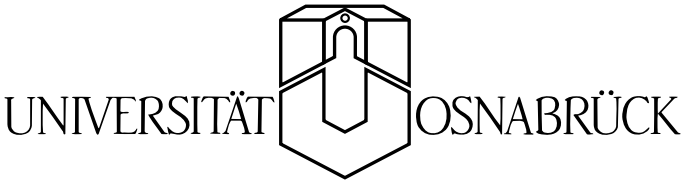


Institut für Informatik
Arbeitsgruppe Wissensbasierte Systeme

Institut für Kognitionswissenschaft
Arbeitsgruppe Neurokybernetik

Evolving Complex Neuro-Controllers with Interactively Constrained Neuro-Evolution

Dissertation

zur Erlangung des akademischen Grades

Dr. rer. nat.

Fachbereich Informatik
der
Universität Osnabrück

von

Christian Wilhelm Rempis

2012

Betreuer: Prof. Dr. Frank Pasemann

Eingereicht im April 2012

Verteidigt am 9. Oktober 2012

Gutachter: Prof. Dr. Frank Pasemann
Institut für Kognitionswissenschaft, AG Neurokybernetik,
Universität Osnabrück, Germany

Prof. Dr. Joachim Hertzberg
Institut für Informatik, AG Wissensbasierte Systeme,
Universität Osnabrück, Germany

Prof. Dr. Ralf Der
Max Planck Institut für Mathematik in den Naturwissenschaften,
Leipzig, Germany

Preface

Parts of this dissertation have already been published. The corresponding sections are not explicitly marked as citations. Section 2.4 and 7 are partially identical with published work in Rempis and Pasemann (2012a). Parts of section 5.1 have been published in Rempis and Pasemann (2010). Also, parts of section 8 have been used in Rempis and Pasemann (2012b). Appendix D describes material published in Rempis et al. (2010). The corresponding content of these publications has been my own work.

Abstract

In the context of evolutionary robotics and neurorobotics, artificial neural networks, used as controllers for animats, are examined to identify principles of neuro-control, network organization, the interaction between body and control, and other likewise properties. Before such an examination can take place, suitable neuro-controllers have to be identified. A promising and widely used technique to search for such networks are evolutionary algorithms specifically adapted for neural networks. These allow the search for neuro-controllers with various network topologies directly on physically grounded (simulated) animats. This neuro-evolution approach works well for small neuro-controllers and has lead to interesting results. However, due to the exponentially increasing search space with respect to the number of involved neurons, this approach does not scale well with larger networks. This *scaling problem* makes it difficult to find non-trivial, larger networks, that show interesting properties. In the context of this thesis, networks of this class are called *mid-scale* networks, having between 50 and 500 neurons. Searching for networks of this class involves very large search spaces, including all possible synaptic connections between the neurons, the bias terms of the neurons and (optionally) parameters of the neuron model, such as the transfer function, activation function or parameters of learning rules. In this domain, most evolutionary algorithms are not able to find suitable, non-trivial neuro-controllers in feasible time.

To cope with this problem and to shift the frontier for evolvable network topologies a bit further, a novel evolutionary method has been developed in this thesis: the Interactively Constrained Neuro-Evolution method (ICONE).

A way to approach the problem of increasing search spaces is the introduction of measures that reduce and restrict the search space back to a feasible domain. With ICONE, this restriction is realized with a unified, extensible and highly adaptable concept: Instead of evolving networks freely, networks are evolved within specifically designed *constraint masks*, that define mandatory properties of the evolving networks. These constraint masks are defined primarily using so called *functional constraints*, that actively modify a neural network to enforce the adherence of all required limitations and assumptions. Consequently, independently of the mutations taking place during evolution, the constraint masks repair and readjust the networks so that constraint violations are not able to evolve. Such functional constraints can be very specific and can enforce various network properties, such as symmetries, structure reuse, connectivity patterns, connectivity density heuristics, synaptic pathways, local processing assemblies, and much more. Constraint masks therefore describe a narrow, user defined subset of the parameter space – based on domain knowledge and user experience – that focuses the search on a smaller search space leading to a higher success rate for the evolution.

Due to the involved domain knowledge, such evolutions are strongly biased towards specific classes of networks, because only networks within the defined search space can evolve. This, surely, can also be actively used to lead the evolution towards specific solution approaches, allowing the experimenter not only to search for *any* upcoming solution, but also to confirm assumptions about possible solutions. This makes it easier to investigate specific neuro-control principles, because the experimenter can systematically search

for networks implementing the desired principles, simply by using suitable constraints to enforce them.

