limitations of NGC are discussed along with possible ways of incorporating these advantages into Jacobian methods as well. It has been shown to provide more stable and robust control than the standard task Jacobian in the case of a reaching grasp. The reasons for this stability and robustness arise from the use of a natural gradient on a space of parameterised probability distributions that add hidden degrees of freedom to the robot that can be used to avoid ill-conditioned and oscillatory control signals. NGC has also been shown to easily incorporate very complex cost functions, such as statistics of rendered images, which is the main advantage over traditional Jacobian-based methods.

**Advantages of NGC** NGC finds good gradients in some places where the Jacobian does not. There are several reasons for this fact. Firstly, the sampling-based natural gradient includes curvature information, allowing it to avoid plateaus and other problematic properties of the cost function. Secondly, NGC searches for poses indirectly in a space of Gaussian probabilities. The expanded parameter space for

these distributions makes it possible to avoid ill-conditioned gradients by fixing the width of the covariance matrix. If the covariance matrix were allowed to become small, then the natural gradients could explode. However, since the covariance matrix is separate from the pose being optimised, it is possible to control it separately from the pose. A similar technique might also be possible for Jacobian-based controllers by adding extra dimensions. Thirdly, local sampling in NGC allows NGC to consider movement in directions that would not be proposed by the Jacobian. These movements appear as correlations among the various joints. NGC can thus find ways of improving the cost that are hidden to the Jacobian.

Regarding computational cost, the MoBeE framework is capable of testing $3000$ poses per second for collisions and cost. With a population size of $200$, NGC could theoretically be run at a frequency of $15$ updates per second. In actual practice, reading the state of the iCub robot introduces a latency that reduces this frequency. A higher frequency of $60$ updates per second would be desirable. It may be possible to further optimise the system to achieve this goal. The computational cost and the difficulty of implementing the NGC system cannot be ignored. The vastly improved versatility seems to merit these costs, however. In its current state, NGC has sufficient frequency to provide good control of the robot in demonstrations.

**Limitations** NGC requires significantly more computation than most inverse kinematics approaches since many samples have to be taken and calculated per time-step to estimate the gradient. With current increases in computation power and assuming more optimised code, we believe this constraint can be alleviated. Also, NGC calculates a gradient directly in joint-space and thus does not take dynamics into account. Even if the gradient is pointing in the perfect direction to minimise the cost, it cannot be assumed that all joints respond with the same speed (or at all). Such dynamics are inherently tied to the real world and thus are hard to model, although they can be learned [Atkeson et al., 1997; Ijspeert et al., 2002].

## 6.5 Concluding Remarks

Throughout these chapters, we developed several algorithms that use high computational power to their advantage to control a humanoid. It allowed us to freely create humanoid movements by using the sampling-based NGIK to build TRMs. The use of computational power was maximised by parallelising this algorithm over many cores.

In this chapter we took a step back from the TRM building process and adjusted NGIK to perform online control, resulting in NGC. NGC is applicable for online tasks and continuously outputs a control signal while staying responsive to changes in the

cost function. In total, this gives us a wide set of algorithms that mitigate limitations that other algorithms have in this field, and can be used in many different settings.

Future work could focus on bringing the NGIK methods into the realm of 'dynamic control' in which forces to motors are directly optimised. Such approaches are more involved since they need a detailed model of the physics of the robot, and need even more computational power. However such physical models could be learned by the algorithm from experience [Atkeson et al., 1997; Ijspeert et al., 2002] and the computational power is expected to catch up in the future.

This concludes our work on humanoid control. In the next part we will investigate algorithms for computer vision.

# Part II

# Vision

# Chapter 7

# Artificial Neural Networks

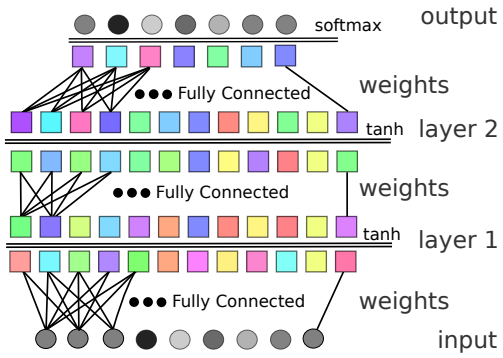



**Figure 7.1. Artificial Neural Network** A multi-layer neural network is shown. The input is processed by multiplying each value with weights, denoted by lines. The results are stored in the next layer, after which a non-linear function like $\tanh$ or $softmax$ is applied. This process is repeated for every layer until the output is reached. We formalise neural networks in Section 7.3.

## 7.1 Vision

We developed several control algorithms, but a humanoid also needs to perceive the world. In the following chapters, we develop several visual processing algorithms. Big leaps have been made in this field with the introduction of the fast convolutional neural network (Section 7.5), that push the computational capabilities of today's computers to their limits.

However, most algorithms in this field approach vision as a static problem; applying a fixed amount of computation for every sample, and then forgetting about it. In

contrast, a humanoid robot has to deal with dynamic environments where time plays an important role. Depending on the situation and the difficulty of the visual task, it should use more or less time to process input.

The algorithms we will develop in the following chapters bring such dynamism to existing visual processing architectures. In essence, this allows these algorithms to use their computational power more efficiently. Even though this adds complexity and pushes the hardware limits even further, we find efficient learning algorithms that make this manageable.

In this chapter, we introduce the basis of our work; the neural network and its derivatives. Then, Chapter 8 develops our dynamic neural network that learns to focus on different features of the image using a gating mechanism. Following that, Chapter 9 develops a 3D volumetric segmentation algorithm that uses the complex gating mechanisms of an LSTM (explained in Section 7.6) to its advantage. Chapter 10 shows an application combining our work in humanoid control with the visual processing algorithms developed here. Finally, Chapter 11 concludes the thesis.

## 7.2 Neural Networks

Artificial Neural Networks (NNs) are biologically inspired computational architectures that try to mimic the processing power of brains in a computer. Such architectures were first invented around the time the first primitive computers arose [McCulloch and Pitts, 1943], but they were not able to learn. The first network that could learn was the perceptron, which performed a simple linear operation on its input, followed by a non-linearity [Rosenblatt, 1958; Rosenblatt, 1961]. A variant of linear regression [Legendre, 1805] was used to train the networks.

Quickly, researchers figured out that they could improve the computational capacity by increasing the number of 'layers', where each layer performs a linear operation followed by a nonlinear operation. The more layers a network has, the 'deeper' it is. The first of such *deep networks* were trained one layer at a time [Ivakhnenko and Lapa, 1965; Ivakhnenko and Lapa, 1967; Ivakhnenko, 1968].

However, soon a better approach emerged to train such deep NNs. The winning approach was in hindsight simple and came from the traditional differentiable calculus [Leibniz, 1684; Gauss, 1809]. By defining an error function and using differentiation, a gradient can be calculated. Then, *gradient descent* gives the direction in which the NN parameters should be changed to decrease the error [Hadamard, 1908]. Linnainmaa, 1970 was the first to apply this method to NNs, but his method did not scale well for deep NNs. Thich was solved by the invention of the efficient backpropagation algorithm [Kelley, 1960; Bryson, Jr. and Denham, 1961; Linnainmaa, 1970;

Werbos, 1974; Dreyfus, 1973], a dynamic programming approach [Bellman, 1957] to calculating the gradient by propagating the error backwards through the network while reusing computations. This approach was later applied to NNs [Linnainmaa, 1970; Werbos, 1981; Parker, 1985; LeCun, 1985; LeCun, 1988; Rumelhart et al., 1986] and is discussed in Section 7.3.1. This simple approach is what achieves state-of-the-art results until this day [Ciresan et al., 2012b; Krizhevsky et al., 2012; Goodfellow et al., 2013; Lin et al., 2013].

**Deep Learning** In the process of building more efficient NNs for image processing, *convolutional neural networks* (CNN) were invented [Fukushima, 1979; Fukushima, 1980; Fukushima, 2013], by using the locality properties of images to reduce the number of connections (see Section 7.5). Also, CNNs are trained with backpropagation [LeCun et al., 1989; LeCun et al., 1990],

However, even with these changes, the training of such architectures was impractical since computers were simply not fast enough. This changed with the introduction of programmable Graphical Processing Units (GPUs). GPUs are normally used to draw graphical scenes in computer games and are very good at doing local computations in parallel. Ciresan et al., 2011 and Krizhevsky et al., 2012 realised GPUs were perfectly suited to do convolution operations and used them to train convolutional neural networks, achieving remarkable results, in some case performing better than humans on image recognition tasks [Ciresan et al., 2011; Ciresan et al., 2012b; Krizhevsky et al., 2012; Goodfellow et al., 2013; Lin et al., 2013; Ciresan et al., 2013; Ciresan et al., 2012a; Farabet et al., 2013; Sermanet et al., 2013].

Since then, the field of machine learning and computer vision has been dominated by GPU based CNNs. With the ever increasing power of GPUs, the computational architectures are able to use more and more layers of computations, increasing the complexity of their computations and increasing their representational power. Architectures with many layers have been popularised under the name 'deep' architectures, referring to the 'depth' of the information flow through the layers. 'Deep' also seems to refer to 'profoundness' of their representation, often corresponding to higher level concepts [Zeiler and Fergus, 2014] such as the famous 'cat'-neuron which only responds to images of cats after being trained on many YouTube videos [Markoff, 2012]. Such representational power has been likened to the concept of 'Grandmother' neurons [Gross, 2002], a theory that there is a neuron that specifically responds to your grandmother. These have been identified in human brains with the discovery of the 'Jennifer Aniston' neuron, a neuron that actually explicitly turns on when the person sees or hears about the actress Jennifer Aniston [Quiroga et al., 2005]! After training, CNNs show remarkable similarities with cells in the brain [Hubel and Wiesel, 1959;

Hubel and Wiesel, 1962] resembling Gabor filters [Gabor, 1946], adding validity to their name. CNNs are introduced in Section 7.5. See the review paper by Schmidhuber, 2015 for a comprehensive overview of CNNs and deep learning.

**Recurrent Neural Networks** To process arbitrary sequences, recurrent neural networks (RNN) were invented. An RNN introduces recurrent connections that turn a regular NN into a dynamical system that can sequentially process inputs, one step at a time. Again, backpropagation is applied to train these networks [Williams, 1989; Robinson and Fallside, 1987; Werbos, 1988; Schmidhuber, 1992]. However, due the equations governing the RNN, which extensively use multiplications, the error cannot be propagated far through the RNN before diffusing, vanishing or exploding [Hochreiter, 1991]. In essence, the multiplications cause an exponential reduction of the error during backpropagation.

The Long Short-Term Memory network (LSTM; [Hochreiter and Schmidhuber, 1995]) was designed to mitigate this issue by changing the equations. A 'cell' variable was added that retains information and is governed by several gates, such that errors can flow unhindered over long time periods. The LSTMs now have state-of-the-art performance on many tasks [Fan et al., 2014; Sak et al., 2014a; Graves and Jaitly, 2014]. RNNs and LSTM equations are discussed in Section 7.6.

The following sections formalise the NN, CNN, RNN and LSTM and their training algorithms.

## 7.3 NN Formalisation

Here we formalise multi-layer neural networks, the basis of the computational architectures that will be presented in this thesis. We denote the input of the network as $x \in \mathbb{R}^n$ and the output of the network as $y \in \mathbb{R}^m$.

A neural network consists of $N$ layers; each taking input from below, performing computations on it, and sending the result to the next layer. Each layer represents a vector of activations $h^i \in \mathbb{R}^{H_i}$, where $H_i = dim(h^i)$ is the size of the layer. The strength of neural networks lies in the consecutive layer to layer computations, which result in very complex information processing on the input. Each layer has a set of parameters which define which computations are done, defined by a weight matrix $\mathbf{W}^i$ and a bias vector $b^i$, where $i$ denotes the layer to which the matrix or bias belong to. Changing these parameters changes the computations, and as such learning can occur. The *input layer* has the index $i = 0$, the *output layer* has index $i = N - 1$. If $N > 2$ then there are one or more layers in-between the output and input layer, called *hidden layers*.

The activations $h^i$ of every layer depends on the activations $h^{i-1}$ of the layer below it, and is calculated as follows:

$$h^{i+1}(b^i, \mathbf{W}^i, h^i) = f(b^i + \mathbf{W}^i h^i) \tag{7.1}$$

where $h^i$ and $h^{i+1}$ are the activations of hidden layers $i$ and $i+1$, $\mathbf{W}^i \in \mathbb{R}^{H_{i+1} \times H_i}$ is a weight matrix, and $b^i \in \mathbb{H}$ a bias vector.

**Non-Linear Operations** The function $f$ is a non-linear function applied element-wise; typically used functions are:

the *Hyperbolic Tangent* function: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$,

the *Sigmoid* function: $\sigma(x) = \frac{1}{1+e^{-x}}$,

and the *Rectified Linear* function: $ReLu(x_i) = \max(x_i, 0)$. The Rectified Linear (ReLu) function [Nair and Hinton, 2010; Dahl et al., 2013; Jarrett et al., 2009]. It simply assures a value is non-negative, by returning $0$ for negative numbers.

If the task of the NN is to classify, the *Softmax* function is used in the final layer:

$$softmax(x)_i = \exp(x_i) / \sum_i \exp(x_i) \tag{7.2}$$

The result of *softmax* is a vector of activations that are positive and sum up to 1. It can thus be directly interpreted as a probability for different classes. Without non-linear operations, there is no use for multiple layers since the linear operations can be collapsed in one layer.

The full process of calculating the network's output activation $y$ from the input activation $x$ works as follows:

1. Define the input activation for the lowest layer $h^0 = x$.
2. Calculate the activations $h^{1..N-1}$ of the subsequent layers according to formula (7.1).
3. Define the output as the last activation vector $y = h^{N-1}$.

To concisely write this computation, we define:

$$y = f_\theta(x) \tag{7.3}$$

where $\theta = \{(\mathbf{W}^i, b^i) \mid i = 0 \cdots N-1\}$ are the network parameters.

### 7.3.1 Backpropagation

Now that we know how to evaluate neural networks, we want to train them. One of the most used methods is gradient descent [Hadamard, 1908]. A well-known method to calculate the gradient through the network efficiently is backpropagation [Werbos, 1981].

Say we have an example from the dataset $(x, \hat{y})$ where $x$ is the input and $\hat{y}$ the corresponding desired output. The output of the neural network is denoted as $y = f_\theta(x)$. We define an error metric between the desired output and actual output that we want to minimise:[1]

$$\min_\theta E(x, \hat{y}, \theta) \tag{7.4}$$

$$E(x, \hat{y}, \theta) = \frac{1}{2}\|f_\theta(x) - \hat{y}\|^2 \tag{7.5}$$

where $f_\theta$ is the neural network to be trained, parameterised by $\theta$.

The next step is to calculate the gradient of this error $\frac{\partial E}{\partial \theta}$ with respect to the parameters of the network. Since the neural network is a composition of simple mathematical operations, we can use the chain rule to calculate the gradient towards the parameters of a layer, e.g. the output layer. In the following derivation, we assume $\tanh$ function is used $f = \tanh$ and use the following notation to calculate the activation of *one* neuron $j$ in layer $i + 1$ :

$$h_j^{i+1}(b_j^i, \mathbf{W}_j^i, h^i) = \tanh\left(b_j^i + \mathbf{W}_j^i h^i\right) \tag{7.6}$$

Then the gradient of the error towards the weights of the output layer is calculated as:

$$\frac{\partial E}{\partial \mathbf{W}_{ij}^{N-1}} = \tag{7.7}$$

$$\frac{\partial \frac{1}{2}\|f_\theta(x) - \hat{y}\|^2}{\partial \mathbf{W}_{ij}^{N-1}} = \tag{7.8}$$

$$(y_i - \hat{y}_i)\frac{\partial f_\theta(x)}{\partial \mathbf{W}_{ij}^{N-1}} = \tag{7.9}$$

$$(y_i - \hat{y}_i)\frac{\partial h_i^N}{\partial \mathbf{W}_{ij}^{N-1}} = \tag{7.10}$$

$$(y_i - \hat{y}_i)[1 - \tanh^2(b_i^{N-1} + \mathbf{W}_i^{N-1} h^{N-1})]h_j^{N-1} \tag{7.11}$$

[1] This derivation of backpropagation is taken from my Master's thesis [**M. F. Stollenga**, 2011]

where $\mathbf{W}_{ij}$ is the element from matrix $\mathbf{W}$ on row $i$ and column $j$, $\mathbf{W}_i$ is the row $i$ of matrix $\mathbf{W}$.

These formulas are manageable because the weight $\mathbf{W}_{ij}^{N-1}$ only contributes to one element of the output $y_j$, avoiding complicated dependencies. However, if we want to derive the gradient for a weight in the next layer down $\mathbf{W}_{ij}^{N-2}$ the gradient becomes rather unwieldy because this weight indirectly contributes to all elements in $y$:

$$\frac{\partial E}{\partial \mathbf{W}_{ij}^{N-2}} = \tag{7.12}$$

$$(y_1 - \hat{y}_1)\frac{\partial h_1^N}{\partial h_i^{N-1}}\frac{\partial h_i^{N-1}}{\partial \mathbf{W}_{ij}^{N-2}} + (y_2 - \hat{y}_2)\frac{\partial h_2^N}{\partial h_i^{N-1}}\frac{\partial h_i^{N-1}}{\partial \mathbf{W}_{ij}^{N-2}} + \cdots + (y_O - \hat{y}_O)\frac{\partial h_O^N}{\partial h_i^{N-1}}\frac{\partial h_i^{N-1}}{\partial \mathbf{W}_{ij}^{N-2}} = \tag{7.13}$$

$$\left[(y_1 - \hat{y}_1)\frac{\partial h_1^N}{\partial h_i^{N-1}} + (y_2 - \hat{y}_2)\frac{\partial h_2^N}{\partial h_i^{N-1}} + \cdots + (y_O - \hat{y}_O)\frac{\partial h_O^N}{\partial h_i^{N-1}}\right]\frac{\partial h_i^{N-1}}{\partial \mathbf{W}_{ij}^{N-2}} \tag{7.14}$$

It is even worse for the layer after that. It seems our calculations grow exponentially, the deeper our weights are. But we have a solution; we already calculated $\frac{\partial h_i^N}{\partial \mathbf{W}_{ij}^{N-1}}$ in equation 7.11, which is very similar to $\frac{\partial h_i^N}{\partial h_j^{N-1}}$ on the left side of equation 7.14:

$$\frac{\partial h_i^n}{\partial \mathbf{W}_{ij}^{n-1}} = \left[1 - \tanh^2(b_i^{n-1} + \mathbf{W}_i^{n-1}h^{n-1})\right] h_j^{n-1} \tag{7.15}$$

$$\frac{\partial h_i^n}{\partial h_j^{n-1}} = \left[1 - \tanh^2(b_i^{n-1} + \mathbf{W}_i^{n-1}h^{n-1})\right] \mathbf{W}_{ij}^{n-1} \tag{7.16}$$

The solution of backpropagation is to use extra variables, called deltas $\delta_i^n$, which store and reuse intermediate computed values. These deltas are defined in the following

way:

$$\delta_i^N = (y_i - \hat{y}_i) \tag{7.17}$$

$$\delta_i^{n-1} = \left[\delta_1^n \frac{\partial h_1^n}{\partial h_i^{n-1}} + \delta_2^n \frac{\partial h_2^n}{\partial h_i^{n-1}} + \cdots + \delta_M^n \frac{\partial h_M^n}{\partial h_i^{n-1}}\right] \tag{7.18}$$

$$= \sum_{j=1..M} \delta_j^n \frac{\partial h_j^n}{\partial h_i^{n-1}} \tag{7.19}$$

Substituting 7.18 in 7.11 and 7.14 gives our equation to calculate the gradient of the error to the weights using deltas:

$$\frac{\partial E}{\partial \mathbf{W}_{ij}^{n-1}} = \delta_i^n \frac{\partial h_i^n}{\partial \mathbf{W}_{ij}^{n-1}} \tag{7.20}$$

We write the gradient of the error towards the parameters as:

$$\nabla_\theta E_\theta(x) = \{(\frac{\partial E}{\partial \mathbf{W}^i}, \frac{\partial E}{\partial b^i}) \mid i = 0 \cdots N-1\}. \tag{7.21}$$

## 7.4 Training

One of the simplest approaches to learning is to define a parameterised cost function $E_\theta(x, \hat{y})$ and calculate its derivative towards the parameters $\nabla_\theta E_\theta(x, \hat{y})$. The result is a vector in the parameter space of $\theta$, pointing to the direction of steepest ascent of the cost. Thus, going in the opposite direction should decrease the cost.

However, the gradient $\nabla_\theta E_\theta(x, \hat{y})$ is a local measure and only gives the slope for the particular input $x$ and parameters $\theta$. As soon as another input is given, or the parameters change, the gradient can change. A simple and effective approach to deal with this is to create a dataset of samples $(x, \hat{y}) \in D$, randomly sample from it and perform a small update to the parameters in the opposite direction of the gradient: $\theta = \theta - \lambda \nabla_\theta E_\theta(x, \hat{y})$. The learning algorithms used in this thesis are all related to this approach, which are explained below.

### 7.4.1 Stochastic Gradient Descent (SGD)

Backpropagation can efficiently calculate the gradient of the error towards the network parameters. Moving the parameters in this direction is the quickest way to increase the error; we want to minimise the error so we move in the opposite direction of the

gradient, creating our update rules:

$$\theta \overset{\lambda_{learning}}{\longleftarrow} \theta - \nabla_\theta E(x, \hat{y}) \tag{7.22}$$

The stochastic gradient descent algorithm is as follows:

1 Take a random input sample with corresponding output $(x, \hat{y})$ out of the dataset $D$.

2 Update parameters $\theta$ according to Eqn. (7.22).

3 Repeat 1 and 2 until learning rate $\lambda_{learning}$ is lower than a threshold (other stopping criteria are possible).

**Learning rates** The learning rates $\lambda_{weight}$ and $\lambda_{bias}$ are typically set such that they slowly decrease over time. In this work, an exponential decay is used to set the learning rate such that in the $n$'th iteration, the learning rate is:

$$\lambda_{learning}(n) = \lambda_{min} + (\lambda_{max} - \lambda_{min}) \cdot \gamma^n \tag{7.23}$$

where $n$ is the epoch number, $\lambda_{max}$ and $\lambda_{min}$ are the maximum and minimum learning rates and $0 < \gamma < 1$ is a decay rate. The learning rate exponentially decays from $\lambda_{max}$ with $\lambda_{min}$ as its asymptote. The decay rate $\gamma$ is typically defined in a half-time; the number of epochs which it takes to half the distance of $\lambda$ to $\lambda_{min}$. It is defined by $\gamma = \frac{1}{2}^{\frac{1}{halftime}}$, where $halftime$ defines the number of steps until halving.

**Momentum** Gradient descent can end up in local minima. A simple method to mitigate this is to add momentum to the learning updates [Rumelhart et al., 1986]. Just as in physics, the momentum picks up speed in the direction that the gradient points to on average. This avoids local minima by allowing the momentum to push the state of the system through a local dip in the search space.

Instead of directly using the gradient to update the weights, as in Eqn. (7.22), an extra variable is introduced:

$$\mathbf{m} \overset{\lambda_{momentum}}{\longleftarrow} \nabla_\theta E \tag{7.24}$$

$$\theta \overset{\lambda_{weight}}{\longleftarrow} \theta - \mathbf{m} \tag{7.25}$$

The added momentum term $\mathbf{m}$ builds a moving average of the gradient and thus holds the momentum. This approach is used in Chapter 8.

### 7.4.2 RMSPROP

A recent adaptation of SGD called Root Mean Squared Error Propagation (RMSPROP; Tieleman and Hinton, 2012), has become popular. RMSPROP is useful for complex neural networks where certain weights get small gradients due to their position in the network, but are important for learning and need to be updated. RMSPROP estimates the variance of the gradient for every parameter, using a moving average. Before updating a parameter, the gradient is divided by the root of this moving average. The result is that the variance of the update step is (approximately) the same for every weight. Even weights with very small gradients get significant updates.

Formally the following steps are performed every iteration:

$$\text{MSE} \xleftarrow{\lambda_{MSE}} \nabla_\theta^2 E \tag{7.26}$$

$$G = \frac{\nabla_\theta E}{\sqrt{\text{MSE}} + \epsilon} \tag{7.27}$$

$$\theta = \theta - \lambda_{lr} G \tag{7.28}$$

where $E$ is the error function, MSE a running average of the variance of the gradient, $\lambda_{MSE}$ a constant, $\nabla^2$ the element-wise squared gradient, $G$ the normalised gradient, $\theta$ the weights, and $\epsilon$ a small value to prevent division by 0.

**Momentum** Additionally, a momentum can be added before the update step to reduce noise in the gradient:

$$\text{MSE} \xleftarrow{\lambda_{MSE}} \nabla_\theta^2 E \tag{7.29}$$

$$G = \frac{\nabla_\theta E}{\sqrt{MSE} + \epsilon} \tag{7.30}$$

$$M \xleftarrow{\lambda_M} G \tag{7.31}$$

$$\theta = \theta - \lambda_{lr} M \tag{7.32}$$

where $M$ is the smoothed gradient holding momentum, and $\lambda_M$ and $\lambda_{MSE}$ are constants. This approach is used in Chapter 9.

## 7.5 Convolutional Neural Networks

The strong representational powers of a neural network come from its consecutive application of computations on the input throughout its layers. However, the computational burden grows quickly as the size of the input or the number of the hidden units

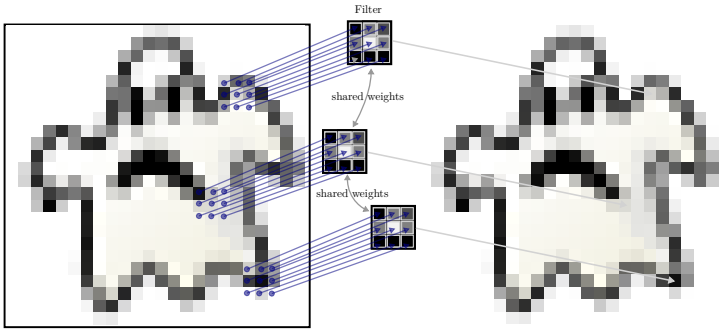



**Figure 7.2. Convolutions** Convolutions performed on an image are shown. The same filter is applied over the image at every position by calculating the dot product between the pixel values and the weight values element-wise, as shown in Eqn. (7.34). This results in efficient and powerful computations which are used in convolutional neural networks.

in a layer increases, since *all* neurons in two consecutive layers are connected to each other. This is especially a problem for images, where the input is of type:

$$x \in \mathbb{R}^{W \times H \times C} \tag{7.33}$$

where $W$ is the width of the image, $H$ the height and $C$ the number of channels (typically red, green and blue for a colour image). Even a low-resolution colour image of $640 \times 480$ pixels has about a million input values; multiplied with the number of say 500 hidden units (not atypical) this results in half a billion computations for the first layer!

Luckily, researchers have thought of smarter ways to connect neural networks for image-based input. The key observation is that images have strong local correlations; i.e. nearby pixels are often similar since they tend to be part of the same object. Inspired by connections in the human visual cortex, Fukushima, 1980 introduced convolutional neural networks (CNN). In a CNN, each neuron in a layer corresponds to a specific location and connects to all neurons in a small region around that location in the previous layer. Thus, in contrast to NNs, neurons in CNNs connect only locally to neurons that correspond to the same neighbourhood. This dramatically reduces the number of connections and thus allows the use of larger hidden layers for the same amount of computation.

Additionally, another useful observation can be made about images: in general, they are translation-invariant. A tree is still a tree if it is moved in an image, and an object can appear in any part of an image since cameras can change direction easily. And, even if most images have positional biases (the sky is typically on top, a road typically on the bottom) we don't want to put these biases in our architectures for

reasons of robustness.

The application of this observation is that the weights of our connections can be shared for all locations, as can be seen in Figure 7.2. The weights in a CNN are referred to as *filters*, following their origin in signal processing.

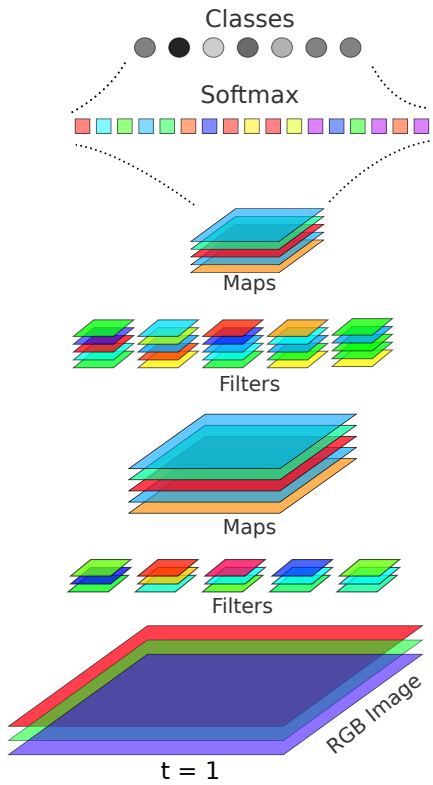



**Figure 7.3. Convolutional Neural Network** A convolutional neural network is shown. The image is convolved with sets of *filters*, where the result of each set is stored in a hidden layer, often called *maps*. After a non-linear operation, these maps are again convolved in the next layer. If a classification is desired, the final layer does not convolve but implements a regular neural network, with the desired number of outputs for classification.

**Convolutions** With these changes we can efficiently perform computations using the discrete 2D-*cross-correlation* function, which due to some misunderstanding has been called the *convolution* function in the machine learning community. The *cross-correlation* and *convolution* are functionally the same and only differ in the alignment of their operations. The weights in a CNN are referred to as *filters*, following the signal processing vocabulary. It is written as:

$$(f * g)[x, y] = \Sigma_{m=-\infty}^{\infty} \Sigma_{n=-\infty}^{\infty} f[x, y]\, g[x + m, y + n] \tag{7.34}$$

where $f$ and $g$ are the input image and filters respectively, and $f[x, y]$ selects the value at position $x, y$ from the image, and if $x, y$ reaches outside the range of the image, it results in a $0$. The ranges for $-\infty$ to $\infty$ seem extreme, but are there to keep the notation general. In the implementation, the ranges that exceed the image or filters will return $0$ and can thus be ignored. To perform a convolution for every pixel, we simply write: $f * g$, without indexing.

**Formalisation** A CNN uses the convolutions instead of the matrix multiplications of a regular neural network. Formally we say the network activations of layer $i$ are $h^i \in \mathbb{R}^{W \times H \times C}$, the next layer activations $h^{i+1} \in \mathbb{R}^{W' \times H' \times C'}$, and $W^i \in \mathbb{R}^{K \times K \times C \times C'}$ are the filters. Here $W$ and $H$ refer to the width and height of the hidden layers, $C$ and $C'$ the number of channels in layers $i$ and $i + 1$, and $K$ to the width and height of the convolution filters.

Note that the activation layers in CNNs have a three-dimensional structure, instead of the one-dimensional vector of a regular NN. To deal with this complexity we introduce the following notation to denote we take one *slice* of the activations $h^i$ at channel $c$:

$$h^i_{\bullet,\bullet,c} \tag{7.35}$$

The size of this object is $h^i_{\bullet,\bullet,c} \in \mathbb{R}^{W \times H}$ and is referred to as a *map* in CNN parlance.

Then the next layer is computed by:

$$h^{i+1}_{\bullet,\bullet,c'} = \Sigma_c^C f(h^i_{\bullet,\bullet,c} * W^i_{\bullet,\bullet,c,c'} + b^i) \tag{7.36}$$

where $b^i \in \mathbb{R}^{C'}$ is a bias that is added element-wise, $f$ is a non-linear function applied element-wise, and $\bullet$ denotes the free parameters for the convolution. The parameters of a CNN are denoted by $\theta = \{(W^i, b^i) \mid i = 1 \cdots N\}$.

We let the activations of the lowest layer be the input $h^0 = x$. Then, the output of each successive hidden layer is calculated in the same way as was done for the NN in Section 7.3. Using the economic use of connections, a CNN can process images much more efficiently and use many more hidden units than is possible with an NN.

### 7.5.1 Maxout Pooling

An additional layer for convolutional neural networks that will be used in Chapter 8 is the *Maxout pooling* layer [Goodfellow et al., 2013]. The idea of *Maxout pooling* is to reduce the number of channels by only keeping the maximum value for local

groups of channels. For a hidden layer $h^i$, it is calculated as

$$\tilde{h}^i_{x,y,c'} = max_{0<n<\frac{C}{C'}} h^i_{x,y,c'\cdot\frac{C}{C'}+n} \tag{7.37}$$

where $h^i \in \mathbb{R}^{W\times H\times C}$ are the hidden values, $\tilde{h}^i \in \mathbb{R}^{W\times H\times C'}$ are the Maxout hidden values, and $C'$ is the number of channels after the Maxout, where $C' < C$ and $C \bmod C' \equiv 0$ ($C$ is divisible by $C'$). What Maxout achieves is adding an invariance to the network by allowing it to ignore values with lower activations. A CNN with Maxout will be used in Chapter 8.

## 7.6 Recurrent NNs and Long Short-Term Memory

Until now we have discussed NN architectures that deal with static inputs, like images, where there is no concept of change over time. However, in the world of robotics inputs are constantly changing, and time is an important factor. Recurrent Neural Networks (RNN) are architectures that are designed to deal with such inputs. The RNN is schematically drawn in Figure 7.4.

Say we are given a sequence where at every time step $t$ we get an input $x_t \in \mathbb{R}^X$, where $X$ is the dimensionality of the input. This results in a stream of vectors $x_0 \dots x_T$, where $T$ is the length of the sequence. These inputs $x_t$ can be images from a video-stream or intensities of an audio-signal. The RNN equations are as follows:

$$h_t = f(\theta_{hx}\, x_{t-1} + \theta_{hh}\, h_{t-1} + b_h) \tag{7.38}$$

$$y_t = f(\theta_{yh}\, h_t + b_y) \tag{7.39}$$

where $f$ is a non-linear function, such as the sigmoid or the $\tanh$ function, $h_t \in \mathbb{R}^H$ is the hidden layer at time $t$, $y_t \in \mathbb{R}^Y$ is the output, and $W_{hx} \in \mathbb{R}^{H\times X}$, $W_{hh} \in \mathbb{R}^{H\times H}$ and $W_{yh} \in \mathbb{R}^{Y\times H}$ are weight matrices and $b_h \in \mathbb{R}^H$ and $b_y \in \mathbb{R}^Y$ are bias vectors. It is important to notice that $h_t$ depends on $h_{t-1}$ and thus is recursively connected, which is where RNNs get their name. Say we are also given a desired output at every time step $\hat{y}_0 \dots \hat{y}_T$. We can then create an error function over the time period:

$$E_t = (y_t - \hat{y}_t)^2 \tag{7.40}$$

$$E = \Sigma_0^T E_t \tag{7.41}$$

With this error function, backpropagation can be applied to RNNs as described in Section 7.3.1 and trained using the algorithms described in Section 7.4.

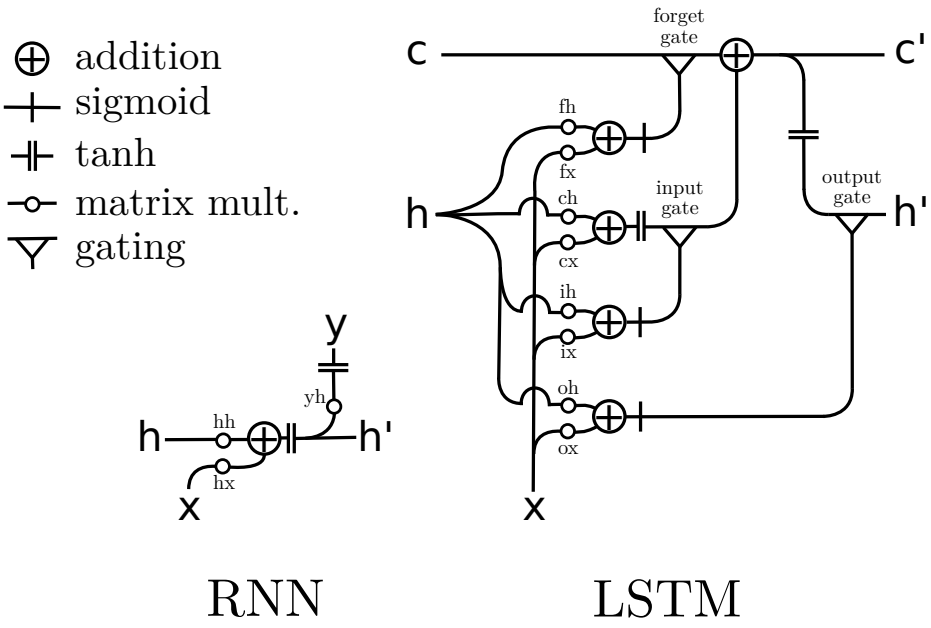



**Figure 7.4. RNN and LSTM** A graphical representation of the RNN and LSTM networks are shown. The inputs are denoted by $x$, the hidden units by $h$, and the output units by $y$. For the LSTM, the hidden unit $h$ functions as the output unit. The accent ($'$) denotes a value in the next time step $h' = h_{t+1}$; otherwise the value is current $h = h_t$. The RNN has a much simpler structure and lacks gating operations. The LSTM has several gating mechanisms, allowing it to propagate information efficiently over long periods of time. If the forget gate is on and the input gate off, the line from $c$ to $c'$ essentially copies information. By switching on the input gate, information can be added to $c'$, and by switching off the forget gate, the information from $c$ will not reach $c'$ as is forgotten. These gating mechanisms let the LSTM control the flow of information and propagate it over long time ranges.

**LSTM** To improve on RNNs, Long Short-Term Memory (LSTM) networks were conceived [Hochreiter and Schmidhuber, 1997; Gers et al., 1999]. LSTMs are designed to learn long-range time dependencies. They achieved state-of-the-art results on challenging tasks such as handwriting recognition [Graves et al., 2009], large vocabulary speech recognition [Sak et al., 2014b] and machine translation [Sutskever et al., 2014].

The LSTM adds an extra variable, called the 'cell', that can store information indefinitely. By switching several *gates on* or *off*, the LSTM can decide whether to *store* new information, *forget* the old information, or *output* information, from that cell. These gates allow the LSTM to retain information over long time intervals, and

are implemented through simple element-wise multiplications:

$$\hat{x}_i = x_i \cdot g_i \tag{7.42}$$

where $x$ is a vector of activations, $g$ is a vector of gate values, and $\hat{x}$ is the result after gating. If $g_i = 0$, no information reaches $\hat{x}_i$ and if $g_i = 1$ the value is unaltered $\hat{x}_i = x_i$.

A schematic drawing of the LSTM is shown in Figure 7.4. An LSTM unit consists of an input gate ($i$), forget gate[2] ($f$), output gate ($o$), and memory cell ($c$) which control what should be remembered or forgotten over potentially long periods of time. The input $x$ and all gates and activations are real-valued vectors: $x_t \in \mathbb{R}^X$ and $i_t, f_t, \tilde{c}_t, c_t, o_t, h_t \in \mathbb{R}^H$, for time $0 \leq t \leq T$, where $H$ is the number of hidden units, and $T$ is the length of the input. The gates and activations at discrete time $t = 1, 2, \cdots, T$ are computed as follows:

$$i_t = \sigma(\theta_{ix} \cdot x_t + \theta_{ih} \cdot h_{t\text{-}1} + \theta_{i_{bias}}) \tag{7.43}$$

$$f_t = \sigma(\theta_{fx} \cdot x_t + \theta_{fh} \cdot h_{t\text{-}1} + \theta_{f_{bias}}) \tag{7.44}$$

$$\tilde{c}_t = \tanh(\theta_{\tilde{c}x} \cdot x_t + \theta_{\tilde{c}h} \cdot h_{t\text{-}1} + \theta_{\tilde{c}_{bias}}) \tag{7.45}$$

$$c_t = \tilde{c}_t \odot i_t + c_{t\text{-}1} \odot f_t \tag{7.46}$$

$$o_t = \sigma(\theta_{ox} \cdot x_t + \theta_{oh} \cdot h_{t\text{-}1} + \theta_{o_{bias}}) \tag{7.47}$$

$$h_t = o_t \odot \tanh(c_t) \tag{7.48}$$

where $(\cdot)$ is a (matrix) multiplication, $(\odot)$ an element-wise multiplication, and $\theta$ denotes the weights. $\tilde{c}$ is the input to the 'cell' $c$, which is gated by the input gate, and $h$ is the output. The non-linear functions $\sigma$ and $\tanh$ are applied element-wise, where $\sigma(x) = \frac{1}{1+e^{-x}}$.

As can be seen in the equations and Figure 7.4, there are several gating mechanisms that allow for efficient propagation of information, *both forward and backward* in time. The line connecting the current *cell state* $c$ with the next cell state $c'$ runs through the *forget gate*. When this gate is turned on, information can flow freely to the next time step. The *input gate* controls the flow of new information into the cell, and the *output gate* controls the read-out of information. These mechanisms allow the LSTM to control information flow and retain information over long-range time ranges.

[2] Although the forget gate output is inverted and actually 'remembers' when it is on, and forgets when it is off, the traditional nomenclature is kept.