Constraint masks in ICONE are built up by functional constraints working on so called *neuro-modules*. These modules are used to structure the networks, to define the scope for constraints and to simplify the reuse of (evolved) neural structures. The concept of functional, constrained neuro-modules allows a simple and flexible way to construct constraint masks and to inherit constraints when neuro-modules are reused or shared.

A final cornerstone of the ICONE method is the interactive control of the evolution process, that allows the adaptation of the evolution parameters and the constraint masks to guide evolution towards promising domains and to counteract undesired developments. Due to the constraint masks, this interactive guidance is more effective than the adaptation of the evolution parameters alone, so that the identification of promising search space regions becomes easier.

This thesis describes the ICONE method in detail and shows several applications of the method and the involved features. The examples demonstrate that the method can be used effectively for problems in the domain of mid-scale networks. Hereby, as effects of the constraint masks and the herewith reduced complexity of the networks, the results are – despite their size – often easy to comprehend, well analyzable and easy to reuse. Another benefit of constraint masks is the ability to deliberately search for very specific network configurations, which allows the effective and systematic exploration of distinct variations for an evolution experiment, simply by changing the constraint masks over the course of multiple evolution runs.

The ICONE method therefore is a promising novel evolution method to tackle the problem of evolving mid-scale networks, pushing the frontier of evolvable networks a bit further. This allows for novel evolution experiments in the domain of neurorobotics and evolutionary robotics and may possibly lead to new insights into neuro-dynamical principles of animat control.

Acknowledgments

My first thanks go to my supervisor Prof. Dr. Frank Pasemann, who guided my work with inspiration, discussion and advise, giving me the freedom to explore various ideas without limitations. This especially includes his support to realize the method with an elaborate software framework despite the required time and effort. His concrete ideas and research questions give this thesis the practical relevance making it more than just another theoretical approach.

I also want to thank Prof. Dr. Joachim Hertzberg for valuable advice on the thesis and – together with Prof. Dr. Ralf Der – for being part of my examination committee.

Exceptionally, I thank my parents for their constant support on all levels, which gave and gives me a strong backing regardless of what I do. Not less than that, I thank my soulmate Leni Beck for her unlimited patience.

Many thanks also go to my colleagues and friends I had the pleasure to work with during the last 6 years, who made the work not only interesting, but also a lot of fun. For valuable discussions, feedback and inspirations, I want to thank especially Arndt von Twickel, Mario Negrello, Johannes Schumacher, Hermann Prüm, Verena Thomas, Ferry Bachmann, Manfred Hild, Nina Mühleis, Vishal Patel and Chris Reinke. In the same spirit I value the support of the guys from the Neurorobotics Laboratory of the Humboldt University Berlin, particularly Christian Thiele, Christian Benckendorff, Benjamin Werner, Thorsten Siedel and Matthias Kubisch. Special thanks also go to all contributors, users and testers of the NERD Toolkit software, including the students participating in the practice sessions for the lectures on evolutionary robotics and neuro-control. For the time-consuming corrections of my English writing skills, I particularly thank Tanya Beck, who saved the world from superfluous grammatical inventions.

This thesis was written during my work as a scientific assistant at the Neurocybernetics Laboratory of the Institute of Cognitive Science at the University of Osnabrück. My thanks go to all staff members of the institute that have supported my work, especially Beate Eibisch, and the IKW administrators Udo Wächter, Martin Schmidt and Michael Rausch for their support concerning the cluster integration of the NERD Toolkit.

Parts of this work have been funded by EU-Project Number ICT 214856 (ALEAR - Artificial Language Evolution on Autonomous Robots. http://www.alear.eu), which I acknowledge gratefully.

This thesis uses a latex template of Wolfgang Runte, which saved me a lot of trouble and effort.

For Leni

Contents

Chapter 1

Introduction

1.1 Motivation

The regulatory, adaptive and behavioral abilities of living organisms have always fascinated researchers all over the globe. With their capability of self-regulation and their reactive adaptability these systems maintain robust conditions to survive and reproduce, involving the acquisition and processing of nutrients, the maintenance of essential inner body states to provide a stable environment for all body processes (e.g. temperature, acid-base-state), the protection of the organism from harmful environmental conditions (e.g. predators, radiation, toxic chemicals) and the ability to reproduce, evolve and to adapt to their ecological niche. The interplay of regulation is so diverse that even today many questions regarding even simple organisms are not fully answered. The study of such regulatory systems, especially of their interaction with their environment through feedback loops, is called *cybernetics* (Ashby 1956; Wiener 1948). Among the living organisms, animals are particularly fascinating to study, because of their ability to strongly influence their environment through interactions, particularly using sophisticated limbs to move and to manipulate their environment. This also requires corresponding sophisticated sensors and proprioceptors, which together allow the complex behaviors that we can observe every day in animals (including ourselves). The basis of that control for the vast majority of animals has been discovered already in the early twentieth century: the nerve cells (neurons) and the nervous system (neural networks) (Kandel et al. 2000; López-Muñoz et al. 2006). Understanding the function and dynamical properties of the nervous systems of animals is considered an important step towards understanding the nervous system of humans and eventually also of human cognition with its creative, adaptive and constructive abilities. This makes the study of nervous systems not only interesting for biology, but also for many, probably far-reaching future technical applications.