## 7.7 Multi-Dimensional Long Short-Term Memory

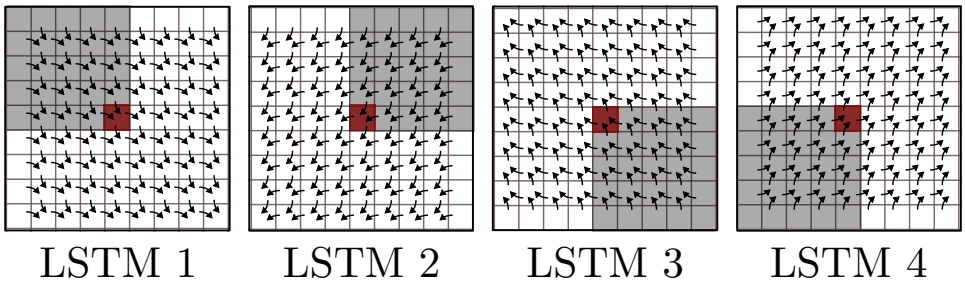



**Figure 7.5. Multi-Dimensional LSTM** The 2D Multi-Dimensional LSTM is connected such that it follows the topology of an image. Since the connection needs to be a directed acyclic graph, such that no loops occur in the computation, four separate LSTMs are used to process the full image. The four separate LSTMs are combined by summing their activations. The *grey* squares denote the inputs that affect the activations in the *red*; this shows that all four directions are needed to calculate the context of the *red* square.

LSTM networks can learn dependencies over long time ranges in sequence data. However, such data are necessarily one-dimensional with this dimension being time. To carry these properties over to different kinds of data with more dimensions, Multi-Dimensional LSTMs (MD-LSTM) were invented [Graves and Schmidhuber, 2009]. Three changes are made:

1. The LSTM unit is allowed to get input from M dimensions, instead of only one. This is done by adding weight matrices for each direction and summing the result.

2. An order is defined over the multi-dimensional input data, which makes sure the order of computations forms an acyclic graph.

3. Due to the constraints on the ordering, multiple LSTM units are needed to account for all information directions, as shown in Figure 7.5.

Figure 7.5 shows the four orderings needed to process 2-dimensional data, such as an image. Each direction has its own LSTM unit and weights.

The MD-LSTM has great applications for different tasks such as image segmentation [Byeon et al., 2014; Byeon et al., 2015]. In Chapter 9 we introduce a novel adaptation of MD-LSTMs that allows for fast 3D volumetric segmentation.

## 7.8 Concluding Remarks

In this chapter, we have described the neural network and the recurrent neural network and showed how to train them. The convolutional neural network is designed for visual processing and pushes the current hardware capabilities to their limits. This has resulted in state-of-the-art performance in many tasks.

However, we see a need for more refined control of the visual computations *by the algorithm itself*. We think such dynamism is needed for humanoid applications since their environment is inherently dynamic and thus, there is a need to control their visual processing. In the following chapters, we will implement several methods that implement dynamic visual processing in different ways. In Chapter 8 we develop a CNN that can focus its attention on specific features of an image with a feedback mechanism to improve recognition performance. And, in Chapter 9 we describe an adapted version of MD-LSTMs that can perform fast computations on 3D volumes for segmentation in bio-medical images. Finally Chapter 10 shows an application combining our work in humanoid control and visual processing algorithms. Chapter 11 concludes the thesis.

# Chapter 8

# Deep Attention Selective Neural Networks

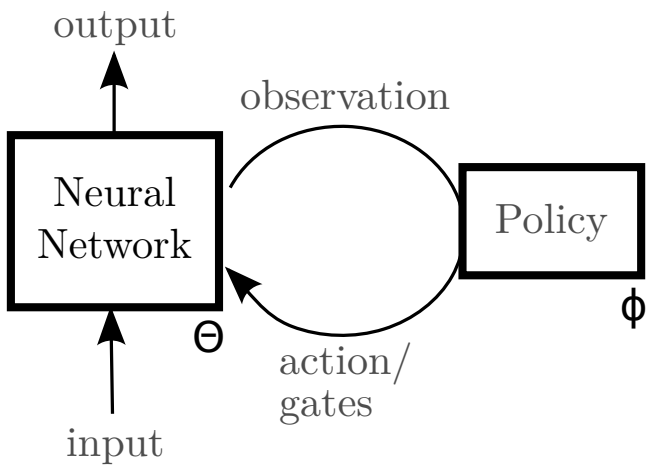



**Figure 8.1. Schematic Representation of dasNet** A schematic representation shows the basic workings of dasNet. The neural network takes an input and outputs the result. But this result is altered by the policy, which observes the activations in the network, and outputs a gating action to steer the computations in the neural network.

What opportunities arise for computer vision algorithms by the increase in computational power? The biggest advances in computer vision in recent years have been CNNs which push the hardware capabilities of GPUs to their limits, resulting in state-of-the-art results. However, traditional CNNs are feed-forward neural networks, processing an input image solely in one direction through the network, resulting in a classification.

This process is inherently *static*; the flow of information is acyclic and never inter-

acts with itself. In contrast, human vision is *dynamic* and extensively uses *feedback*; information paths that flow backwards and affect the computations done on that same information. In other words, the flow of information interacts with and affects itself.

The effects of dynamic processing is seen in humans. If an image is unclear we take a longer time to process it. In an experiment with ornithologists, it was shown the classification of birds that belong to one of two similar species took much longer [Branson et al., 2010; Welinder et al., 2010]. Such response times are not explained by single feed-forward processing which takes the same amount of time regardless; it seems there is a feedback process where the image is revisited to elicit more information from the image, in the light of what has been established in the first processing step. Since humans benefit greatly from this strategy, we hypothesise CNNs can too.

In this chapter, we develop Deep Attention Selective Neural Networks (*dasNet*) which adds dynamic processing capabilities to a CNN. We perform the following changes to a regular CNN (see Figure 8.1):

1. A gating mechanism is added on the filters such that they can be strengthened or weakened. This feedback mechanism affects the computational processing of the CNN and works as an attentional mechanism.

2. A policy is added which steers the attentional mechanism. It observes the current state of activation of the CNN, and outputs an action vector. This action vector is directly fed into the gating mechanism, closing the loop in the dynamics.

We will first formalise *dasNet* and show how it is trained. Then experiments are performed to show its performance. The *dasNet* architecture was developed in joint work with Jonathan Masci[1] and published in a shared first author publication [**M. F. Stollenga* and J. Masci*** et al., 2014]. It achieved state-of-the-art results at the time of publication. Parts of that publication appear in this chapter.

## 8.1 Formalisation

The power of *dasNet* comes from its dynamics, allowing it to sequentially process an image and balance the filter strengths in each iteration. This creates a dynamic attention mechanism where the focus is put on certain features by strengthening and weakening filters. Thus, at every iteration, the same image is viewed, but under a different light, possibly resulting in a different classification. The experiments will show this improves classification performance.

[1] Both authors contributed equally to this work.

**Construction** The *dasNet* is build from a regular CNN. A Maxout CNN (see Section 7.5.1), denoted by $\mathbf{N}$, is augmented to allow the filters to be weighted differently on different passes over the same image (compare to Eqn. (7.36)):

$$h^{i+1}_{\bullet,\bullet,c'} = a^{i+1}_{c'} \Sigma^{C}_{c} ReLu((h^{i}_{\bullet,\bullet,c} * W^{i}_{\bullet,\bullet,c,c'}) + b^{i}) \tag{8.1}$$

where $a^{\ell}_{j}$ is the gate value of the $j$-th output map in layer $\ell$, changing the strength of its activation, *before* applying the Maxout pooling operator, $ReLu$ is the rectified linear function (see Notation) and $\bullet$ denotes the free dimensions. The vector of gates at time $t$ is put together in one vector: $\mathbf{a}_t = [a^0_0, a^0_1, \cdots, a^0_{C'}, a^1_0, \cdots, a^1_{C'}, \cdots]$. This gating vector is the output of a policy $\pi_\phi(x_t) \rightarrow \mathbf{a}_t$, where $\phi$ are the policy parameters, and gating values $\mathbf{a}$ are seen as the *actions*.

The output of the network $\mathbf{N}$ now depends on the gating values $\mathbf{a}$, indicated with the following notation:

$$\mathbf{y}_t = \mathbf{N}_\theta(\mathbf{a}_t, x) \tag{8.2}$$

where $\theta$ is the NN parameter vector, and $\mathbf{y}_t$ is the output of the network on pass $t$.

**Policy** The policy $\pi$ is responsible for choosing the gate values for the next iteration. This choice is based on the activation of the network at the current iteration. To do this, the current state of the network is *observed* by creating an observation vector, $\mathbf{o}_t$. This vector contains the following values:

1. the average activation of *every* output map $Avg(h^i_j)$, of each Maxout layer.

2. the pre-classification activations $h^{N-2}$.

3. the class probabilities, $y_t$.

These values give a summary of the current activations. The policy function $\pi_\phi$ then chooses an action as follows:

$$\mathbf{a_t} = \pi_\phi(\mathbf{o_t}) = dim(A) \cdot \sigma(\boldsymbol{\phi} \cdot \mathbf{o_t}) \tag{8.3}$$

where $\boldsymbol{\phi} \in \mathbb{R}^{dim(A) \times dim(O)}$ is the weight matrix of the neural network, and $\sigma$ is the softmax function. Every action value is positive $a_i > 0$ and the average action is equal to 1: $Avg(a) = 1$. This ensures that the dynamics of *dasNet* are stable and prevents exploding activations in the NN.

## 8.2 Training dasNet

There are two sets of parameters to learn in a dasNet; the neural network parameters $\theta$ and the policy parameters $\phi$. We train these parameters in two steps. First, the network parameters are trained using SGD following the exact steps of the original Maxout paper [Goodfellow et al., 2013].

Once the network parameters $\theta$ are trained, they are used in a *dasNet* and the policy parameters $\phi$ need to be optimised. The search space of policy parameters is very large because our action vector and state vector will contain close to one thousand values, the policy will contain hundreds of thousands of parameters! Additionally, the search space is very complex since it involves complex dynamics and interactions through the feedback process. To deal with this large search space, we apply SNES (see Section 3.2); the scalable version of the NES algorithm that we used in our work in robotic control. Using such optimisation algorithms to optimise a policy is known as direct policy search and has seen many applications in robotics [Peters and Schaal, 2006; Peters and Schaal, 2008]. Algorithm 8 describes the *dasNet* training algorithm.

Each generation starts by selecting a subset of $n$ images from X at random. Then $p$ samples are drawn from the SNES search distribution (with mean $\mu$ and covariance $\Sigma$) representing the parameters, $\phi_i$, of a policy, $\pi_{\phi_i}$. Each sample undergoes $n$ trials, one for each image in the batch. The update parameters are set to $\eta_\mu = .8$ for the centre, and $\eta_\Sigma = .3$ for the covariance. The population size is $p = 50$, and the batch size is 128.

**Boosting** The full *dasNet* process is illustrated in Figure 8.2. During a trial, the image is presented to the Maxout net $T$ iterations, allowing the feedback to function. In the first pass, $t = 0$, the action, $\mathbf{a}_0$, is set to $a_i = 1, \forall i$, so that the Maxout network functions as it would normally. On the next pass, the same image is processed again, but this time, the gates $\mathbf{a}_1$ are changed. This cycle is repeated until pass $T$, at which time the performance of the network is scored by:

$$L_i = -\lambda_{boost}(d)\log(y_T) \tag{8.4}$$

$$\tag{8.5}$$

$$\lambda_{boost}(d) = \begin{cases} \lambda_{correct} & \text{if } d = \|\mathbf{v}_T\|_\infty \\ \lambda_{misclassified} & otherwise \end{cases} \tag{8.6}$$

where $\mathbf{v}$ is the output of $\mathbf{N}$ at the end of the pass $T$, $d$ is the correct classification, and $\lambda_{correct}$ and $\lambda_{misclassified}$ are constants.

**Algorithm 8 TrainDasNet** Trains a dasNet given a trained network. It uses SNES to learn dynamic attention by optimising policy parameters.

```
function TrainDasNet(Network N, Initial distribution parameters μ, Σ, population size K, dasNet timesteps T)
    while True do
        x ⇐ NextBatch()
        for i = 0 → p do
            φ_i ~ 𝒩(μ, Σ)                          ▷ Sample new policy parameters
            for j = 0 → n do                        ▷ Iterate over batch
                a_0 ⇐ 𝟙                            ▷ Initialise gates a with identity activation
                for t = 0 → T do
                    y_t = N(a_t, x_j)               ▷ Evaluate network given actions a_t
                    o_t ⇐ h(N_t)                    ▷ Create observation vector o_t
                    a_{t+1} ⇐ π_{φ_i}(o_t)          ▷ Calculate new actions given policy
                end for
                L_i = −λ_boost(d) log(y_T)          ▷ Calculate loss
            end for
            ℱ[i] ⇐ −L_i                             ▷ Store negative loss as fitness value
        end for
        μ, Σ ← UpdateSNES(ℱ, φ, μ, Σ)               ▷ Update SNES distribution parameters
    end while
end function
```

$L_i$ measures the weighted loss, where misclassified samples are weighted higher than correctly classified samples $\lambda_{misclassified} > \lambda_{correct}$. This simple form of boosting is used to focus on the 'difficult' misclassified images. Once the input images have been processed, the policy is assigned the fitness:

$$f(\phi_i) = \overbrace{\sum_{i=1}^{n} L_i}^{\text{cumulative score}} + \overbrace{\lambda_{\ell^2} \|\phi_i\|_2}^{\text{regularisation}} \tag{8.7}$$

where $\lambda_{\ell^2}$ is a regularisation parameter. Once the candidate policies have been evaluated, SNES updates its distribution parameters $(\mu, \Sigma)$ according to the natural gradient calculated from the sampled fitness values $\mathscr{F}$. As SNES repeatedly updates the distribution over the course of many generations, the expected fitness of the distribution improves, until the stopping criterion is met, i.e. no improvement is made for several consecutive epochs.

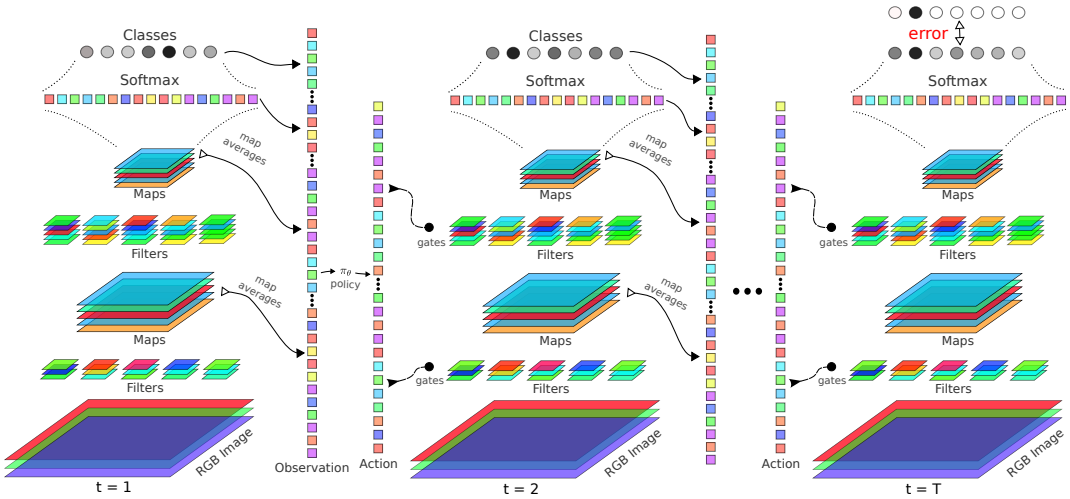



**Figure 8.2. The dasNet Network** Each image is classified after $T$ passes through the network. After each forward propagation through the Maxout net, the output classification vector, the output of the second to the last layer and the averages of all feature maps are combined into an observation vector used by a deterministic policy to choose an action that changes the weights of all the feature maps for the next pass of the same image. After pass $T$, the output of the Maxout net is finally used to classify the image.

## 8.3 Experiments

The network architecture of **N** is directly taken from Goodfellow et al., 2013: Several convolution layers are used, with $192$ maps in layer 1 with $96$ layers after Maxout, $384$ maps in layer 2 with $192$ maps after Maxout, again $384$ maps in layer 3 with $192$ after Maxout, projecting to a $2500$ dimensional vector in layer 4, which is scaled down to $500$ using Maxout. The final classification layer projects to $10$ values corresponding to the $10$ classes. The result is a large observation vector $dim(o) = 192+384+384+500+10 = 1470$ and action vector $dim(a) = 96 + 192 + 192 = 480$. The number of policy parameters is $|\phi| = 1470 \times 480 = 705600$

The experimental evaluation of *dasNet* focuses on ambiguous classification cases in the CIFAR-10 and CIFAR-100 data sets where, due to a high number of common features, two classes are often mistaken for each other. These are the most interesting cases for our approach. By learning on top of an already trained model, *dasNet* must aim at fixing these erroneous predictions without disrupting, or forgetting, what has been learned. The CIFAR-10 dataset [Krizhevsky, 2009] is composed of $32 \times 32$ colour images split into $5 \times 10^4$ training and $10^4$ testing samples, where each image is assigned to one of $10$ classes. The CIFAR-100 is similarly composed but contains $100$ classes. The number of steps was experimentally determined and fixed at $T = 5$; small

**Table 8.1. Classification results on CIFAR-10 and CIFAR-100 datasets**. The error on the test-set is shown for several methods (lower is better). Note that the result for Dropconnect is the average of 12 models. Our method improves over the state-of-the-art reference implementation to which feedback connections are added. The recent Network in Network architecture [Lin et al., 2013] has better results when data-augmentation is applied.

| Method | CIFAR-10 | CIFAR-100 |
|---|---|---|
| Dropconnect [Wan et al., 2013] | 9.32% | - |
| Stochastic Pooling [Zeiler and Fergus, 2013] | 15.13% | - |
| Multi-column CNN [Ciresan et al., 2012b] | 11.21% | - |
| Maxout [Goodfellow et al., 2013] | 9.38% | 38.57% |
| Maxout (trained by us) | 9.61% | 34.54% |
| **dasNet** | 9.22% | **33.78**% |
| NiN [Lin et al., 2013] | 10.41% | 35.68% |
| NiN (augmented) | **8.81**% | - |

enough to be computationally tractable while still allowing for enough interaction. In all experiments we set $\lambda_{correct} = 0.005$, $\lambda_{misclassified} = 1$ and $\lambda_{\ell^2} = 0.005$.

The Maxout network, $\mathbf{N}$, was trained with the same method as the original authors used (see Goodfellow et al., 2013 for details). The model consists of three convolutional Maxout layers followed by a fully connected Maxout layer and softmax outputs. Then, *dasNet* was trained as described above. The population for SNES was set to $K = 50$, with update parameters $\eta_\mu = 0.8$ and $\eta_\Sigma = 0.3$. Training of *dasNet* took around 4 days on a GTX 560 Ti GPU, excluding the original time used to train $\mathbf{N}$.

Table 8.1 shows the performance of *dasNet* vs. other methods, where it achieves a relative improvement of $6\%$ with respect to the vanilla CNN. This established a new state-of-the-art result at the time of publication for this challenging dataset, for unaugmented data. Figure 8.4 shows the classification of a cat image from the test-set. All output map activations in the final step are shown at the top. The difference in activations compared to the first step, i.e., the (de-)emphasis of each map, is shown on the bottom. On the left are the class probabilities for each time-step. At the first step, the classification is 'dog', and the cat could indeed be mistaken for a puppy. Note that in the first step, the network has not yet received any feedback. In the next step, the probability for 'cat' goes up dramatically and subsequently drops a bit in the following steps. The network has successfully disambiguated a cat from a dog. If we investigate

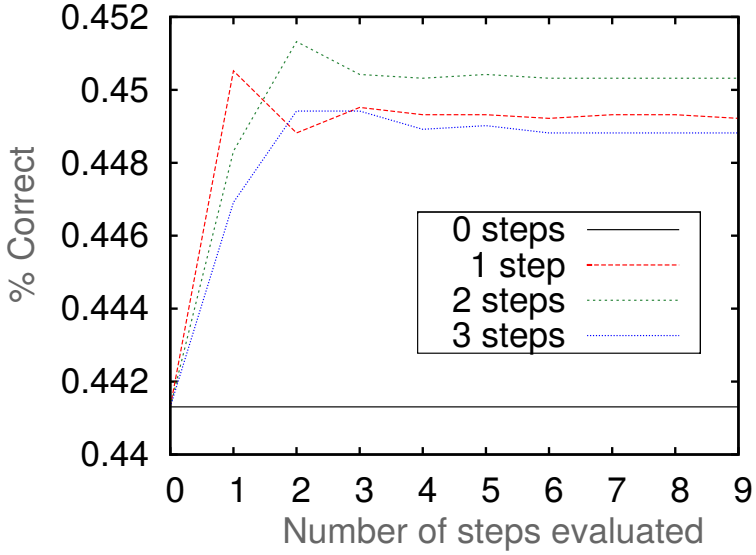



**Figure 8.3. Dynamics of dasNet** Two *dasNets* were trained on CIFAR-100 for different values of $T$. Then they were allowed to run for [0..9] iterations for each image. The performance peaks at the number of steps that the network is trained on, after which the performance drops, but does not explode, showing the dynamics are stable.

the filters, we see that in the lower layer emphasis changes significantly (see 'change of layer 0'). Some filters focus more on surroundings whilst others de-emphasise the eyes. In the second layer, almost all output maps are emphasised. In the third and highest convolution layer, the most complex changes to the network can be seen. At this level, the positional correspondence is largely lost, and the filters are known to code for 'higher level' features. It is in this layer, that changes are the most influential because they are closest to the final output layers.

It is hard to qualitatively analyse the effect of the alterations. If we compare each final activation in layer 2 to its corresponding change (see Figure 8.4, right), we see that the activations are not simply uniformly enhanced. Instead, complex suppression and enhancement patterns are found, increasing and decreasing activation of specific pixels. Another classification is shown in Figure 8.5, where an unclear image of a dog has to be classified. A competition can be seen between the classes 'cat', 'dog' and 'frog'. The 'frog' classification is quickly ruled out, but the 'cat' and 'dog' classes keep competing and 'dog' finally wins (correctly). The feedback of *dasNet* does not always improve the classification; in Figure 8.6 the image of a car is misclassified as a truck after *dasNet* incorrectly changes the classification. At the last two time-steps, the classification switches dramatically and shows unstable behaviour. Due to the normalisation for the gates, the activations are bounded, but still this behaviour is

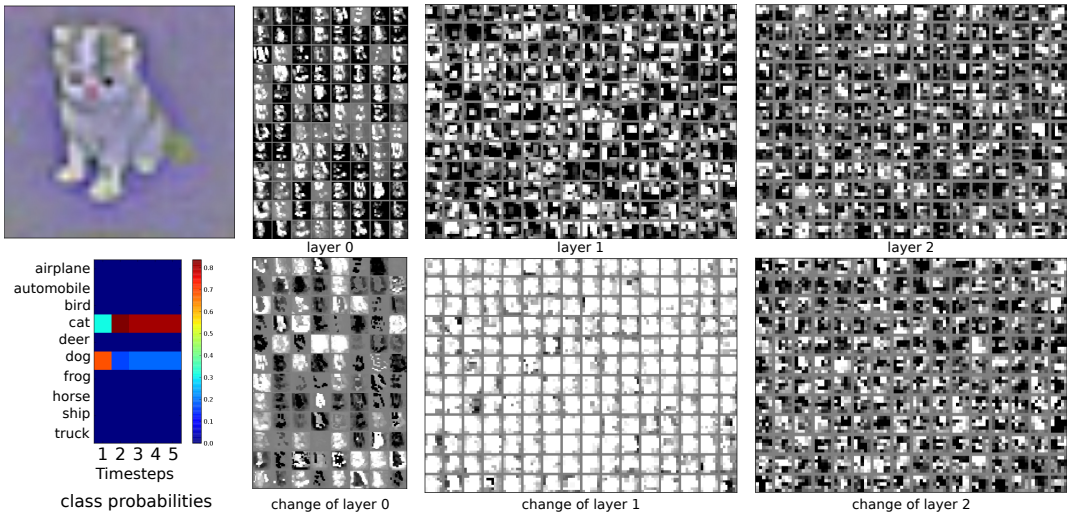



**Figure 8.4. Cat Classification** The classification of a cat by the *dasNet* is shown. All output map activations in the final step are shown on the top. Their changes relative to initial activations in the first step are shown at the bottom (white = emphasis, black = suppression). The changes are normalised to show the effects more clearly. The class probabilities over time are shown on the left. The network first classifies the image as a dog (wrong) but corrects by emphasising its convolutional filters to see it is actually a cat.

undesired. Visualising what these high-level features actually do is an open problem in deep learning.

**Dynamics** To investigate the dynamics, several small 2-layer *dasNets* were trained for different values of $T$. Then they were evaluated by allowing them to run for $[0..9]$ steps. Figure 8.3 shows results of training *dasNet* on CIFAR-100 for $T = 1$ and $T = 2$. The performance goes up from the vanilla CNN, peaks at the $step = T$ as expected, and reduces but stays stable after that. So even though the *dasNet* was trained using a small number of steps, the dynamics stay stable when these are evaluated for as many as 10 steps.

To verify whether the *dasNet* policy is actually making good use of its gates, we estimate their information content by directly using the gate values for classification. The hypothesis is that if the gates are used properly, then their *activation* should contain information that is relevant for classification. For this purpose, a *dasNet* was trained with $T = 2$. Then using *only* the final gate-values (so without e.g. the output of the classification layer), a classification using 15-nearest neighbour and logistic regression was performed. This resulted in a performance of *40.70%* and *45.74%* correct respectively, similar to the performance of *dasNet*, confirming that they contain significant information.

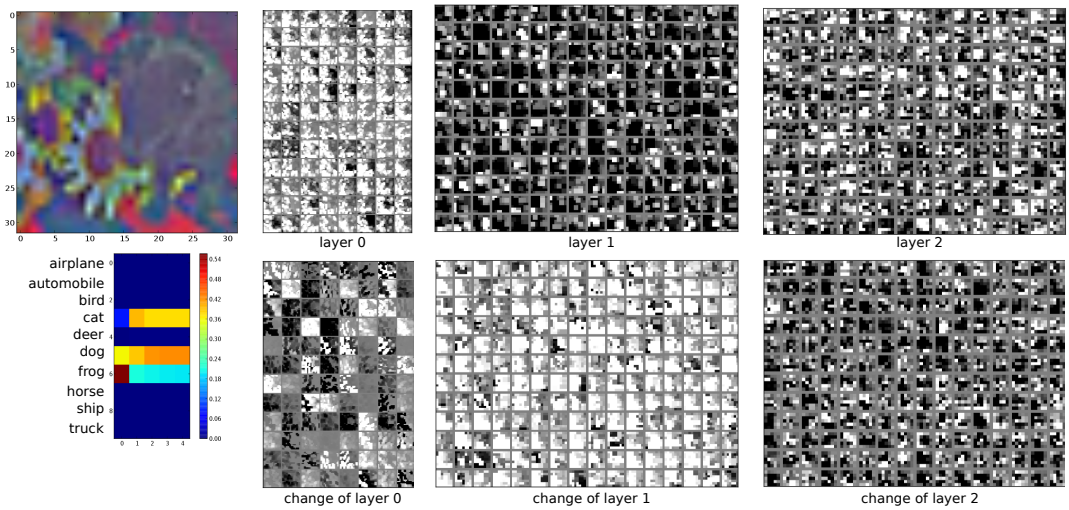



**Figure 8.5. Dog Classification** The classification of the picture of a dog. All output map activations in the final step are shown on the top. Their changes relative to initial activations in the first step are shown at the bottom (white = emphasis, black = suppression). A competition is seen between 'cat', 'dog' and 'frog'. The 'frog' class is quickly ruled out, but competition between the remaining classes keeps going until 'dog' finally wins.

In the Figure 8.5, a very difficult classification is shown. The correct classification is a dog (the dog is on the right part of the image). Initially, without feedback, the class 'frog' has a high activation. After that, the classes 'dog' and 'cat' compete and from step 3 onward, 'dog' has a higher activation. Figure 8.6 shows a misclassification, where initially 'car' has a slightly higher activation, but eventually the network flips and ends up giving 'truck' a strong classification. It shows that the dynamics don't always lead to correct results.

## 8.4 Concluding Remarks

In this chapter, we developed *dasNet*, a CNN with attention through an internal feedback process. Where CNNs already push the computational boundaries of today's hardware, we push them even further by allowing *dasNet* to review the input several times while focusing its attention to different features through feedback. The result is a powerful algorithm that achieved state-of-the-art performance at the time of publication [**M. F. Stollenga* and J. Masci*** et al., 2014].

Such feedback allows for dynamic visual processing that guides the computational power such that it is used more efficiently. We believe this is important for applications in robotics where given the dynamic nature of tasks, directed visual processing can

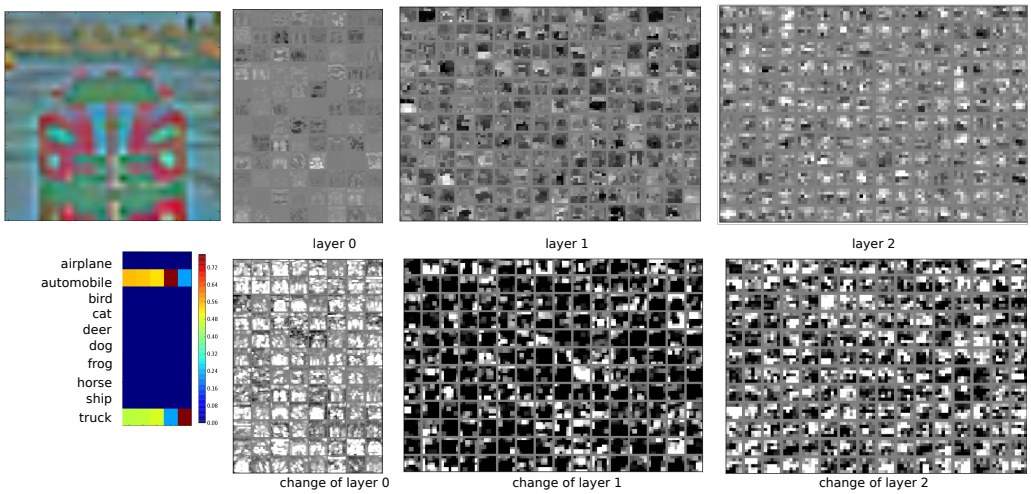



**Figure 8.6. Car Miss-Classification** The misclassification of a car. All output map activations in the final step are shown on the top. Their changes relative to initial activations in the first step are shown at the bottom (white = emphasis, black = suppression). The car is misclassified as a truck. The classification changes dramatically under the feedback in the last time-steps. This shows unstable behaviour, which is undesired. How to prevent such behaviour is an open problem.

result in better use of computational powers. In the next chapter, we will develop the idea of attention further with a different architecture based on the LSTM (Section 7.6).

# Chapter 9

# Pyramidal Multi-Dimensional Long Short-Term Memory Networks

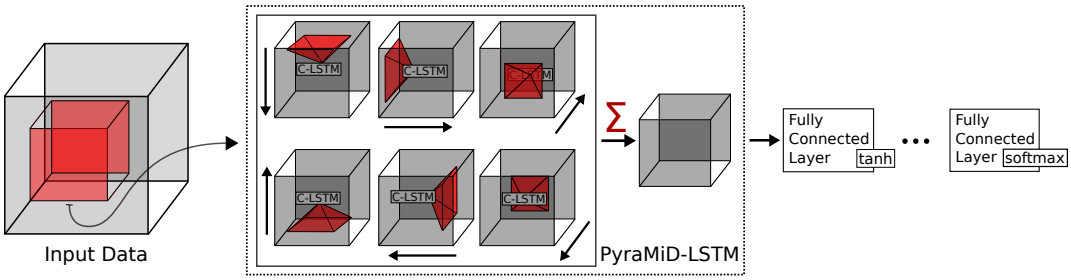



**Figure 9.1. PyraMiD-LSTM Training** Randomly rotated and flipped inputs are sampled from random locations and fed to a PyraMiD-LSTM. Six C-LSTMs perform their computations in six directions, and their results are summed together. The output of the PyraMiD-LSTM layer is sent to the fully-connected layer with a $\tanh$ non-linear function. Several PyraMiD-LSTM layers and fully-connected layers can be applied. The last layer is fully-connected which uses a softmax function to compute probabilities for each class for each pixel.

In the previous chapter, we presented *dasNet*, a CNN with attentional feedback that improved the performance of an already powerful regular CNN. The attentional feedback was implemented by gating activations within the neural network, such that certain features could be emphasised while others were suppressed. We believe such control of feature activations is key to improve the computational power of these architectures through the internal guidance of computational processing power.

In this chapter we develop this idea further, using a different attentional mechanism. In an effort to explore different ways of adding feedback, inspiration came from MD-LSTMs (see Section 7.7), which can sequentially process images by 'scanning' them with an LSTM in an ordered fashion. However, the connection topology of an MD-

LSTM is not optimised for GPU computations, making it hard to use the full hardware capabilities. Due to these issues, there are few implementations of MD-LSTMs and their applications have not gone beyond processing 2D data.

In this chapter, we develop Pyramidal Multi-Dimensional Long Short-Term Memory Networks (PyraMiD-LSTM). The PyraMiD-LSTM architecture introduces a new connection topology that combines powerful convolution operations (Section 7.5) with the advanced gating mechanisms of the LSTM (Section 7.6). This architecture seamlessly translates to an efficient GPU implementation, creating an efficient implementation that can be applied on 3D-data.

In the rest of this chapter we explain this topology change, and then apply it on challenging 3D volumetric segmentation tasks to show the practicality of PyraMiD-LSTM. In Chapter 10 we apply a variant of PyraMiD-LSTM to the iCub robot.

The PyraMiD-LSTM architecture was developed together with Wonmin Byeon[1] and published in a shared first author publication [**M. F. Stollenga* and W. Byeon*** et al., 2015], parts of that publication appear in this chapter. PyraMiD-LSTM achieved state-of-the-art performance on the ISBI NEATBrainS1 [Mendrik et al., 2015] dataset.

## 9.1 Pyramidal Connection Topology

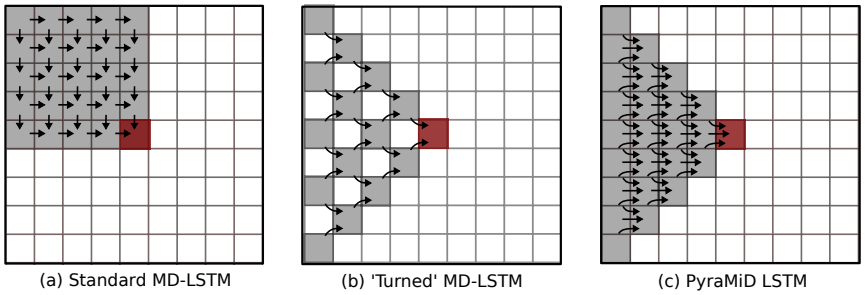



**Figure 9.2. Connection Topology** (a) The standard MD-LSTM topology connects LSTM-units to two neighbours over the axis, recursively resulting in a square context. (b) Rotating the direction of connections by $45°$ results in a pyramidal context with gaps. (c) The resulting gaps are filled by adding extra connections, resulting in a full pyramidal context.

The multi-dimensional LSTM (MD-LSTM; Graves et al., 2007) aligns LSTM-units in a grid and connects them over the axis. Multiple grids are needed to process information from all directions (see Figure 7.5 on page 87). For 2D data, such as an image, four LSTMs are needed such that each pixel receives information from the

[1] Both authors contributed equally to this work.

full image. Figure 9.2–a shows one of these LSTMs that has performed computations up to the red pixel shown. The context of this pixel has the shape of a square.

The PyraMiD-LSTM makes a simple change to the connection topology: The connections of the LSTM are rotated by $45°$, as shown in Figure 9.2–b. All connections are now strictly coming from the left; one from top-left and one from down-left. However, gaps have formed in the context, which are filled by adding an extra connect resulting in the final connection topology (Figure 9.2–c). All connections are strictly coming from the left, which greatly simplifies memory access and facilitates an efficient implementation on GPUs.

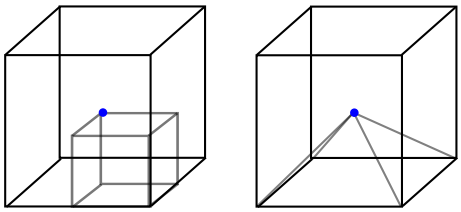


**Figure 9.3. Pyramidal Context** On the **left**, we see the context scanned so far by one of the eight LSTMs of a 3D MD-LSTM: a cube. In general, given $d$ dimensions, $2^d$ LSTMs are needed. On the **right**, we see the context scanned so far by one of the six LSTMs of a 3D-PyraMiD-LSTM: a pyramid. In general, $2 \times d$ LSTMs are needed.

The change in topology between an MD-LSTM and a PyraMiD-LTSM is not just an implementation difference. It has big implementations in the shape of the context, which is best shown in the case of 3D data. Figure 9.3 shows the context for 3D data, for (a) an MD-LSTM and (b) a PyraMiD-LSTM. The context of an MD-LSTM forms a *cube* while the context of a PyraMiD-LSTM forms a *pyramid*. Therefore, an MD-LSTM needs eight LSTMs to scan a volume, while a PyraMiD-LSTM needs only six since it takes eight cubes or six pyramids to fill a volume. Given $d$-dimension data, the number of needed LSTMs grows as $2^d$ for an MD-LSTM (*exponentially*) while the number of needed pyramids grows as $2 \times d$ for a PyraMiD-LSTM (*linearly*)!

**Convolutions** On the implementation side, the big difference between a regular MD-LSTM and the PyraMiD-LSTM is that the latter can use efficient convolutional operations; the same as used in CNNs (Section 7.5). This is a direct result of the change in connection topology that connects to a small neighbourhood of pixels, as shown in Figure 9.2-c. The resulting LSTMs can thus be efficiently implemented using fast convolutional operations as shown in Figure 9.4 (compare to regular LSTM in Figure 7.4, page 85).

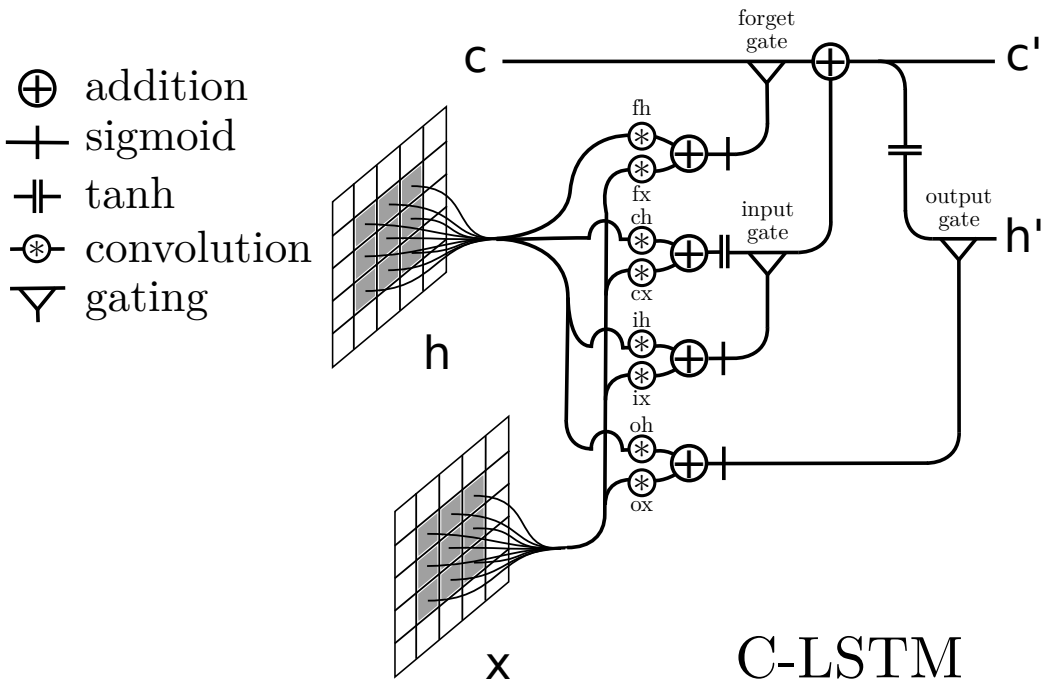



**Figure 9.4. C-LSTM** The convolutional LSTM (C-LSTM) layer is the workhorse of the PyraMiD-LSTM and performs convolutional operations to efficiently calculate contextual information for every pixel. The ability to use convolutions comes from the change in connection topology in Figure 9.2.

## 9.2 Architecture

Here we explain the PyraMiD-LSTM network architecture for 3D volumes (see Figure 9.1). The working horses of PyraMiD-LSTM are six convolutional LSTMs (C-LSTM) layers, one for each direction, that compute the full context of each pixel (see Figure 9.4). Each C-LSTM processes the entire volume in one direction. The directions $\mathscr{D}$ are formally defined over the three axes $(x, y, z)$:

$$\mathscr{D} = \{(\cdot,\cdot,1),(\cdot,\cdot,-1),(\cdot,1,\cdot),(\cdot,-1,\cdot),(1,\cdot,\cdot),(-1,\cdot,\cdot)\} \tag{9.1}$$

Here $\cdot$ denotes over which axis the convolution takes place, and $1$ or $-1$ refers to the direction of computation over the third axis. They essentially choose which axis is the *time* direction; i.e. with $(\cdot,\cdot,1)$ the positive direction of the z-axis represents the time.