The study of feedback-driven neural control is often referred to as *neuro-cybernetics*. As a sub-discipline of biocybernetics, researchers in this field try to understand the role, function and properties of neuro-control with respect to their application in technical systems. Research in this domain also includes work on interfaces between biological nervous systems and technical systems (e.g. neuroprosthetics), but for the context of this thesis, the

term neuro-cybernetics is used to refer primarily to the understanding of feedback-driven neuro-control by means of their organizational and dynamical principles.

A related discipline with overlapping research goals is neurorobotics. This sub-discipline of artificial intelligence focuses on the understanding and creation of intelligence as a result of a body-brain-environment interaction. Here, physically grounded robots are used to test biologically inspired theories on information processing and control, especially including theories on neuro-control. Experiments in this domain are often used to empirically prove that certain theories – derived from biological experiments – are also practically applicable and thus sound (albeit not necessarily correct).

Being highly interdisciplinary, the fields of (neuro-)cybernetics and neurorobotics involve many disciplines, such as biology, chemistry, physics, ethology, medicine, robotics, electronics, mechatronics and computer science. So, the problem of understanding neuro-control can be approached from many different perspectives. While some disciplines concentrate on understanding the details of the function of the many different neuron cell types and their signal transmission, other disciplines focus on the broader principles of control or the interplay of high-level behaviors. Others investigate how different configurations of the required physical *body* with its sensors, proprioceptors and actuators affect the organization of neural control, to investigate how much (neural) control actually is necessary for a behavior and how much of it originates from an elaborated body design. A research focus therefore can also be the understanding of the various feedback-loops between neuro-control (including feedback-loops within the network), the body (including its inner states) and the environment (including other organisms). Another interesting research topic is learning and memory in neural networks, that allows animals to adapt their behavior during their life-time. These examples of research questions should show the broadness and complexity of the field.

Some researchers approach the problems *top-down* (biology, ethology), starting with a working, observable behavior of a living animal and trying to decompose that behavior by analyzing the underlying biological nervous systems with increasing detailedness. Others investigate the questions *bottom-up* (robotics, computer science, physics) by trying to synthesize artificial, simulated (and simplified) nervous systems for artificial reactive agents, starting with low-level functions and extending them to more and more complex networks.

The bottom-up approach, involving artificial (physical or simulated) animals, is also called the animat approach (Meyer 1998; Meyer and Guillot 2008; Watts 1998; Wilson 1991) and is a sub-discipline of artificial intelligence and artificial life (Bedau 2003; Langton 1989). This approach is a valuable complement to the examination of living organisms. The artificial replication of nervous systems on animats allows a detailed examination of neural principles in working systems that can be modified and varied quite easily to systematically investigate the properties of control networks. The hereby required modifications and variations of the neural networks would not be possible with living animals. Also, the analysis of an artificial neural network is less difficult, because all inner states of the artificial neural network can be accessed at any time with absolute accuracy. In contrast, when examining living organisms, only partial, noisy data of the neural network activities can be obtained despite the usually high technical effort. Thus, artificial nervous systems are also especially suited to be analyzed and understood from the perspective of the dynamical

systems theory (Beer 1995, 2000; Cruse 2009; Pasemann 1998; Thelen and Smith 1994). This view recognizes animats as dynamical subsystems embedded in an all-embracing dynamical system (environment) and understands behavior and cognition as a result of the interaction of internal (brain) and external (body, environment) dynamics. Understanding the neuro-dynamics of artificial neural networks of animats in their environmental context (and not just as isolated system) is considered a promising approach to explain behavior and cognition.

This thesis focuses on a problem involved with this bottom-up approach: The synthesis of artificial neuro-controllers. As prerequisite for the analysis of neuro-controllers, one obviously requires a way to design such controllers for a particular animat. Replicating nervous systems of living beings by simply copying their network structure is – presently – not possible for many reasons, e.g. the complexity of the networks, the inability to derive the necessary synaptic weights, and the inability to accurately simulate all details of the necessary artificial body of the animal. Accordingly, networks have to be designed from scratch, naturally often guided and inspired by knowledge obtained from experiments with living organisms. Designing such controllers is difficult because of the parallel, distributed processing nature of the usually highly interconnected complex neural networks with their many parameters.