The input to the PyraMiD-LSTM is $x \in \mathbb{R}^{W\times H\times D\times C}$, where $W$ is the width, $H$ the height, $D$ the depth, and $C$ the number of channels of the input, or the num-

ber of hidden units in second- and higher layers. Similarly, we define the volumes $f^d, i^d, o^d, \tilde{c}^d, c^d, h^d, h \in \mathbb{R}^{W \times H \times D \times O}$, where $d \in \mathscr{D}$ is a direction and $O$ is the number of hidden units per pixel, and $f$, $i$, $o$ and $c$ refer to the *forget gate*, *input gate*, *output gate* and *cell* volumes. Since each direction needs a separate volume, we denote volumes with $(\cdot)^d$.

The time index $t$ selects a slice in direction $d$. For instance, for direction $d = (\cdot, \cdot, 1)$, $v_t^d$ refers to the plane $x, y, z, c$ for $x = 1..X, y = 1..Y, c = 1..C$, and $z = t$. For a negative direction $d = (\cdot, \cdot, -1)$, the plane is the same but moves in the opposite direction: $z = D - t$. A special case is the first plane in each direction, which does not have a previous plane, hence we omit the corresponding computation.

**C-LSTM equations:**

$$i_t^d = \sigma(x_t^d * \theta_{xi}^d + h_{t\text{-}1}^d * \theta_{hi}^d + \theta_{i_{bias}}^d) \tag{9.2}$$

$$f_t^d = \sigma(x_t^d * \theta_{xf}^d + h_{t\text{-}1}^d * \theta_{hf}^d + \theta_{f_{bias}}^d) \tag{9.3}$$

$$\tilde{c}_t^d = \tanh(x_t^d * \theta_{x\tilde{c}}^d + h_{t\text{-}1}^d * \theta_{h\tilde{c}}^d + \theta_{\tilde{c}_{\text{bias}}}^d) \tag{9.4}$$

$$c_t^d = \tilde{c}_t^d \odot i_t^d + c_{t\text{-}1}^d \odot f_t^d \tag{9.5}$$

$$o_t^d = \sigma(x_t^d * \theta_{xo}^d + h_{t\text{-}1}^d * \theta_{ho}^d + \theta_{o_{bias}}^d) \tag{9.6}$$

$$h_t^d = o_t^d \odot \tanh(c_t^d) \tag{9.7}$$

$$h = \sum_{d \in \mathscr{D}} h^d \tag{9.8}$$

where $(*)$ is a convolution[2], and $h$ is the output of the layer. The outputs $h^d$ for all directions are summed together in the last equation.

**Fully-Connected Layer:** The output of our PyraMiD-LSTM layer is connected to a pixel-wise fully-connected (FC) layer, whose output is squashed by the hyperbolic tangent ($\tanh$) function:

$$h_{FC}^{\ell}(x, y, z, c) = \tanh(\sum_{c'} w_{c,c'} h^{\ell-1}(x, y, z, c')). \tag{9.9}$$

This step is used to increase the number of channels for the next layer.

[2]In 3D volumes, convolutions are performed in 2D; in general an n-D volume requires n-1-D convolutions. All convolutions have stride 1, and their filter sizes should at least be $3 \times 3$ in each dimension to create the full context.

**Classification Layer:** The final classification is done using a pixel-wise softmax function:

$$y(x, y, z, c) = \frac{e^{-h(x,y,z,c)}}{\sum_{c'} e^{-h(x,y,z,c')}} \tag{9.10}$$

giving pixel-wise probabilities for each class. Typically, multiple PyraMiD-LSTM layers and fully connected layers are stacked on top of each other, followed by a final classification layer.

## 9.3 Experiments

We evaluate our approach on two 3D bio-medical image segmentation datasets: electron microscopy (EM) and MR Brain images.

**EM dataset** The EM dataset [Cardona et al., 2010] is provided by the ISBI 2012 workshop on Segmentation of Neuronal Structures in EM Stacks [Segmentation of Neuronal Structures in EM Stacks Challenge, 2012]. Two stacks consist of 30 slices of $512 \times 512$ pixels obtained from a $2, \times 2 \times 1.5\ \mu m^3$ micro-cube with a resolution of $4 \times 4 \times 50\ nm^3$/pixel and binary labels. One stack is used for training, the other for testing. Target data consists of binary labels (membrane and non-membrane).

**MR Brain dataset** The MR Brain images are provided by the ISBI 2015 workshop on Neonatal and Adult MR Brain Image Segmentation (ISBI NEATBrainS15) [Mendrik et al., 2015]. The dataset consists of twenty fully annotated Magnetic Resonance Image (MRI) scans. Three different scan methods were used: a 3D T1-weighted scan (T1), a T1-weighted inversion recovery scan (IR), and a fluid-attenuated inversion recovery scan (FLAIR). These three methods emphasise different tissues by different field energies and post-processing methods.

The dataset is divided into a training set with five volumes and a test set with fifteen volumes. Each volume includes 48 slices with $240 \times 240$ pixels ($3mm$ slice thickness). The slices are manually segmented through nine labels: cortical grey matter, basal ganglia, white matter, white matter lesions, cerebrospinal fluid in the extra-cerebral space, ventricles, cerebellum, brainstem, and background. Following the ISBI NEATBrainS15 workshop procedure, all labels are grouped into five classes:

1. Cortical grey matter and basal ganglia (GM).

2. White matter and white matter lesions (WM).

3. Cerebrospinal fluid and ventricles (CSF).

4. Cerebellum and brainstem.

5. Background (anything not in class 1-4).

The NEATBrainS15 procedure ignores class (4) for evaluation.

**Sub-volumes and Augmentation** The full dataset requires more than the 12 GB of memory provided by our GPU, hence, we train and test on sub-volumes. We randomly pick a position in the full data and extract a smaller cube (see the details in *Bootstrapping*). This cube is possibly rotated at a random angle over some axis and can be flipped over any axis. For EM images, we rotate over the z-axis and flipped sub-volumes with 50% chance along x-, y-, and z-axes. For MR brain images rotation is disabled and only flipping along the x direction is considered, since brains are (mostly) symmetric in this direction.

During test-time, rotations and flipping are disabled and the results of all sub-volumes are stitched together using a Gaussian kernel, providing the final result.

**Pre-processing** The scan images show large variability in contrast and brightness due to the nature of MRI. To normalise the input data, the three data-types of the MR Brain dataset are pre-processed as follows:

1. Each input slice is normalised to a mean of zero and variance of one.

2. To remove global brightness changes (e.g. see Figure 9.6-b), from all slices we subtract the Gaussian smoothed images (filter size: $31 \times 31$, $\sigma = 5.0$).

3. Finally, a Contrast-Limited Adaptive Histogram Equalisation (CLAHE) [Pizer et al., 1987] is applied to enhance the local contrast (tile size: $16 \times 16$, contrast limit: 2.0).

The before and after images for the three data types are shown in Figure 9.6. The original and pre-processed images are all used, except the original IR images (Figure 9.6b), which have high variability. We do not apply the complex structural pre-processing methods common in bio-medical image segmentation [Wang et al., 2015].

**Training** We apply RMS-prop [Tieleman and Hinton, 2012] with momentum (see Section 7.4.2). The squared loss was chosen as it produced better results than using the log-likelihood as an error function.

We use a decaying learning rate: $\lambda_{lr} = 10^{-6} + 10^{-2} \cdot \frac{1}{2}^{\frac{epoch}{100}}$, which starts at $\lambda_{lr} \approx 10^{-2}$ and halves every 100 epochs asymptotically towards $\lambda_{lr} = 10^{-6}$. Other hyper-parameters used are $\epsilon = 10^{-5}$, $\lambda MSE = 0.9$, and $\lambda_M = 0.9$.

**Bootstrapping** To speed up training, we run three learning procedures with increasing sub-volume sizes: first, 3000 epochs with size $64 \times 64 \times 8$, then 2000 epochs with size $128 \times 128 \times 15$. Finally, for the EM-dataset, we train 1000 epochs with size $256 \times 256 \times 20$, and for the MR Brain dataset 1000 epochs with size $240 \times 240 \times 25$. After each epoch, the learning rate $\lambda_{lr}$ is updated.

**Experimental Setup** All experiments are performed on a desktop computer with an NVIDIA GTX TITAN X 12GB GPU. Due to the pyramidal topology, all major computations can be done using convolutions with NVIDIA's cuDNN library [Chetlur et al., 2014], which has reported $20\times$ speedup over an optimised implementation on a modern 16 core CPU. On the MR brain dataset, training took around three days, and testing per volume took around 2 minutes.

We use exactly the same hyper-parameters and architecture for both datasets. Our networks contain three PyraMiD-LSTM layers. The first PyraMiD-LSTM layer has 16 hidden units followed by a fully-connected layer with 25 hidden units. In the next PyraMiD-LSTM layer, 32 hidden units are connected to a fully-connected layer with 45 hidden units. In the last PyraMiD-LSTM layer, 64 hidden units are connected to the fully-connected output layer whose size equals the number of classes.

The convolutional filter size for all PyraMiD-LSTM layers is set to $7 \times 7$. The total number of weights is $10,751,549$, and all weights are initialised according to a uniform distribution: $\mathscr{U}(-0.1, 0.1)$.

**Implementation Bug in Convolutional Alignment** The PyraMiD-LSTM implementation for the experiments performed within this chapter contained a bug. This bug was corrected in the experiments in the following Chapter 10. For completeness, we explain the bug here.

The *convolution function* is provided with a pointer to memory containing the input pixels, with all rows of pixels concatenated in one long sequence. The width and height of the image are provided as extra arguments in order to calculate the right index. In the implementation, the width and height were swapped, and thus the wrong indexes were calculated.

In both datasets, the width and height are the same, but the depth has a different value. As a result, the convolutions over the z-axis are correct (since swapping had no effect), but the convolutions over the x- and y-axis are using wrong indexes. Since

**Table 9.1. Results PyraMiD-LSTM** Performance comparison on EM images. Some of the competing methods reported in the ISBI 2012 website are not yet published. Comparison details can be found under `http://brainiac2.mit.edu/isbi_challenge/leaders-board`.

| Group | Rand Err. | Warping Err.($\times 10^{-3}$) | Pixel Err. |
|---|---|---|---|
| Human | 0.002 | 0.0053 | 0.001 |
| Simple Thresholding | 0.450 | 17.14 | 0.225 |
| IDSIA [Ciresan et al., 2012a] | 0.048 | 0.434 | 0.060 |
| DIVE | 0.048 | **0.374** | **0.058** |
| **PyraMiD-LSTM** | **0.047** | 0.462 | 0.062 |
| IDSIA-SCI | 0.0189 | 0.617 | 0.103 |
| DIVE-SCI | 0.0178 | 0.307 | 0.058 |

the index calculation affects both the output and input index calculation in the same way, the indexes within the row of the target pixel are correct, but outside this row the convolution uses wrong and spatially distant parts of the input.

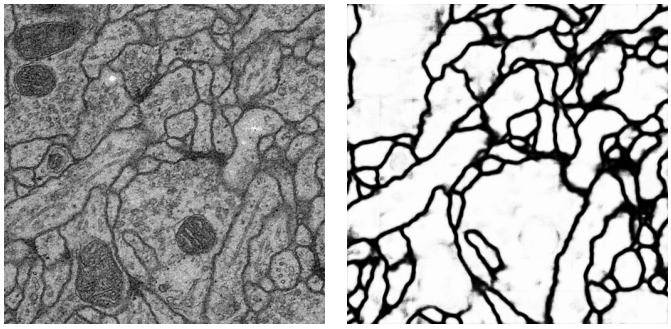
In principle, the weights can learn to adjust for this bug by switching off the 'wrong' indexes by setting the corresponding weights to 0, and only learn using the weights with correct indexes within the row. This allows the PyraMiD-LSTM to still use context information over the x- and y-axis, however not to full effect.

Still, as shown below, PyraMiD-LSTM performs well even with this bug. In the PyraMiD-LSTM architecture used in the next chapter, the bug is corrected.

### 9.3.1 Neuronal Membrane Segmentation

Membrane segmentation is evaluated through an online system provided by the ISBI 2012 organisers. The measures used are the Rand error, warping error, and pixel error [Segmentation of Neuronal Structures in EM Stacks Challenge, 2012]. Comparisons to other methods are reported in Table 9.1. The teams IDSIA and DIVE provide membrane probability maps for each pixel, like our method. Note that the team IDSIA generated the map by one of the state-of-the-art methods [Ciresan et al., 2012a], CNNs, but the team DIVE did not provide the information of their approach. These maps are adapted by the post-processing technique of the teams SCI [Liu et al., 2014], which directly optimises the rand error (DIVE-SCI (top-1) and IDSIA-SCI (top-2)); this is most important in this particular segmentation task.

Without post-processing, PyraMiD-LSTM networks outperform other methods in



**(a)** Input **(b)** Segmentation

**Figure 9.5. Segmentation Results** Segmentation results of PyraMiD-LSTM on the EM dataset (slice 26)

rand error, and are competitive in warping error and pixel error. Figure 9.5 shows an example segmentation result.

### 9.3.2 MR Brain Segmentation

The results are compared using the DICE overlap (DC), the modified Hausdorff distance (MD), and the absolute volume difference (AVD) [Mendrik et al., 2015]. MR brain image segmentation results are evaluated by the ISBI NEATBrain15 organisers [Mendrik et al., 2015] who provided the extensive comparison to other approaches on `http://mrbrains13.isi.uu.nl/results.php`. Table 9.2 compares our results to those of the top five teams. The organisers compute nine measures in total and rank all teams for each of them separately. These ranks are then summed per team, determining the final ranking (ties are broken using the standard deviation). PyraMiD-LSTM leads the final ranking with a new state-of-the-art result and outperforms other methods for CSF in all metrics.

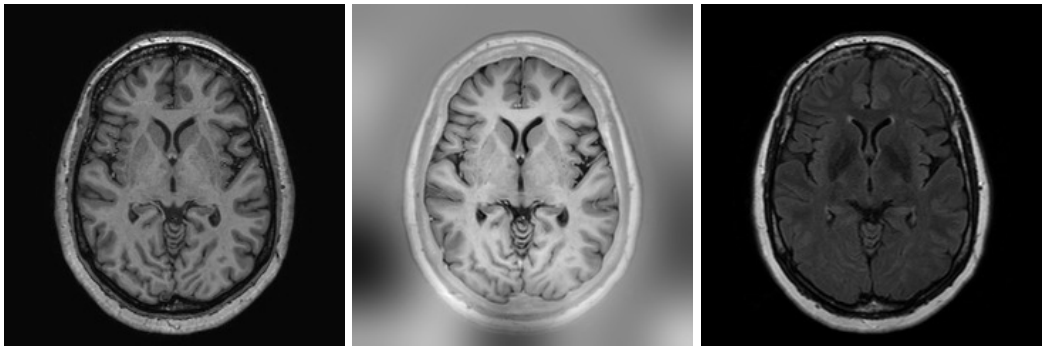
## 9.4 Concluding Remarks

In this chapter we developed PyraMiD-LSTM, a visual processing architecture for multi-dimensional data with LSTM gated convolutional operations. Similar to *dasNet* (Chapter 8), the gating operations allow for the architecture to control the internal processing and decide what gets emphasised and what gets ignored. In contrast to *dasNet*, the adaptive process does not review the same data several times but processed the data in an ordered way where the gating of the operations at a certain pixel is determined by its context.

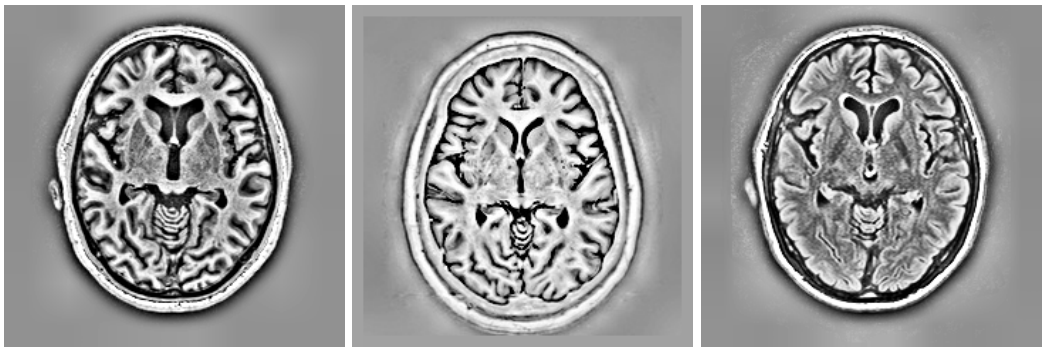
**Table 9.2. PyraMiD-LSTM on MR Brain Images** The performance comparison on MR brain images. The three error measures used are: the dice overlap (DC), the modified Hausdorf distance (MD) and the absolute volume difference (AVD). These are evaluated for the classes: grey matter (GM), white matter (WM) and cerebral spinal fluid (CSF). The rank is determined by summing the ranks of a method in each of these categories.

| Structure | GM | | | WM | | | CSF | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Metric | DC (%) | MD (mm) | AVD (%) | DC (%) | MD (mm) | AVD (%) | DC (%) | MD (mm) | AVD (%) | # |
| BIGR2 | 84.65 | 1.88 | 6.14 | 88.42 | 2.36 | **6.02** | 78.31 | 3.19 | 22.8 | 6 |
| KSOM GHMF | 84.12 | 1.92 | **5.44** | 87.96 | 2.49 | 6.59 | 82.10 | 2.71 | 12.8 | 5 |
| MNAB2 | 84.50 | 1.69 | 7.10 | 88.04 | 2.12 | 7.73 | 82.30 | 2.27 | 8.73 | 4 |
| ISI-Neonatology | **85.77** | **1.62** | 6.62 | 88.66 | **2.06** | 6.96 | 81.08 | 2.66 | 9.77 | 3 |
| UNC-IDEA | 84.36 | **1.62** | 7.04 | **88.69** | **2.06** | 6.46 | 82.81 | 2.35 | 10.5 | 2 |
| **PyraMiD-LSTM** | 84.82 | 1.69 | 6.77 | 88.33 | 2.07 | 7.05 | **83.72** | **2.14** | **7.10** | **1** |

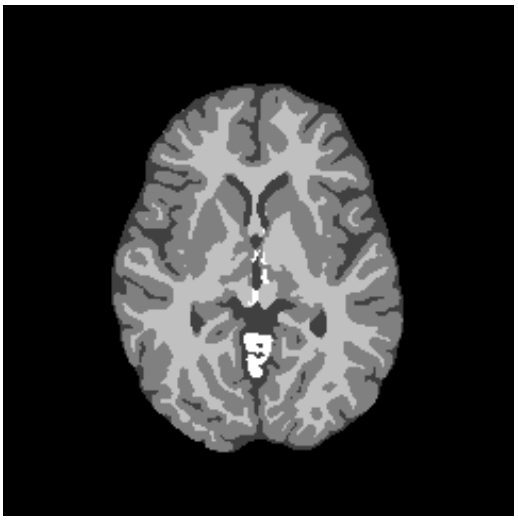
The PyraMiD-LSTM runs on a top of the line Titan X NVIDIA GPU, and pushed its capabilities and memory capacity to its limits, but therefore achieves state-of-the-art performance on the ISBI NEATBrain15 [Mendrik et al., 2015] dataset. In the next chapter, we will develop a version of PyraMiD-LSTM that works with data from the iCub, showing how this method can be applied on a humanoid.



**(a)** T1 **(b)** IR **(c)** FLAIR



**(d)** T1 (pre-processed) **(e)** IR (pre-processed) **(f)** FLAIR (pre-processed)



**(g)** segmentation result from PyraMiD-LSTM

**Figure 9.6. Segmentation of Slice 19, Image 1** (a)-(c) are examples of three scan methods used in the MR brain dataset, and (d)-(f) show the corresponding images after our pre-processing procedure (see pre-processing in Section Section 9.3). Input (b) is omitted due to strong artefacts in the data — the other data-types are all used as input to the PyraMiD-LSTM. The segmentation result is shown in (g).

# Part III

# Integration

# Chapter 10

# Fast-Weight PyraMiD-LSTM

Throughout this thesis, we investigated two aspects of humanoid robotics: *robotic control* and *visual processing*. However, we tackled these directions in isolation from each other. Additionally, the visual processing algorithms have not been applied on the iCub.