A quite successful approach to this problem are techniques from evolutionary robotics (ER) and artificial life (Bornhofen and Lattaud 2006; Floreano et al. 2008b; Harvey et al. 1997, 2005; Lungarella et al. 2003; Meyer et al. 1998; Nolfi and Floreano 2004; Pfeifer and Bongard 2006). Instead of manually constructing neuro-controllers for artificial systems, neuro-controllers are developed using evolutionary algorithms (see section 2.2). With this gradient-based class of search algorithms, neuro-controllers are developed by trial-and-error, inspired by principles of Darwinian evolution (Darwin 1859). This approach has been successfully used for many years (e.g. Bongard 2003; Capi and Doya 2005; Gomez and Schmidhuber 2005; Hornby et al. 2003; Hülse et al. 2004; Kodjabachian and Meyer 1998; Lipson et al. 2006; Meyer et al. 2003; Pasemann and Dieckmann 1997; Pasemann et al. 2003b; Rempis 2007; von Twickel and Pasemann 2007; Walker et al. 2003; Wischmann and Pasemann 2006). One problem with this technique is, however, that evolutionary algorithms do not scale well with the size and complexity of the neural networks. This is often referred to as the *scaling-problem* of neuro-evolution (Hornby et al. 2003; Maja J. Matarić 1996). As a result, most of the neuro-controllers in this domain are very small compared to the nervous systems of even simple animals. Using the evolutionary approach to evolve larger neuro-controllers for complex robots with many sensors and actuators often fails due to the immense involved search space. Therefore, the approach – although promising – could not provide the anticipated rich pool of complex neuro-controllers yet.

Therefore, researchers are trying to push the frontier of evolvable network complexities with new, specialized neuro-evolution techniques. A number of such algorithms have been proposed during the last 20 years (see section 2.4), speeding up the evolution of neuro-controllers and increasing the probability of successful experiments. However, the complexity of the evolved networks still remains comparably small.

With this thesis, a new evolution technique is introduced, that pushes the frontier of evolvable network complexities again a bit further, opening the research area for new in-

teresting experiments that require larger networks and may involve more complex animats with richer sets of sensors, proprioceptors and actuators. The expectation is that this method may help to gain new insights into neuro-cybernetic, neurorobotics and evolutionary robotics problems, that have not been explored so far.

1.2 Approach and Delimitations

Problems to Solve. Evolving larger neuro-controllers in the context of neuro-cybernetics and neurorobotics comes with a number of problems. First, as already mentioned, the search space for the evolutionary algorithms increases drastically with every new neuron in the network. At some point, the search space becomes too large to provide a reasonable chance to find a suitably working network at all. A reduction of the search space therefore seems mandatory to keep the search space in a feasible domain. An important observation that can be made when evolving larger networks is, that often large parts of the networks are of regular, repetitive nature, such as repeating structures, symmetries or the repeated utilization of specific organization principles. In cases where the existence of such properties is known or assumed, a free evolution without any restrictions unnecessarily increases the search space and reduces the probability of finding suitable controllers. A mechanism to induce such regularities to the evolving networks can significantly reduce the search space. Furthermore, it can be observed that similar neural structures are required over and over again in different (independent) experiments, for instance motor control structures, sensor processing, memory functions and pattern generators. Evolving such structures from scratch in each experiment again blows up the search space without leading to new insights regarding the actual, intrinsic problem. So it would be beneficial to provide such structures as building blocks during evolution to relieve evolution from reinventing already known structures. In the same spirit, it seems reasonable to divide networks into structures, which are the focus of interest of an evolution (*focus structures*), and those structures, that are – in principle – already known, but are still required for the fully working neuro-controller (*peripheral structures*). Being able to define such peripheral structures in advance relieves evolution from the time-consuming search for structures whose development does not contribute to answering the scientific questions underlying the experiment. In the context of neurorobotics research, these questions are usually very clear and detailed, so that it also would be beneficial to guide or to bias the evolution towards very specific solution approaches. Evolutionary algorithms have the tendency to solve problems with the most probable, often most simple solutions, that frequently are not the ones the experimenter is looking for. This becomes especially important, if controllers are evolved to replicate or realize a given concept, e.g. a control hypothesis derived from biological organisms. Here, some kind of guidance is indispensable.