In this chapter, we develop a synthesis between the two parts and show how this can be applied on the iCub humanoid. We will show how the robotic control algorithms allow us to easily set up an experiment where the iCub explores movements, and integrate this with our visual processing algorithms in order to build a predictive model.

## 10.1 Introduction

To perform intelligent actions, a humanoid needs to build a model of its world so it can predict the effects of its actions. Given such a model, it could decide which actions are the best and act intelligently. We attempt to build such a model by integrating the dynamic visual processing capabilities that we developed in Chapter 8 and Chapter 9 with the control algorithms developed in Chapter 3 to Chapter 6. These methods provide:

- Flexible control methods to create humanoid movements related to certain tasks.
- Powerful adaptive visual processing algorithms.

We will employ the NGC control algorithm from Chapter 6 to move the robot and capture the video stream coming from the cameras. NGC is chosen since it allows for flexible continuous control and the planning capabilities of the TRM are not

needed in this setting. These images will then be processed by an adapted version of PyraMiD-LSTM (Chapter 9), that is made suited for prediction of the video stream from the iCub's cameras. The images are 2-dimensional, and a stream of them can be concatenated in a 3-dimensional cube. In this 3-dimensional arrangement, the dynamics between consecutive images becomes more natural to process.

We will first describe what adaptations we make to PyraMiD-LSTM to do prediction. Then we will introduce the exploration strategy that uses NGC to generate the data to be processed by the PyraMiD-LSTM. Finally, we show the results of this experiment.

## 10.2 Information Flow

The PyraMiD-LSTM is a powerful method that can capture long-range dependencies in the input data, which is important for prediction. However, the vanilla PyraMiD-LSTM has several problems that become apparent while predicting images:

- For every pixel, the PyraMiD-LSTM can process information from the entire input volume. However, this means that information about future images can leak into parts of the PyraMiD-LSTM that needs to predict those images! It needs to make sure information can only flow forward in time.
- We want our model of the world to incorporate our actions; e.g. what will I see if I move my hand to the left? Thus, we need to find a way to condition the prediction by the PyraMiD-LSTM not only on the previous images but also on the actions taken by the robot.

To remedy the problem of backwards information flow, we only use *one* C-LSTM layer, which flows forward in time, in contrast to the *six* of PyraMiD-LSTM. Additionally, the weights of the C-LSTM will be generated based on the actions chosen, as explained in the next section.

## 10.3 Fast Weights

How can we condition the predictions made by a PyraMiD-LSTM on the actions that are chosen? A solution to this problem comes from 'Fast Weights' [Schmidhuber, 1991]. This method introduces a weight-generating NN that takes as input the *actions* and *outputs the weights* for the predictive model. This idea in incorporated into the PyraMiD-LSTM resulting in Fast-Weight PyraMiD-LSTM. In the Fast-Weight PyraMiD-LSTM, the weights of the C-LSTM layer are generated through a multi-layer

NN. Thus, not the weights of the C-LSTM layer are trained, but the weights of the NN *generating* the weights of the C-LSTM layer!

First, the secondary input data are formed into vectors $o_1, o_2 \cdots o_T$, where $T$ is the length of the sequence and $o_t \in \mathbb{R}^O$ is an observation vector. This vector can incorporate all kinds of information; in this experiment, it contains the difference in joint parameters $\Delta q_t = q_t - q_{t-1}$ and task vector $\Delta\tau_t = \tau_t - \tau_{t-1}$ compared to the previous time step. These values are concatenated in the observation vector:

$$o_t = \Delta q_t \oplus \Delta\tau_t \tag{10.1}$$

The observation vector is served as input to a neural network:

$$\phi_t = f_\theta(o_t) \tag{10.2}$$

where $f_\theta$ is the neural network, $\theta$ are the network parameters and $\phi_t$ is the output, at time $t$. The output $\phi$ has the same size as the number of parameters in a C-LSTM layer:

$$\vec{\theta}^d_{xi}(t) \oplus \vec{\theta}^d_{hi}(t) \oplus \vec{\theta}^d_{xf}(t) \oplus \vec{\theta}^d_{hf}(t) \oplus \vec{\theta}^d_{xo}(t) \oplus \vec{\theta}^d_{ho}(t) = f_\theta(o_t) \tag{10.3}$$

where $\vec{\theta}$ denotes that the matrix $\theta$ is flattened into a vector, and $\oplus$ denotes vector concatenation. Each parameter is now dependent on the observation $o_t$, which is denoted as: $\theta^d_{xi}(o_t)$. Together we write the *Fast Weight C-LSTM equations*:

**Fast Weight C-LSTM equations:**

$$i^d_t = \sigma(x^d_t * \theta^d_{xi}(o_t) + h^d_{t\text{-}1} * \theta^d_{hi}(o_t) + \theta^d_{i_{bias}}) \tag{10.4}$$
$$f^d_t = \sigma(x^d_t * \theta^d_{xf}(o_t) + h^d_{t\text{-}1} * \theta^d_{hf}(o_t) + \theta^d_{f_{bias}}) \tag{10.5}$$
$$\tilde{c}^d_t = \tanh(x^d_t * \theta^d_{x\tilde{c}}(o_t) + h^d_{t\text{-}1} * \theta^d_{h\tilde{c}}(o_t) + \theta^d_{\tilde{c}_{\text{bias}}}) \tag{10.6}$$
$$c^d_t = \tilde{c}^d_t \odot i^d_t + c^d_{t\text{-}1} \odot f^d_t \tag{10.7}$$
$$o^d_t = \sigma(x^d_t * \theta^d_{xo}(o_t) + h^d_{t\text{-}1} * \theta^d_{ho}(o_t) + \theta^d_{o_{bias}}) \tag{10.8}$$
$$h^d_t = o^d_t \odot \tanh(c^d_t) \tag{10.9}$$
$$h = \sum_{d \in \mathscr{D}} h^d \tag{10.10}$$

which are the same equations as in Section 9.2, except for the change in weight terms, which are now dependent on $o_t$ through a neural network $f_\theta$: $\theta^d_\bullet(o_t)$. The resulting architecture is shown in Figure 10.1. Instead of six C-LSTM layers, as in the regular PyraMiD-LSTM, the Fast-Weight PyraMiD-LSTM uses only *one* Fast Weight C-LSTM layer to prevent information flowing backward in time.

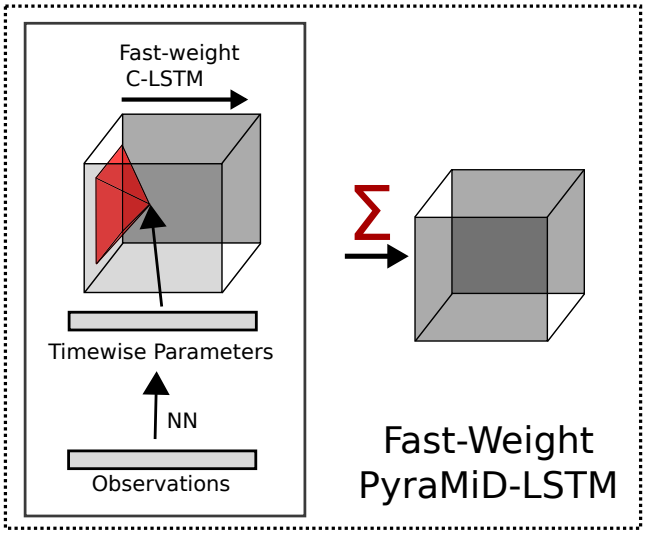



**Figure 10.1. Fast-Weight PyraMiD-LSTM** The Fast-Weight PyraMiD-LSTM architecture is shown. One Fast-Weight C-LSTM block, whose weights are created by a neural network with actions as input, predicts future images given the previous images and actions.

## 10.4 Exploration

The NGC algorithm is used to move the robot and build a dataset of the resulting image stream from the cameras. An exploration policy is needed to steer NGC that explores the task-space. A possible policy is to choose a random direction at every time step and move in that direction. However this would result in very jittery and unnatural movements; we want to have random behaviour that moves more smoothly.

A better alternative is to choose random directions for the *acceleration* of the control variable and integrate the result to get a control direction. Our approach uses a normally distributed acceleration:

$$\ddot{x} = \mathcal{N}(0, \sigma \cdot \mathbb{I}) \quad \underbrace{-\lambda\, \dot{x}}_{damping\ term} \tag{10.11}$$

$$\tag{10.12}$$

where $\ddot{x}$ is the acceleration of $x$, $\sigma$ controls the speed of exploration, and $\lambda$ is a damping value.

If we use Euler integration to update the model we get:

$$\dot{x}_t = \dot{x}_{t-\triangle t} + \mathcal{N}(0, \sigma \cdot \mathbb{I}) \cdot \triangle t - \lambda\, \dot{x}_t\, \triangle t \tag{10.13}$$

$$x_t = x_{t-\triangle t} + \dot{x}_{t-\triangle t} \cdot \triangle t \tag{10.14}$$

where $\triangle t$ is a step size. The resulting system builds up momentum resulting in directed movements in the space it's exploring. The same approach has been used to simulate the learning of place-cells in mice; relying on the efficient exploration of a space by this method [Franzius et al., 2007].

Without the damping term, the velocity is expected to become increasingly bigger over time as it integrates a normally distributed value $\dot{x}_t \sim \mathcal{N}(0, \sqrt{(t)}\sigma \cdot \mathbb{I}))$. This is obviously not desired and thus, the damping term is added.

**Kinematic Cost Function** We will use a simple hand movement as our control function. The cost function comprises:

**Home Pose Cost** A *home pose cost* biases the search algorithm to a natural pose.

**Hand Orientation Cost** The orientation of the right and left hand are kept straight, such that the fingers point forward and the hand palm is kept vertical.

**Right Hand Cost** The position of the right hand is kept fixed with a position cost, such that is does not interfere with the movements of the left hand.

**Looking Cost** The left and right eye are pointed toward the left hand by adding a *pointing cost*.

The task function is formed by using the position of the left hand:

$$g(q) = v_{\textit{left hand}} \tag{10.15}$$

This task function is used in the *control cost* of NGC (see Chapter 6). The robot is then controlled by changing the goal position $\tau^*$ of the task function according to the exploration policy described above. Care is taken to keep the control within certain bounds of the task-space that is reachable by the robot.

## 10.5 Experiment

The robot was controlled according to the exploration policy for approximately $30$ minutes, taking one frame per second. At every frame, the left image is stored, together with the task vector and joint parameter vector. Together they are stored in a database consisting of $1733$ images. The resulting database is separated in a training and testing dataset; the first $90\%$ of the images form the *training* set and the remaining $10\%$ form the *test* set. The images are converted to grey-scale images and are scaled down to

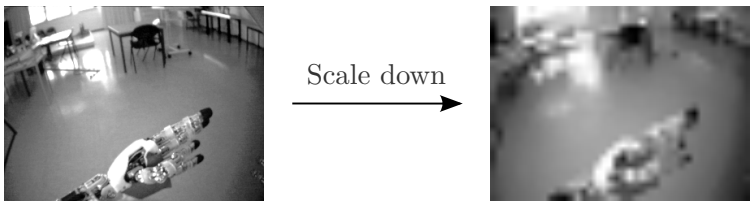



**Figure 10.2. Preparation of Data** The images in the dataset are scaled down to a resolution of $40 \times 30$ pixels to reduce the amount of information and keep training times manageable.

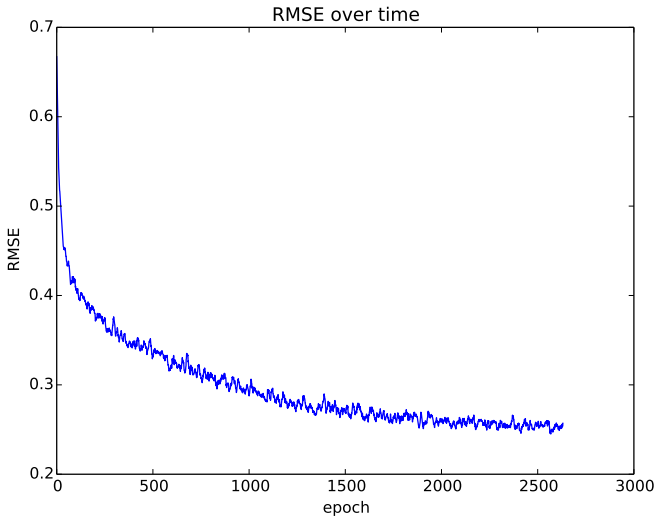



**Figure 10.3. Training of Fast-Weight PyraMiD-LSTM** The RMSE of the prediction is shown over training epochs. The RMSE showed here is smoothed over 10 epochs. Every epoch involves one 20 frames of recorded video. We can see the error slowly descends over time.

$40 \times 30$ pixels, as seen in Figure 10.2. This is done to keep training times low and reduce the difficulty of the problem.

The Fast-Weight PyraMiD-LSTM uses the following parameters: first, a fully connected layer expands the image into $32$ channels, after which an $\tanh$ operation is performed. This is fed to two consecutive Fast-Weight PyraMiD-LSTMs with $16$ and $32$ hidden units respectively. The resulting activations are expanded into 64 hidden activations using another fully connected layer, and after another $\tanh$ operation it is projected down to $1$ channel, to form the prediction. The resulting architecture has *451584* parameters. The network is trained on a state-of-the-art NVIDIA Titan X GPU, using the CUDNN library [Chetlur et al., 2014] to maximally use the $6,1$ GFLOPS available.

These parameters are not directly optimised, but are instead defined by the output of a fast-weight neural network. This neural network takes the $41+3 = 44$ dimensional observation vector as input and feeds it through a $64$, $32$, and $16$ hidden unit layer in that order. The final output layer projects towards a layer with $451584$ values; the number of weights in the PyraMiD-LSTM network. The network is trained by gradient descent, running from the prediction images, all the way down to the fast-weight neural network. It is trained using RMSPROP with momentum (see Section 7.4.2), with learning rates $\lambda_{max} = .01$, $\lambda_{min} = .00001$ and $halftime = 400$.

At every epoch, a random time-slice of 21 consecutive images is taken, of which the first $20$ form the input to the network and the last $20$ the target. The observation vector is given as input to the fast-weight NN, resulting in weights for the Fast-Weight PyraMiD-LSTM, and predictions are generated, and the backpropagation and learning update are performed. The drop in training error during training is shown in Figure 10.3. The training error drops gradually as expected over about $2500$ epochs.

**Comparison** The performance of the Fast-Weight PyraMiD-LSTM is compared to two other systems:

**Regular PyraMiD-LSTM without observations** A regular PyraMiD-LSTM network with the same parameters and one-directional C-LSTM layer, just like the Fast-Weight PyraMiD-LSTM, but instead the parameters are trained directly and not generated by a neural network. Since the weights are not generated, this network does not receive any information from the observations. It serves to show how difficult this problem is without observation information.

**Regular PyraMiD-LSTM with observations** The same network as above, but with the observation vectors as input. The observations are added using extra channels in the image; instead of a one-channel grey-scale image, the images are 1+44 channel images where the 44 extra channels encode the observations.

The regular PyraMiD-LSTM without observation information is expected to perform badly since it does not know which way the robot moves. The PyraMiD-LSTM with observations should perform better by using this information.

For the training of both systems, we use the same learning parameters as for the Fast-Weight PyraMiD-LSTM.

**Evaluation** For evaluation, all possible test-sequences of $31$ images are taken from the test-set. The first $10$ images are fed as input to the network to give it some starting information. Then the $11$th image is predicted and added as the $11$th input image.

Now there are 11 input images, and the 12th image is predicted. This image is added as the 12th input, and the process continues until a full prediction is formed.

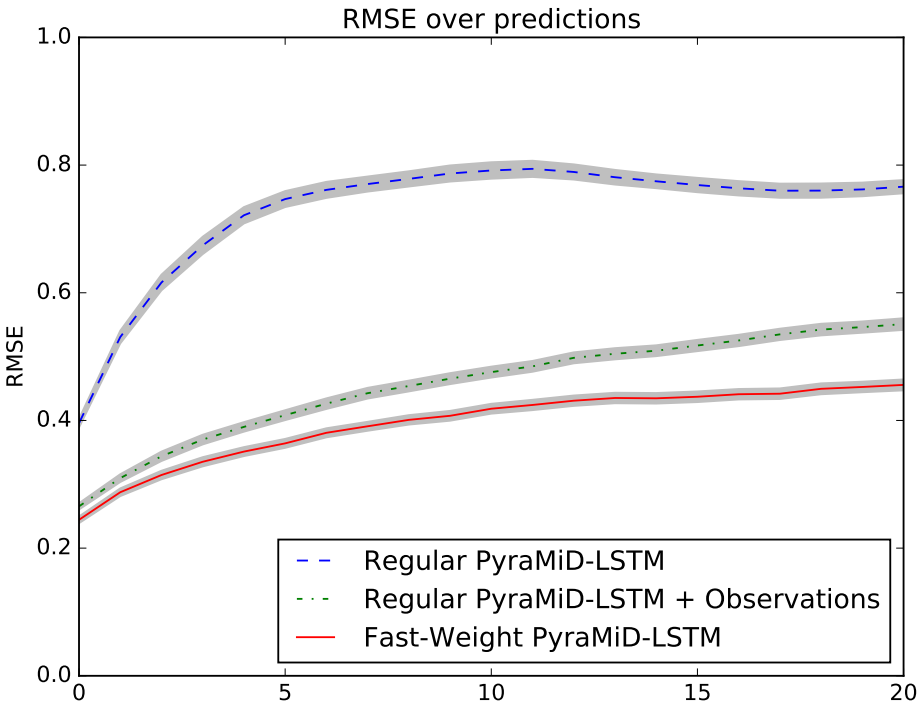



**Figure 10.4. Prediction RMSE of Fast-Weight PyraMiD-LSTM** The RMSE of the predictions of a Fast-Weight PyraMiD-LSTM and a regular PyraMiD-LSTM with and without observation information is shown. The *mean* and *estimated standard deviation of the mean* RMSE over sequences in the test-set are shown for the prediction of 1 to 21 time steps into the future. In general, the RMSE goes up as prediction time goes up. The Fast-Weight PyraMiD-LSTM clearly outperforms the regular PyraMiD-LSTMs. It is obvious that the networks perform better predictions when they can use the observations. It is unclear why the dynamics of the regular PyraMiD-LSTM are such that the error decreases slightly after 10 steps.

The resulting images are compared to the target and the RMSE is calculated. For every number of steps into the future, the *mean* and *standard deviation of the mean* are estimated. The results are shown in Figure 10.4. The RMSE goes up as the number of steps in the future increases, as expected. The Fast-Weight PyraMiD-LSTM clearly outperforms the regular PyraMiD-LSTM with and without observations.

Qualitative prediction results are shown in Figure 10.5. For four different time sequences, the predictions are shown. It should be noted that predicting several

images into the future is a very difficult task, especially given the number of degrees of freedom at work.

As can be seen, the Fast-Weight PyraMiD-LSTM can model perspective shifts caused by head movements, and local changes caused by hand movements. However, the predictions get blurry further into the future, especially in details of the hand. The regular PyraMiD-LSTM completely loses track of the future and just predicts a blurry image. The PyraMiD-LSTM with observations performs much better and shows the capability to adjust for robot actions, although on average more detail is lost than with the Fast-Weight PyraMiD-LSTM.

It should be noted that the weight-generating network of the Fast-Weight PyraMiD-LSTM has more parameters than the regular PyraMiD-LSTM. However, the evaluation time of this network is negligible compared to the complex convolutions of the PyraMiD-LSTM, taking up only $0.1\%$ of the time on average, and thus the overall evaluation time is practically the same. Thus the Fast-Weight PyraMiD-LSTM has a better performance while having similar computational complexity.

## 10.6 Concluding Remarks

In this chapter we have shown how our NGC kinematic control algorithm can work together with the Fast-Weight PyraMiD-LSTM to learn non-trivial models of the environment. NGC provided an easy way to set up the experiment to explore a task-space. The Fast-Weight PyraMiD-LSTM was subsequently trained on the images from the cameras, given the state of the robot given by its parameter vector and task vector.

The Fast-Weight PyraMiD-LSTM is capable of learning non-trivial predictions including changes in perspective and movements of the hand. The quality of the predictions deteriorated as the predictions went further into the future, but still showed resemblance with the actual images.

The predictions are done on scaled-down, low-resolution images to keep the computational burden low. Scaling up the approach to higher-resolution images is left for future work, and would require advances in the amount of memory available.

The main result of this chapter is to show how easily an experiment is set up that learns non-trivial visual processing in a real robotic setting. This is made possible by advancements made in this thesis, which maximally use the computational power available to them.