The evolution method presented in this thesis addresses all mentioned problems to reduce the search space, to increase the success rate also for larger networks, and to allow the guidance of the evolution with domain-knowledge and interactive supervision.

Approach. The proposed method, called **I**nteractively **Co**nstrained **N**euro-**E**volution (ICONE), primarily focuses on strategies to reduce the search space and to influence the possible network structures in advance. This allows the induction of domain knowledge to the networks and a guidance of the evolution towards preselected subspaces of the overall search space. To achieve this, ICONE uses so called *functional constraints* to define a constraint mask on the networks, forcing all evolving networks to remain within the given limitations. Different from passive constraints, which are often used with evolutionary algorithms and are merely used to calculate a degree of constraint violation for a solution, functional constraints can actively modify a neural network to enforce the given limitations. With such constraints, it becomes quite easy to exploit regularities in the network (symmetries, structure repetitions, specific connectivity patterns) and to apply domain knowledge to the evolving networks. The functional constraints in ICONE are not limited to a fixed set, so that even complex, very specific constraint functions can be defined on demand.

Constraints are not applied globally to the entire network, but merely on freely selectable *groups* of neurons. This allows a simple selection of affected neurons and a flexible definition of the constraint mask by constraining only relevant neurons. A special case of neuron groups are the so called *neuro-modules*. Unlike ordinary groups, these neuro-modules do not allow intersections with other neuro-modules. However, they allow a hierarchical stacking (as submodules) and provide a neural interface. Only neurons that are part of this neural interface can have connections to neurons outside of the module. Thus, neuro-modules separate a network into hierarchical network areas with well-defined connectivity. Due to these properties, neuro-modules are suitable as neural building blocks, that can be used during evolution or during network preparations to extend a network by fully functional units. Since such modules are usually well-designed (section 5.1.2) and equipped with functional constraints, such modules extend the search space only marginally and avoid the reinvention of already known neural structures. Due to the constraints, these modules are still mutable to some extent, but their function is preserved by the constraints to avoid a destruction of the module's purpose through mutations. Modules are also the base of a new crossover operator that allows the exchange of neural structures between lineages (see section 3.3.3).

To guide an evolution experiment, the experimenter has to define one or more initial networks manually, hereby structuring the networks into neuro-modules, defining peripheral structures and attaching functional constraints. This process is called *constrained modularization* (section 5.1). Such an initial network describes a constraint mask, that defines the set of all network structures that are possible to evolve. For this, the experimenter obviously requires domain knowledge and experience. In addition, a graphical software tool is mandatory to define and test such networks efficiently. The initial networks – once defined – can be used to perform various variants of evolution experiments by slightly diversifying the evolution parameters or the constraint masks of the initial networks.

ICONE approves the conduction of interactive, iterative evolutions, i.e. step-wise evolution experiments under supervision. This allows the constant guidance and adaptation of the evolution process and the reaction to unforeseen problems during the course of an

experiment. Once a promising subspace of the search space is identified, unsupervised evolutions can further help to quickly exploit that search area and find variations to the solutions.

ICONE allows a wide range of constrained evolutions. Depending on the constraint masks, evolutions with ICONE can reach from unconstrained evolutions similar to many other evolution methods, over slightly constrained search spaces excluding only obviously faulty controllers or inducing simple heuristics, up to very strongly constrained search spaces used to only confirm very specific assumptions. In all cases, the approach allows a very detailed influence of the networks to be evolved and increases the chance of finding not only working controllers, but also varieties of very specific kinds of controllers. This makes the method valuable for the neurorobotics approach.

Delimitations. The proposed method does not claim to solve the scaling problem of neuro-evolution. However, the method opens the domain of neuro-evolution to new experiments that have been out of scope so far. The focus of the method, unlike many other evolution methods, has not been on performance, but instead on a good success rate, the induction of domain knowledge and the guidance of the neuro-evolution process. Hence, the focus is on its *practical* applicability in the domain of *mid-scale* (section 2.3) networks.

1.3 Contributions

ICONE Neuro-Evolution Method. The main contribution of this thesis is the general evolution method ICONE, that allows the application of modularization, functional constraints and other measures to enforce arbitrary network features to heterogeneous, recurrent neural networks. With this method new kinds of experiments can be designed that may lead to novel insights into the dynamical principles of neuro-control. This is achieved through four major effects:

(1) The search space of the experiments can – problem-specifically – be greatly reduced, which makes neuro-evolution applicable to more complex animats and allows the evolution of larger, non-trivial neuro-controllers.

(2) The outcome of the neuro-evolution can be biased by the experimenter towards very specific results. This allows the use of neuro-evolution to confirm or explore given (theoretical) approaches of the organization of neural networks.