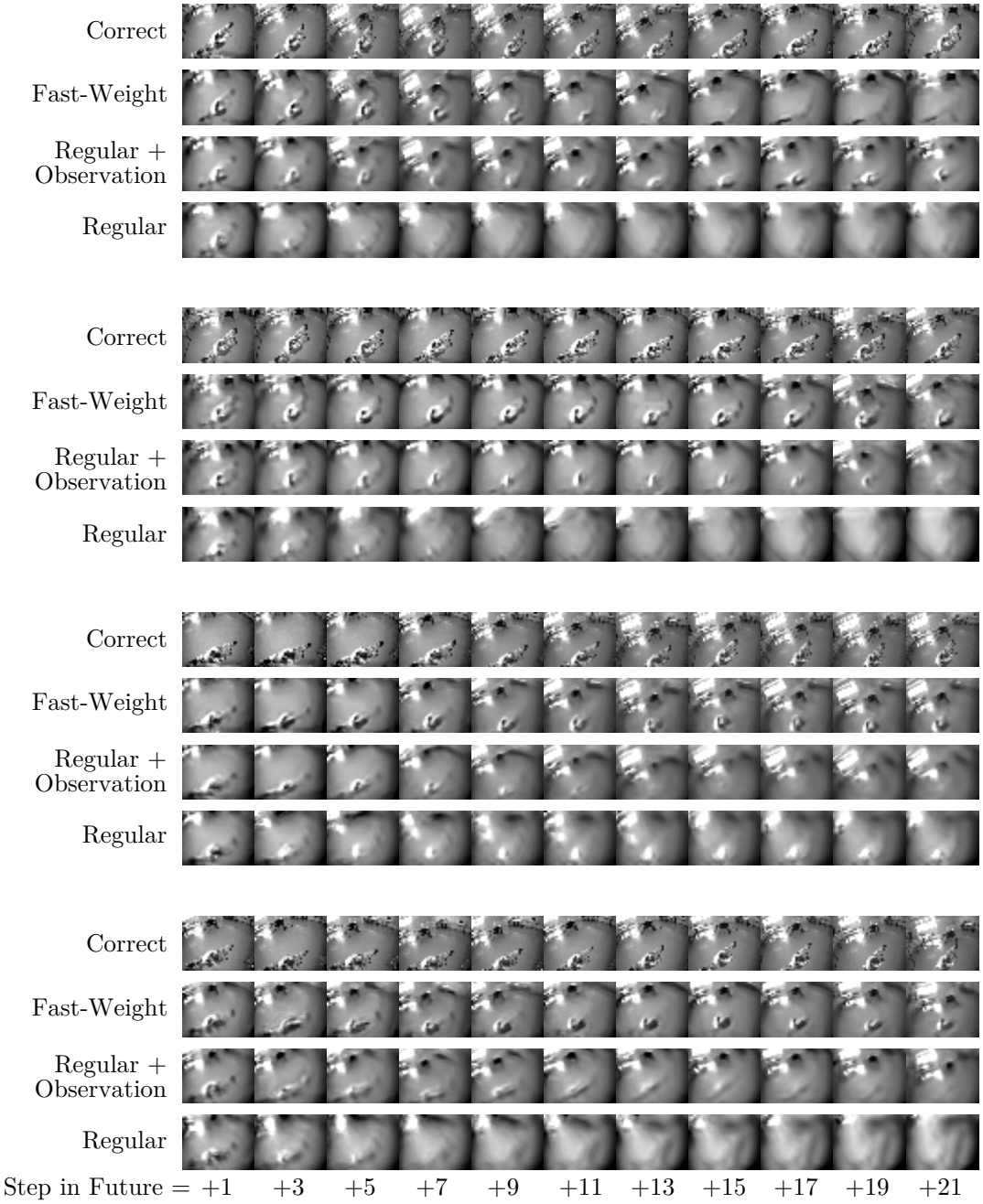



**Figure 10.5. Qualitative Prediction of Fast-Weight PyraMiD-LSTM** Qualitative results of a Fast-Weight PyraMiD-LSTM and a regular PyraMiD-LSTM with and without observations are shown for four sequences. It should be mentioned that this is a very difficult task; a head movement can change the perspective of the whole image while hand movements result in local changes. To learn to accommodate these both possibilities is non-trivial. The PyraMiD-LSTMs are given 10 actual frames, and from there predict from 1 up to 21 frames into the future; only the odd time steps are shown. The Fast-Weight PyraMiD-LSTM and PyraMiD-LSTM with observations are responsive to changes in view-angle and position of the hand, although the hand quickly becomes blurry. The regular PyraMiD-LSTM without observations hopelessly loses track of the future, since it can not incorporate action information. The regular PyraMiD-LSTM with observations performs decent predictions, but on average loses more detail than the Fast-Weight PyraMiD-LSTM.

# Chapter 11

# Conclusion

We have asked ourselves the main research question: *What innovations are needed to harness the vast amount of computational power to advance humanoid robotics?* In this light, we developed new control algorithms throughout Chapter 2 to Chapter 6 and developed novel visual processing strategies throughout Chapter 7 to Chapter 9. In Chapter 10 we showed how the methods from both of these directions can be integrated into a single experiment. In this final chapter, we will summarise the results and contributions of this thesis and relate them to the research questions asked in Section 1.4.

## 11.1 Main Results and Contributions

We explored the main research question into two parts: one focusing on robotic control, and the other on visual processing. The first part of the thesis explored Research Question 1: *What innovations are needed for humanoid robotic control algorithms to harness the vast amount of computational power, and what are the benefits?*
Our main results were:

- Sampling-based algorithms are powerful optimisation algorithms that have many benefits in humanoid kinematic control. Although they require a lot of computational power, their flexibility in optimising non-differentiable functions has great applications for inverse kinematics. This was shown with the development of Natural Gradient Inverse Kinematics (NGIK) (Chapter 3), by applying the powerful Natural Evolution Strategies algorithms (Section 3.2) to the inverse kinematics problem. NGIK has been shown successful in applications on the humanoid robot and was published in 2013 [**M. F. Stollenga** et al., 2013b].

- The computations by NGIK can be put to better use by reusing previous solutions. NGIK is thus used in Chapter 4 to build graphs of solutions that cover the desired task-space. The task-space forms the degrees of freedom defined by a freely definable function. The resulting graphs are called Task-Relevant Roadmaps (TRM) that can be built offline on a fast computer and reused later for planning. The many applications of this method are shown in the corresponding publication [**M. F. Stollenga** et al., 2013b] and a corresponding video [**M. F. Stollenga** et al., 2013a] which won the AAAI Student Video Award 2013. TRMs have also found applications in other work [Kompella et al., 2015].

- The building of TRMs is done offline because it takes a substantial amount of time, even on a fast computer. To fully use the available computational power, we need to use all available cores. Thus, we developed a parallel TRM builder (Chapter 5). Using a repel cost, several NGIK instances communicate and coordinate the search through the search space without interfering with each other. This resulted in substantial speed-ups and was published in 2014 [**M. F. Stollenga** et al., 2014].

- Instead of running NGIK until a solution is found, the internal optimisation step within NGIK can be adjusted to directly form control commands. In this way, an algorithm that can be run fast enough to perform kinematic control on the fly emerges, called Natural Gradient Control (NGC) (Chapter 6). Care must be taken to keep the covariance matrix wide enough to pick up changes in the cost function, but flexible enough that it can adjust to the shape of the cost landscape. The resulting algorithm was published in 2015 [**M. F. Stollenga** et al., 2015].

On the vision side, we asked: *What innovations are needed for computer vision algorithms to harness the vast amount of computational power, and what are the benefits?* In investigating this question we came to these main results:

- With increased computational power, the incorporation of feedback into CNNs becomes practical. Although the dynamical feedback process is very complex, we found that it can be optimised using the NES sampling-based optimisation algorithm which we also used for kinematic control. This is shown with the development of *dasNet*, in joint work with Jonathan Masci, which had state-of-the-art performance at the time of publication [**M. F. Stollenga* and J. Masci*** et al., 2014]). When looking into the activations of *dasNet*, we see how the trained policy enhances and suppresses filters to correct an otherwise wrong classification.

- The Long Short-Term Memory unit (LSTM) can be used as a gating mechanism to control the visual processing of complex 3D volumetric data. This is shown with the development of PyraMiD-LSTM, in joint work with Wonmin Byeon, which is inspired by MD-LSTM but uses a novel connection topology to make large scale visual processing possible. Six convolutional LSTM layers (C-LSTM) efficiently calculate pyramidal contexts for every voxel in each direction in a volume. The six resulting contexts are then combined to perform a classification. By pushing the computational and memory limits of a high-end GPU, state-of-the-art performance was achieved on a difficult brain segmentation dataset. This work was published in 2015 [**M. F. Stollenga* and W. Byeon*** et al., 2015].

Finally, we asked: *How can the algorithms emerging from Part I and Part II be integrated into an actual application on a humanoid robot?* The integration of both parts led to the following result:

- The PyraMiD-LSTM can be used to build a visual predictive model to *predict future images* given chosen actions. By developing a special Fast-Weight C-LSTM layer, whose parameters are not optimised directly but *generated* by a neural network, visual predictions can be conditioned on actions. These actions are formed from control commands from an explorative NGC procedure. The resulting integrated system learned to predict what the robot would *see* given its actions, many time steps into the future.

To summarise, we developed novel algorithms that harness the available computational power in ways that were not possible before. This resulted in a wide array of algorithms that advance the field of humanoid robotics and hopefully show how the ever-increasing computational power allows us to rethink algorithms in a new light.

## 11.2 Future Work

The work presented in this thesis can be improved and extended in many ways. We give an indication of the most promising directions for future work:

**Dynamic Control** Our work in controlling the robot focused solely on kinematic control. The work can be extended to dynamic control if a detailed *physical* model of the full robot is built. Interestingly, such a model could be learned from experience by an algorithm. To apply the control of NGC to such dynamic control, the algorithm needs to be sped up even more. With the introduction of faster simulations, parallel processing, and GPU acceleration this seems a promising possibility.

**Cost Function Inference** NGIK can optimise a wide range of cost functions. However, finding the right cost functions and balance them is still a manual process. Many approaches in robotics can learn from demonstration [Atkeson et al., 1997; Ijspeert et al., 2002] to avoid this manual construction. This approach could also be applied to NGIK; given a set of demonstrations on the robot, the corresponding parameters vectors could be stored. Several positive examples $q_i^+$ and negative examples $q_i^-$ could be demonstrated. Then a cost function $c(q)$ that is minimised on the positive examples $\min_c c(q^+)$, but has a certain higher cost on the negatives $c(q^-) > \tau$ given a threshold $\tau$, could be searched automatically. If $c$ is optimised under these conditions, a large array of cost functions could be created automatically simply by demonstrating movements on the robot.

**Advanced dasNet Policies** The policy used in *dasNet* was essentially a one-layer perceptron. Given that the policy is responsible for steering a complex attention mechanism, it is desirable that is can perform more complex computations. Exploring more interesting policy implementations, such as deep NNs, is a clear future research direction.

**4D PyraMiD-LSTM** The PyraMiD-LSTM was mainly designed for 3D volumetric data, but the equations are general and can be applied to any number of dimensions. An interesting application could be the extension to four-dimensional volumes-over-time data, such as those produced by functional MRI, where volumetric images of oxygen usage in the brain are recorded *over time*. Such applications could provide insights into the workings of the brain. However, the current 3D PyraMiD-LSTM already pushes the limits on the newest GPU cards, especially in memory. Thus, advances in hardware or optimisations of software are needed to make this application possible.

# Publications during PhD

**M. F. Stollenga**, J. Schmidhuber, and F. Gomez (2014). "Rapid Humanoid Motion Learning through Coordinated, Parallel Evolution". In: *Simulation of Adaptive Behavior*.

**M. F. Stollenga**, L. Pape, M. Frank, J. Leitner, A. Förster, and J. Schmidhuber (2013a). *Task Relevant Roadmaps*. Ed. by L. Pape. `http://www.youtube.com/watch?v=N6x2e1Zf_yg`. AAAI Student Video Award.

— (2013b). "Task-Relevant Roadmaps: A Framework for Humanoid Motion Planning". In: *IEEE International Conference on Intelligent Robots and Systems (IROS)*.

**M. F. Stollenga**, A. J. Lockett, and J. Schmidhuber (2015). "The Natural Gradient as a Control Signal for a Humanoid Robot". In: *IEEE RAS Humanoids Conference*.

**M. F. Stollenga* and J. Masci***, F. Gomez, and J. Schmidhuber (2014). "Deep networks with internal selective attention through feedback connections". In: *Advances in Neural Information Processing Systems*, pp. 3545–3553.

**M. F. Stollenga* and W. Byeon***, M. Liwicki, and J. Schmidhuber (2015). "Parallel Multi-Dimensional LSTM, With Application to Fast Biomedical Volumetric Image Segmentation". In: *Advances in Neural Information Processing Systems (NIPS)*.

Kompella, V. R., **Stollenga, M. F.**, M. D. Luciw, and J. Schmidhuber (2014). "Explore to see, learn to perceive, get the actions for free: SKILLABILITY". In: *Neural Networks (IJCNN), 2014 International Joint Conference on*. IEEE, pp. 2705–2712.

Kompella, V. R., **M. F. Stollenga**, M. Luciw, and J. Schmidhuber (2015). "Continual curiosity-driven skill acquisition from high-dimensional video inputs for humanoid robots". In: *Artificial Intelligence*.

Frank, M., J. Leitner, **M. F. Stollenga**, G. Kaufmann, S. Harding, A. Förster, and J. Schmidhuber (2012). "The modular behavioral environment for humanoids

and other robots (MoBeE)". In: *9th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*.

Kompella, V. R., M. Luciw, **M. F. Stollenga**, and J. Schmidhuber (2016). "Optimal Curiosity-Driven Modular Incremental Slow Feature Analysis". In: *Neural Computation (to appear)*.

Kompella, V. R., M. D. Luciw, **M. F. Stollenga**, L. Pape, and J. Schmidhuber (2012). "Autonomous learning of abstractions using Curiosity-Driven Modular Incremental Slow Feature Analysis". In: *ICDL-EPIROB*, pp. 1–8.

Srivastava, R. K., B. R. Steunebrink, **M. F. Stollenga**, and J. Schmidhuber (2012). "Continually Adding Self-Invented Problems to the Repertoire: First Experiments with PowerPlay". In: *Proceedings of the 2012 IEEE Conference on Development and Learning and Epigenetic Robotics IEEE-ICDL-EPIROB*. IEEE.

Wiering, M. A., M. H. V. der Ree, M. J. Embrechts, **M. F. Stollenga**, A. Meijster, A. Nolte, and L. R. B. Schomaker (2013). "The Neural Support Vector Machine". In: *Benelux Conference on Artificial Intelligence (BNAIC)*.

# Bibliography


Amari, S. ichi (1987). "Differential geometrical theory of statistics". In: *IMS Monograph vol. 10, Differential Geometry in Statistical Inference*, pp. 20–94.

Amari, S.-I. (1998). "Natural gradient works efficiently in learning". In: *Neural Computation* 10.2, pp. 251–276.

Ammar, G. and C. Martin (1986). "The geometry of matrix eigenvalue methods". In: *Acta Applicandae Mathematica* 5.3, pp. 239–278.

Angeles, J. (1985). "On the numerical solution of the inverse kinematic problem". In: *The International Journal of Robotics Research* 4.2, pp. 21–37.

Atkeson, C. G., A. W. Moore, and S. Schaal (1997). "Locally weighted learning for control". In: *Artificial Intelligence Review* 11.1-5, pp. 75–113.

Baerlocher, P. and R. Boulic (2004). "An inverse kinematics architecture enforcing an arbitrary number of strict priority levels". In: *The visual computer* 20.6, pp. 402–417.

Belliveau, J., D. Kennedy, R. McKinstry, B. Buchbinder, R. Weisskoff, M. Cohen, J. Vevea, T. Brady, and B. Rosen (1991). "Functional mapping of the human visual cortex by magnetic resonance imaging". In: *Science* 254.5032, pp. 716–719.

Bellman, R. (1957). *Dynamic Programming*. 1st. Princeton, NJ, USA: Princeton University Press.

**M. F. Stollenga** (2011). "Using Guided Autoencoders on Face Recognition". In: *Master's thesis*. University of Groningen.

**M. F. Stollenga**, J. Schmidhuber, and F. Gomez (2014). "Rapid Humanoid Motion Learning through Coordinated, Parallel Evolution". In: *Simulation of Adaptive Behavior*.

**M. F. Stollenga**, L. Pape, M. Frank, J. Leitner, A. Förster, and J. Schmidhuber (2013a). *Task Relevant Roadmaps*. Ed. by L. Pape. `http://www.youtube.com/watch?v=N6x2e1Zf_yg`. AAAI Student Video Award.

**M. F. Stollenga**, L. Pape, M. Frank, J. Leitner, A. Förster, and J. Schmidhuber (2013b). "Task-Relevant Roadmaps: A Framework for Humanoid Motion Planning". In: *IEEE International Conference on Intelligent Robots and Systems (IROS)*.

**M. F. Stollenga**, A. J. Lockett, and J. Schmidhuber (2015). "The Natural Gradient as a Control Signal for a Humanoid Robot". In: *IEEE RAS Humanoids Conference*.

**M. F. Stollenga* and J. Masci***, F. Gomez, and J. Schmidhuber (2014). "Deep networks with internal selective attention through feedback connections". In: *Advances in Neural Information Processing Systems*, pp. 3545–3553.

**M. F. Stollenga* and W. Byeon***, M. Liwicki, and J. Schmidhuber (2015). "Parallel Multi-Dimensional LSTM, With Application to Fast Biomedical Volumetric Image Segmentation". In: *Advances in Neural Information Processing Systems (NIPS)*.

Branson, S., C. Wah, F. Schroff, B. Babenko, P. Welinder, P. Perona, and S. Belongie (2010). "Visual recognition with humans in the loop". In: *Computer Vision–ECCV 2010*. Springer, pp. 438–451.

Bryson, Jr., A. E. and W. F. Denham (1961). *A steepest-ascent method for solving optimum programming problems*. Tech. rep. BR-1303. Raytheon Company, Missle and Space Division.

Byeon, W., T. M. Breuel, F. Raue, and M. Liwicki (2015). "Scene labeling with lstm recurrent neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3547–3555.

Byeon, W., M. Liwicki, and T. M. Breuel (2014). "Texture classification using 2d lstm networks". In: *2014 22nd International Conference on Pattern Recognition (ICPR)*. IEEE, pp. 1144–1149.

Campbell, L. L. (1985). "The relation between information theory and the differential geometry approach to statistics". In: *Information Sciences* 35.3, pp. 199–210.

Cardona, A., S. Saalfeld, S. Preibisch, B. Schmid, A. Cheng, J. Pulokas, P. Tomancak, and V. Hartenstein (2010). "An integrated micro-and macroarchitectural analysis of the Drosophila brain by computer-assisted serial section electron microscopy". In: *PLoS biology* 8.10, e1000502.

Chang, P. (1987). "A closed-form solution for inverse kinematics of robot manipulators with redundancy". In: *IEEE Journal on Robotics and Automation*.

Chetlur, S., C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer (2014). "cuDNN: Efficient Primitives for Deep Learning". In: *CoRR* abs/1410.0759.

Choset, H., K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun (2005). *Principles of robot motion: theory, algorithms, and implementations*. MIT press.

Church, A. (1936). "An unsolvable problem of elementary number theory". In: *American Journal of Mathematics* 58, pp. 345–363.

Ciresan, D. C., A. Giusti, L. M. Gambardella, and J. Schmidhuber (2012a). "Deep Neural Networks Segment Neuronal Membranes in Electron Microscopy Images". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 2852–2860.

Ciresan, D. C., U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber (2011). "Flexible, High Performance Convolutional Neural Networks for Image Classification". In: *Proceedings-International Joint Conference on Artificial Intelligence*, pp. 1237–1242.

Ciresan, D. C., A. Giusti, L. M. Gambardella, and J. Schmidhuber (2013). "Mitosis Detection in Breast Cancer Histology Images with Deep Neural Networks". In: *MICCAI*. Vol. 2, pp. 411–418.

Ciresan, D. C., U. Meier, and J. Schmidhuber (2012b). "Multi-Column Deep Neural Networks for Image Classification". In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. Long preprint arXiv:1202.2745v1 [cs.CV].

Courty, N. and E. Arnaud (2008). "Inverse kinematics using sequential Monte Carlo methods". In: *Articulated Motion and Deformable Objects*. Springer, pp. 1–10.

Dahl, G. E., T. N. Sainath, and G. E. Hinton (2013). "Improving deep neural networks for LVCSR using rectified linear units and dropout". In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, pp. 8609–8613.

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1, pp. 269–271.

Dreyfus, S. E. (1973). "The Computational Solution of Optimal Control Problems with Time Lag". In: *IEEE Transactions on Automatic Control* 18(4), pp. 383–385.