(3) With step-wise variations of the constraint masks it becomes quite easy to systematically explore the search space to also find such neuro-controllers, that would be very unlikely to be discovered in a global, unconstrained search space.

(4) Principles of neural organization can easier be transferred to new animats, when these principles – once identified – are described as functional constraints. Such constraints can then be used for the constraint masks of the new animats, leading to evolved networks using the given principles.

(5) Evolved neural structures can be worked up as constrained neuro-modules. These neural building blocks can be collected in a module library, from which they can be shared

with other researchers and be used as primitive building blocks for the design of initial networks or for macro-mutations during neuro-evolution.

Put together, this method increases the control over the evolution and the related search space of neuro-controllers for the experimenter and simplifies the work with functional subnetworks.

NERD Toolkit. The ICONE method has been implemented as a reference framework for the so-called **N**eurodynamics and **E**volutionary **R**obotics **D**evelopment **Toolkit** (NERD Toolkit). This open-source software provides an extensible, well documented implementation of the ICONE method, an integrated physical general-purpose simulator, a sophisticated neural network editor and various analysis tools for neuro-controllers. Details on the framework can be found in appendix D.

Demonstrators. The method has been applied to various experiments. Two examples are described in detail: The evolution of a bipedal walking behavior for a humanoid robot (chapter 7) and the evolution of variations of locomotion behaviors with a multi-segmented closed-chain animat (chapter 8). The experiments serve as demonstrations of the method and give a first impression on how the method can be applied.

1.4 Thesis Overview

After motivating the work at hand, the remainder of this thesis is structured as follows.

Chapter 2 describes the major related topics that are required to set the main part of the thesis into context. The chapter starts by describing artificial neural networks as they are used to control animats, followed by a description of evolutionary algorithms in general and the related terminology. In the next section the scaling problem for neuro-evolution is addressed and the application domain of the ICONE method – namely *mid-scale* networks – is defined. The chapter is completed by a review of the state-of-the-art in neuro-evolution, describing the current major approaches and their ability to cope with the problem of large search spaces.

The next four chapters describe the ICONE method and its application.

Chapter 3 focuses on the main ICONE algorithm. Sections 3.1 to 3.3 give details on the requirements for the genome encoding, the actual evolution algorithm, the role of neuron groups and neuro-modules, a new modular crossover operator, functional constraints and additional measures to define a constraint mask for a neural network.

Chapter 4 gives detailed information on all currently implemented functional constraints and their effects during the mutation phase. This section is required for an understanding of the applications in chapters 7 and 8.

Chapter 5 focuses on the main procedures involved when evolving neuro-controllers with the ICONE method. This covers the modularization of initial networks (section 5.1), including the optimal refinement of neuro-modules for the structure reuse through a module library (sections 5.1.2 and 5.1.3) and the practice of interactive (section 5.2) and iterative (section 5.3) evolution.

Chapter 6 then concludes the method description by listing a number of useful extensions to the basic ICONE method. These extensions, among many others that can be realized for the method in future work, increase the expressiveness of the constraint masks and enhance the usability of evolved neuro-controllers.

Chapters 7 and 8 demonstrate the application of the ICONE method to several real-world problems. Two experiments are described in detail to demonstrate the experiment design and the evolution process. In the first experiment (section 7) a walking behavior is evolved for a physical humanoid robot with 42 motor neurons and 37 sensor neurons. The second experiment (section 8) demonstrates the systematic search for variations of the locomotion behavior of a multi-segmented closed-chain-animat with up to 30 motor and 120 sensor neurons.

Chapter 9 discusses the ICONE method and suggests corresponding future work.

Chapter 10 finally concludes the thesis and briefly evaluates the potential impact of the ICONE method on the field of neuro-evolution.

The **Appendix** finally gives an overview on the *symbols* used to visualize the network graphs (Appendix A), provides details on the evolution operators (Appendix B), *network tags* (Appendix C, see also section 3.2.4) and the reference implementation of the method, called the NERD Toolkit (Appendix D).

Chapter 2

Foundations

This chapter provides the basic foundations needed to understand and relate the neuro-evolution method described in the subsequent chapters. First, the essentials of artificial recurrent neural networks, as they are used to control artificial agents, are briefly reviewed. Then, the basics of the general evolutionary search algorithms are summarized, introducing the related terminology used in the remainder of the thesis. In the third section, the scaling problem of neuro-evolution is discussed and the term *mid-scale networks* as application domain for the proposed neuro-evolution method is defined. The final section of this chapter contains a review on the current state-of-the-art neuro-evolution techniques. This section also highlights the problems of current neuro-evolution approaches, so that the proposed neuro-evolution approach can be appraised in the appropriate context.

2.1 Artificial Neural Networks for Control

Artificial neural networks (ANN) are a widely used class of (computational) calculation models inspired by biological nerve cells. Since their first appearance in the early twentieth century (Hebb 1949; McCulloch and Pitts 1943; Rosenblatt 1958), many different neural network models have been proposed over the years that serve different purposes, such as simulating biological, spiking neurons as close as possible, to train networks as data classifiers, to learn the prediction of time-series or to control technical systems. Despite their differences, the core components of such neural networks are usually the same: The networks are composed of parallel processing nodes, the so called *neurons*, connected with directed links, called *synapses*. These two main components correlate to the biological neurons and their synaptic connections between their dendrites and axons.

The actual implementation and the properties of the neurons and synapses can be very different, depending on the detail level that has to be simulated. Additional network properties, for instance single spikes (Brette et al. 2007), membrane potentials, ion channels (Hines and Carnevale 1997), neural gas (Fritzke 1995; Martinetz and Schulten 1991)

or direct electric influences between neurons can be modeled. A complete and detailed overview on the differences of the many ANN models and the history of ANNs cannot be given in this section. The interested reader is referred to the vast literature on the topic (e.g. Livingstone 2008; Rojas 1996)).

For the domain of behavior control of animats, one aspect of the ANN models has to be pointed out. The more details a model considers, the more difficult and more time-consuming its calculation becomes. In many applications, especially for the control of physical technical systems, the response of a neural control network has to be very fast to allow an acceptably reactive control. Thus, the choice of a suitable simulation model is crucial for the application: It should reflect all neuro-dynamical features required for the task or desired to be examined, without being too complex and hence too computationally extensive for a given context.

For this reason, relatively simple models of ANNs are used in the domain of control. One advantage of such elementary ANN in the domain of control is that the networks consist of very basic building blocks (neurons and synapses) and therefore are simple to implement and fast to calculate, even on embedded systems. A second advantage is that many of these networks can be trained with numerous learning algorithms (e.g. Anthony (2009), Haykin (2008)), such as backpropagation (Werbos 1990) or evolutionary search algorithms (section 2.2 and 2.4), so that the networks (and therefore the control programs) can be automatically designed based on data samples of the control problem. But also from the perspective of the biological control theory such simple ANN models are interesting, because they still reflect many (dynamical) properties assumed to be responsible for the behavior of biological nervous systems (Beer 2000; von Twickel et al. 2011). This makes them interesting targets for the study of structure and organization of general neural control with the help of artificial systems, which are much easier to analyze and observe than biological systems (Beer 1995; Wischmann and Pasemann 2006).

Recurrent Additive Discrete-Time Neuron Model. In the context of this thesis, one such simple neuron model is used to control animats: the standard additive discrete-time ANN model. This model is described by two functions, the *activation function* a (equation 2.1) and the output function o (essentially defined by the *transfer function* τ) (equation 2.2).

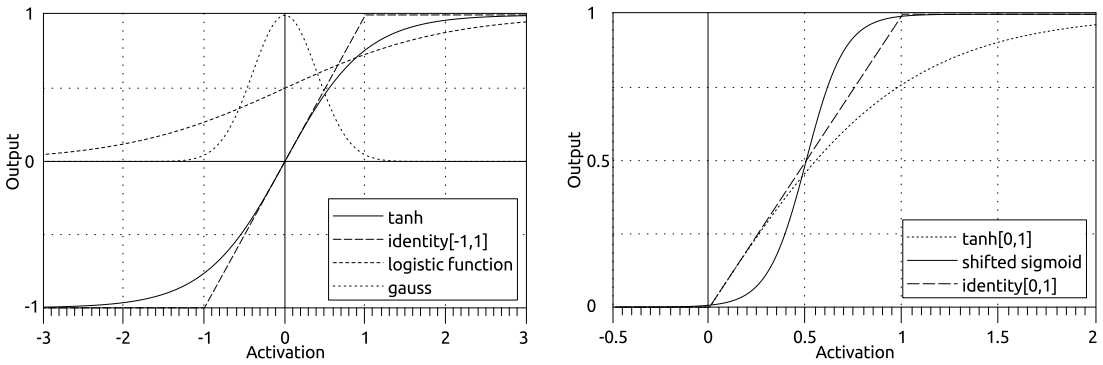
$$a_i(t+1) = \theta_i + \sum_{j=1}^{n} w_{ij}\, o_j(t)); \qquad i = 1, \ldots, n; \qquad t \in \{0, 1, 2, \ldots\}; \qquad o_i, a_i, \theta_i, w_{ij} \in \mathbf{R} \tag{2.1}$$

$$o_i(t) = \tau_i(a_i(t)); \qquad i = 1, \ldots, n; \qquad t \in \{0, 1, 2, \ldots\}; \qquad o_i, a_i \in \mathbf{R} \tag{2.2}$$