Dutra, M. S., I. L. Salcedo, and L. M. P. Diaz (2008). "New technique for inverse kinematics problems using simulated annealing". In: *Int. Conf. on Engineering Optimization*, pp. 01–05.

Edelman, A., T. A. Arias, and S. T. Smith (1998). "The geometry of algorithms with orthogonality constraints". In: *SIAM journal on Matrix Analysis and Applications* 20.2, pp. 303–353.

Fan, Y., Y. Qian, F.-L. Xie, and F. K. Soong (2014). "TTS synthesis with bidirectional LSTM based recurrent neural networks." In: *Interspeech*, pp. 1964–1968.

Farabet, C., C. Couprie, L. Najman, and Y. LeCun (2013). "Learning hierarchical features for scene labeling". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 35.8, pp. 1915–1929.

Frank, M., J. Leitner, **M. F. Stollenga**, G. Kaufmann, S. Harding, A. Förster, and J. Schmidhuber (2012). "The modular behavioral environment for humanoids and other robots (MoBeE)". In: *9th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*.

Franzius, M., H. Sprekeler, and L. Wiskott (2007). "Slowness and sparseness lead to place, head-direction, and spatial-view cells". In: *PLoS Comput Biol* 3.8, e166.

Fukushima, K. (2013). "Artificial vision by multi-layered neural networks: Neocognitron and its advances". In: *Neural Networks* 37, pp. 103–119.

— (1980). "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological cybernetics* 36.4, pp. 193–202.

— (1979). "Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position- Neocognitron". In: *ELECTRON. & COMMUN. JAPAN* 62.10, pp. 11–18.

Gabor, D. (1946). "Theory of communication. Part 1: The analysis of information". In: *Electrical Engineers-Part III: Journal of the Institution of Radio and Communication Engineering* 93.26, pp. 429–441.

Gauss, C. F. (1809). *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*.

Gers, F. A., J. Schmidhuber, and F. Cummins (1999). "Learning to Forget: Continual Prediction with LSTM". In: *Proc. ICANN'99, Int. Conf. on Artificial Neural Networks*. Edinburgh, Scotland: IEE, London, pp. 850–855.

Glasmachers, T., T. Schaul, S. Yi, D. Wierstra, and J. Schmidhuber (2010a). "Exponential natural evolution strategies". In: *12th annual conference on Genetic and Evolutionary Computation*. ACM, pp. 393–400.

— (2010b). "Exponential natural evolution strategies". In: *GEC*. ACM, pp. 393–400.

Gödel, K. (1931). "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". In: *Monatshefte für Mathematik und Physik* 38, pp. 173–198.

Goldenberg, A. A., B. Benhabib, and R. G. Fenton (1985). "A complete generalized solution to the inverse kinematics of robots". In: *Robotics and Automation, IEEE Journal of* 1.1, pp. 14–20.

Goodfellow, I. J., D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio (2013). "Maxout networks". In: *Proceedings of The 30th International Conference on Machine Learning*.

Gordon, N. J., D. J. Salmond, and A. F. Smith (1993). "Novel approach to nonlinear/non-Gaussian Bayesian state estimation". In: *Radar and Signal Processing, IEE Proceedings F*. Vol. 140. 2. IET, pp. 107–113.

Graça Marcos, M. da, J. T. Machado, and T. P. Azevedo-Perdicoúlis (2009). "Trajectory planning of redundant manipulators using genetic algorithms". In: *Communications in Nonlinear Science and Numerical Simulation* 14.7, pp. 2858–2869.

Graves, A. and J. Schmidhuber (2009). "Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks". In: *Advances in Neural Information Processing Systems 21*. Cambridge, MA: MIT Press, pp. 545–552.

Graves, A., M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber (2009). "A Novel Connectionist System for Improved Unconstrained Handwriting Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.5.

Graves, A. and N. Jaitly (2014). "Towards End-To-End Speech Recognition with Recurrent Neural Networks". In: *Proc. 31st International Conference on Machine Learning (ICML)*, pp. 1764–1772.

Graves, A., S. Fernandez, and J. Schmidhuber (2007). "Advances in Neural Network Architectures-Multi-dimensional Recurrent Neural Networks". In: *Lecture Notes in Computer Science* 4668, pp. 549–558.

Gross, C. G. (2002). "Genealogy of the grandmother cell". In: *The Neuroscientist* 8.5, pp. 512–518.

Hadamard, J. (1908). *Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées*. Mémoires présentés par divers savants à l'Académie des sciences de l'Institut de France: Éxtrait. Imprimerie nationale.

Hansen, N., S. D. Müller, and P. Koumoutsakos (2003). "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES)". In: *Evolutionary computation* 11.1, pp. 1–18.

Hart, P. E., N. J. Nilsson, and B. Raphael (1968). “A formal basis for the heuristic determination of minimum cost paths”. In: *Systems Science and Cybernetics, IEEE Transactions on* 4.2, pp. 100–107.

Hecker, C., B. Raabe, R. W. Enslow, J. DeWeese, J. Maynard, and K. van Prooijen (2008). “Real-time motion retargeting to highly varied user-created morphologies”. In: *ACM Transactions on Graphics (TOG)*. Vol. 27. 3. ACM, p. 27.

Helmke, U., K. Hüper, P. Y. Lee, and J. Moore (2007). “Essential matrix estimation using Gauss-Newton iterations on a manifold”. In: *International Journal of Computer Vision* 74.2, pp. 117–136.

Hemami, A (1987). “A more general closed-form solution to the inverse kinematics of mechanical arms”. In: *Advanced robotics* 2.4, pp. 315–325.

Hochreiter, S. (1991). *Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München*. Advisor: J. Schmidhuber.

Hochreiter, S. and J. Schmidhuber (1995). *Long Short-Term Memory*. Tech. rep. FKI-207-95. Fakultät für Informatik, Technische Universität München.

— (1997). “Long Short-Term Memory”. In: *Neural Computation* 9.8. Based on TR FKI-207-95, TUM (1995), pp. 1735–1780.

Hsu, D., J. C. Latombe, and H. Kurniawati (2006). “On the probabilistic foundations of probabilistic roadmap planning”. In: *The Int. Journal of Robotics Research* 25.7, pp. 627–643.

Hsu, D., J.-C. Latombe, and R. Motwani (1997). “Path planning in expansive configuration spaces”. In: *IEEE Int. Conf. on Robotics and Automation (ICRA)*. Vol. 3. IEEE, pp. 2719–2726.

Hubel, D. H. and T. N. Wiesel (1962). “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex”. In: *The Journal of physiology* 160.1, pp. 106–154.

— (1959). “Receptive fields of single neurones in the cat’s striate cortex”. In: *The Journal of physiology* 148.3, pp. 574–591.

*iCub Mechanical Drawings (Subversion Repository)*. `https://svn.robotology.eu/repos/iCubHardware-pub/trunk/mechanics/`.

Ijspeert, A. J., J. Nakanishi, and S. Schaal (2002). “Learning attractor landscapes for learning motor primitives”. In: *Advances in neural information processing systems*, pp. 1523–1530.

Ivakhnenko, A. (1968). “The group method of data handling-a rival of the method of stochastic approximation”. In: *Soviet Automatic Control* 13.3, pp. 43–55.

Ivakhnenko, A. G. and V. G. Lapa (1965). *Cybernetic predicting devices*. CCM Information Corporation.

— (1967). *Cybernetics and forecasting techniques*. Vol. 8. American Elsevier Pub. Co.

Jarrett, K., K. Kavukcuoglu, M. Ranzato, and Y. LeCun (2009). “What is the best multi-stage architecture for object recognition?” In: *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE, pp. 2146–2153.

Kauschke, M. (1996). “Closed form solutions applied to redundant serial link manipulators”. In: *Mathematics and Computers in Simulation* 41.5, pp. 509–516.

Kelley, H. J. (1960). “Gradient Theory of Optimal Flight Paths”. In: *ARS Journal* 30.10, pp. 947–954.

Kitagawa, G. (1996). “Monte Carlo filter and smoother for non-Gaussian nonlinear state space models”. In: *Journal of computational and graphical statistics* 5.1, pp. 1–25.

Kompella, V. R., M. Luciw, **M. F. Stollenga**, and J. Schmidhuber (2016). “Optimal Curiosity-Driven Modular Incremental Slow Feature Analysis”. In: *Neural Computation (to appear)*.

Kompella, V. R., **Stollenga, M. F.**, M. D. Luciw, and J. Schmidhuber (2014). “Explore to see, learn to perceive, get the actions for free: SKILLABILITY”. In: *Neural Networks (IJCNN), 2014 International Joint Conference on*. IEEE, pp. 2705–2712.

Kompella, V. R., M. D. Luciw, **M. F. Stollenga**, L. Pape, and J. Schmidhuber (2012). “Autonomous learning of abstractions using Curiosity-Driven Modular Incremental Slow Feature Analysis”. In: *ICDL-EPIROB*, pp. 1–8.

Kompella, V. R., **M. F. Stollenga**, M. Luciw, and J. Schmidhuber (2015). “Continual curiosity-driven skill acquisition from high-dimensional video inputs for humanoid robots”. In: *Artificial Intelligence*.

Krizhevsky, A. (2009). “Learning multiple layers of features from tiny images”. MA thesis. Computer Science Department, University of Toronto.

Krizhevsky, A., I Sutskever, and G. E Hinton (2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: *NIPS*, p. 4.

Kullback, S. and R. A. Leibler (1951). "On information and sufficiency". In: *The annals of mathematical statistics* 22.1, pp. 79–86.

Lab, D.-H. *First results for swift on a 64-core amd opteron 6376*. `https://community.dur.ac.uk/pedro.gonnet/?p=269`.

Lagarias, J. C., J. A. Reeds, M. H. Wright, and P. E. Wright (1998). "Convergence Properties of the Nelder–Mead Simplex Method in Low Dimensions". In: *SIAM Journal on Optimization* 9.1, pp. 112–147.

LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press.

LeCun, Y. (1988). "A theoretical framework for Back-Propagation". In: *Proceedings of the 1988 Connectionist Models Summer School*. Ed. by D. Touretzky, G. Hinton, and T. Sejnowski. CMU, Pittsburgh, Pa: Morgan Kaufmann, pp. 21–28.

— (1985). "Une procédure d'apprentissage pour réseau a seuil asymmetrique (a Learning Scheme for Asymmetric Threshold Networks)". In: *Proceedings of Cognitiva 85*. Paris, France, pp. 599–604.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel (1989). "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4, pp. 541–551.

— (1990). "Handwritten digit recognition with a back-propagation network". In: *Advances in Neural Information Processing Systems 2 (NIPS*89)*. Ed. by D. Touretzky. Denver, CO: Morgan Kaufman.

Legendre, A. M. (1805). *Nouvelles méthodes pour la détermination des orbites des cometes*. F. Didot.

Leibniz, G. W. (1684). "Nova methodus pro maximis et minimis, itemque tangentibus, quae nec fractas, nec irrationales quantitates moratur, et singulare pro illis calculi genus". In: *Acta Eruditorum*, pp. 467–473.

*libANN: CA library for approximate nearest neighbor searching*. `http://www.cs.umd.edu/~mount/ANN/`.

*libSolid: Collision detection library*. `http://www.dtecta.com/`.

Lin, M., Q. Chen, and S. Yan (2013). "Network In Network". In: *CoRR* abs/1312.4400.

Linnainmaa, S. (1970). "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors". MA thesis. Univ. Helsinki.

Liu, T., C. Jones, M. Seyedhosseini, and T. Tasdizen (2014). "A modular hierarchical approach to 3D electron microscopy image segmentation". In: *Journal of Neu-*

*roscience Methods* 226.0, pp. 88–102. issn: 0165-0270. doi: http://dx.doi.org/10.1016/j.jneumeth.2014.01.022.

Lundström, E. and L. Eldén (2002). "Adaptive eigenvalue computations using Newton's method on the Grassmann manifold". In: *SIAM journal on matrix analysis and applications* 23.3, pp. 819–839.

Markoff, J. (2012). "How many computers to identify a cat? 16,000". In: *New York Times*.

McCulloch, W. S. and W. Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.

Mendrik, A., K. Vincken, H. Kuijf, G. Biessels, and M. Viergever (2015). *MRBrainS Challenge: Online Evaluation Framework for Brain Image Segmentation in 3T MRI Scans, http://mrbrains13.isi.uu.nl*.

Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller (1953). "Equation of state calculations by fast computing machines". In: *The journal of chemical physics* 21.6, pp. 1087–1092.

Metta, G., G. Sandini, D. Vernon, L. Natale, and F. Nori (2008). "The iCub humanoid robot: an open platform for research in embodied cognition". In: *Proceedings of the 8th workshop on performance metrics for intelligent systems*. ACM, pp. 50–56.

Metta, G., P. Fitzpatrick, and L. Natale (2006). "YARP: yet another robot platform". In: *International Journal on Advanced Robotics Systems* 3.1, pp. 43–48.

Nair, V. and G. E. Hinton (2010). "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814.

*Nonlinear Equations with Finite-Difference Jacobian - MATLAB*. http://www.mathworks.com/help/optim/ug/nonlinear-equations-with-finite-difference-jacobian.html.

Parker, D. B. (1985). *Learning-Logic*. Tech. rep. TR-47. Center for Comp. Research in Economics and Management Sci., MIT.

Paul, R. P. and B. Shimano (1979). "Kinematic control equations for simple manipulators". In: *Conference on Decision and Control*. IEEE, pp. 1398–1406.

Peters, J. and S. Schaal (2006). "Policy gradient methods for robotics". In: *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*. IEEE, pp. 2219–2225.

Peters, J. and S. Schaal (2008). “Reinforcement learning of motor skills with policy gradients”. In: *Neural Network* 21.4, pp. 682–697.

Pizer, S. M., E. P. Amburn, J. D. Austin, R. Cromartie, A. Geselowitz, T. Greer, B. T. H. Romeny, and J. B. Zimmerman (1987). “Adaptive Histogram Equalization and Its Variations”. In: *Comput. Vision Graph. Image Process.* 39.3, pp. 355–368. issn: 0734-189X. doi: 10.1016/S0734-189X(87)80186-X.

Post, E. L. (1936). “Finite Combinatory Processes-Formulation 1”. In: *The Journal of Symbolic Logic* 1.3, pp. 103–105.

Quiroga, R. Q., L. Reddy, G. Kreiman, C. Koch, and I. Fried (2005). “Invariant visual representation by single neurons in the human brain”. In: *Nature* 435.7045, pp. 1102–1107.

Reinders, J. (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O’Reilly Media, Inc."

Robinson, A. J. and F. Fallside (1987). *The Utility Driven Dynamic Error Propagation Network*. Tech. rep. CUED/F-INFENG/TR.1. Cambridge University Engineering Department.

Rosenblatt, F. (1961). *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. DTIC Document.

— (1958). “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6, p. 386.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). “Learning Internal Representations by Error Propagation”. In: *Parallel Distributed Processing*. Ed. by D. E. Rumelhart and J. L. McClelland. Vol. 1. MIT Press, pp. 318–362.

Sak, H., A. Senior, and F. Beaufays (2014a). “Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition”. In: *CoRR* abs/1402.1128.

— (2014b). “Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling”. In: *Proc. Interspeech*.

Samir, C., P.-A. Absil, A. Srivastava, and E. Klassen (2012). “A gradient-descent method for curve fitting on Riemannian manifolds”. In: *Foundations of Computational Mathematics* 12.1, pp. 49–73.

Schaul, T., T. Glasmachers, and J. Schmidhuber (2011a). “High Dimensions and Heavy Tails for Natural Evolution Strategies”. In: *Genetic and Evolutionary Computation Conference (GECCO)*. Dublin, Ireland.

Schaul, T., T. Glasmachers, and J. Schmidhuber (2011b). "High dimensions and heavy tails for natural evolution strategies". In: *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, pp. 845–852.

Schmidhuber, J. (1992). "A Fixed Size Storage $O(n^3)$ Time Complexity Learning Algorithm for Fully Recurrent Continually Running Networks". In: *Neural Computation* 4.2, pp. 243–248.

— (1991). *Learning to Control Fast-Weight Memories: An Alternative to Recurrent Nets*. Tech. rep. FKI-147-91. Institut für Informatik, Technische Universität München.

Schmidhuber, J. (2015). "Deep learning in neural networks: An overview". In: *Neural Networks* 61, pp. 85–117.

Segmentation of Neuronal Structures in EM Stacks Challenge (2012). *IEEE International Symposium on Biomedical Imaging (ISBI), http://tinyurl.com/d2fgh7g*.

Sermanet, P., K. Kavukcuoglu, S. Chintala, and Y. LeCun (2013). "Pedestrian Detection with Unsupervised Multi-Stage Feature Learning". In: *CVPR*. IEEE.

Srivastava, R. K., B. R. Steunebrink, **M. F. Stollenga**, and J. Schmidhuber (2012). "Continually Adding Self-Invented Problems to the Repertoire: First Experiments with PowerPlay". In: *Proceedings of the 2012 IEEE Conference on Development and Learning and Epigenetic Robotics IEEE-ICDL-EPIROB*. IEEE.

Sutskever, I., O. Vinyals, and Q. V. Le (2014). *Sequence to Sequence Learning with Neural Networks*. Tech. rep. arXiv:1409.3215 [cs.CL]. NIPS'2014. Google.

Tieleman, T. and G. Hinton (2012). "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural Networks for Machine Learning* 4.

Tsagarakis, N. G., G. Metta, G. Sandini, D. Vernon, R. Beira, F. Becchi, L. Righetti, J. Santos-Victor, A. J. Ijspeert, M. C. Carrozza, et al. (2007). "iCub: the design and realization of an open humanoid platform for cognitive and neuroscience research". In: *Advanced Robotics* 21.10, pp. 1151–1175.

Tsai, L.-W. and A. P. Morgan (1985). "Solving the kinematics of the most general six- and five-degree-of-freedom manipulators by continuation methods". In: *Journal of Mechanical Design* 107.2, pp. 189–200.

Turing, A. M. (1936). "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363, p. 5.

Wan, L., M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus (2013). "Regularization of neural networks using dropconnect". In: *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1058–1066.

Wang, L., Y. Gao, F. Shi, G. Li, J. H. Gilmore, W. Lin, and D. Shen (2015). "LINKS: Learning-based multi-source IntegratioN frameworK for Segmentation of infant brain images". In: *NeuroImage* 108.0, pp. 160–172. issn: 1053-8119. doi: `http://dx.doi.org/10.1016/j.neuroimage.2014.12.042`.

Welinder, P., S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, and P. Perona (2010). *Caltech-UCSD Birds 200*. Tech. rep. CNS-TR-2010-001. California Institute of Technology.

Werbos, P. J. (1981). "Applications of Advances in Nonlinear Sensitivity Analysis". In: *Proceedings of the 10th IFIP Conference, 31.8 - 4.9, NYC*, pp. 762–770.

— (1974). "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". PhD thesis. Harvard University.

— (1988). "Generalization of Backpropagation with Application to a Recurrent Gas Market Model". In: *Neural Networks* 1.

Wiering, M. A., M. H. V. der Ree, M. J. Embrechts, **M. F. Stollenga**, A. Meijster, A. Nolte, and L. R. B. Schomaker (2013). "The Neural Support Vector Machine". In: *Benelux Conference on Artificial Intelligence (BNAIC)*.

Wierstra, D., T. Schaul, J. Peters, and J. Schmidhuber (2008). "Natural evolution strategies". In: *IEEE Congress on Evolutionary Computation*. IEEE, pp. 3381–3387.

Wierstra, D., T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber (2014). "Natural evolution strategies". In: *The Journal of Machine Learning Research* 15.1, pp. 949–980.

Williams, R. J. (1989). *Complexity of exact gradient computation algorithms for recurrent neural networks*. Tech. rep. Technical Report NU-CCS-89-27. Boston: Northeastern University, College of Computer Science.

Wolovich, W. A. and H Elliott (1984). "A computational technique for inverse kinematics". In: *The 23rd IEEE Conference on Decision and Control*. Vol. 23. IEEE, pp. 1359–1363.

Zeiler, M. D. and R. Fergus (2013). "Stochastic pooling for regularization of deep convolutional neural networks". In: *arXiv preprint arXiv:1301.3557*.

— (2014). "Visualizing and understanding convolutional networks". In: *Computer Vision–ECCV*. Springer, pp. 818–833.

Zohdy, M., M. Fadali, and N. Loh (1989). “Robust control of robotic manipulators”. In: *American Control Conference*. IEEE, pp. 999–1004.

Zuse, K. (1967). “Rechnender Raum”. In: *Elektronische Datenverarbeitung* 8, pp. 336–344.

Zuse, K. (1936). “Verfahren zur selbsttätigen Durchführung von Rechnungen mit Hilfe von Rechenmaschinen”. In: *Z23139 IX/42m (11.4. 1936)*.