Here, o_i is the *output* of neuron i, a_i is the *activity* of neuron i, w_{ij} is the weight of the synaptic connection from neuron j to neuron i, and θ_i denotes its fixed bias term. The bias term can be interpreted as a fixed synapse coming from an always active neuron and provides a constant base excitation or inhibition for a neuron. The output o_i of a neuron i is given by applying the *transfer function* $\tau_i(a_i)$ of neuron i on its current activation a_i.

This neuron model is clearly not thought to be an exact model of biological nervous systems; the output of a model neuron may be interpreted as the mean activity (firing rate) of a biological neuron's axon for a certain time interval and the model synapses may be interpreted as the combined effect of all excitatory and inhibitory biological synapses that would exist between the axon of a source neuron and all dendrites of the target neuron. This simple recurrent model, however, already shows many interesting dynamical properties (Beer 2005; Hülse et al. 2007; Pasemann 1998, 2002) that allow insightful experiments on dynamical principles of reactive behavior control with artificial neural networks.

Transfer Function. The transfer function τ often is a bounded, nonlinear, differentiable function like the commonly used logistic function or hyperbolic tangent. Bounded transfer functions limit the neuron's output range, which is an important property of biological neurons. Nonlinear transfer functions can lead to much more interesting neuro-dynamics compared to the sole use of linear transfer functions. On the other hand, linear transfer functions can also be useful when nonlinear effects on a signal are not desired. The differentiability is a prerequisite for many learning methods, so it is required when such methods are applied. Apart from these features, each transfer function obviously also produces a different kind of output and thus defines a different class of dynamical systems with different network dynamics.



(a) (b)

Figure 2.1: Comparison of different transfer functions and their impact on the neuron output. Note the different axis ranges of (a) and (b).

The hyperbolic tangent, for instance, allows a positive and negative output and thus a *dynamic switching* of inhibition and excitation of a neuron's influence. The standard logistic function, in difference, has an output range of [0,1] and is in this respect biologically more similar to a single neuron, but its output behavior provides a quite high output for

low and even negative activations, which differs from biological neurons. Other transfer functions, such as the sigmoid functions shown in figure 2.1 may be biologically more plausible, but results with these functions are difficult to compare to the results using the main-stream transfer functions. Because of the advantages and potential capacities of the many transfer functions, the given neuron model here does not – different from most ANN implementations in ER – assume a single transfer function to be used for all neurons of the network. Instead the transfer function τ_i can be chosen separately for each neuron i, allowing heterogeneous networks that support – if desired – the combination of the strengths of different transfer functions. This widens the range of supported experiments by the interesting field of heterogeneous networks, with homogeneous networks still being supported as a special case.

Activation Function and Synapse Function. The activation function in equation 2.2 is used in all experiments of this thesis. However, the evolutionary method proposed in this thesis also supports the use of different activation functions and of heterogeneous networks, where not only the transfer function, but also the activation function can differ from neuron to neuron. Also, the weights of the synapses are not necessarily fixed (as in the equation), but may also be represented by a separate function σ_i for each synapse, here called the *synapse function*. This mix of neuron and synapse models in a single, heterogeneous network allows powerful networks combining the strengths of different models where they are most beneficial.

2.2 Evolutionary Algorithms

Evolutionary algorithms (EA) with their many variants – such as genetic algorithms (Goldberg 1989; Holland 1992), genetic programming (Koza 1992, 1994; Koza et al. 1999), evolutionary strategies (Back et al. 1991; Rechenberg 1994), evolutionary programming (Fogel and Fogel 1996; Fogel et al. 1966) and neuro-evolution (Schaffer et al. 1992; Yao 1992) – are search algorithms performing a gradient ascent search on populations of candidate solutions. All such algorithms have a common basic search strategy, analogue to Darwinian evolution (Darwin 1859):

EAs maintain a pool of candidate solutions to a given problem, each solution candidate representing a distinct point in the multi-dimensional search space. EAs iteratively modify this pool by replacing bad performing candidates by new ones, following a simple heuristic: Given the problem provides performance gradients relative to the search space, then good solution candidates are likely to be found close to even better candidates. Therefore, the replacement of the worst solution candidates of a population with slightly varied well performing candidates results in a local search around the points in search space that already have shown to have a good performance and that are likely to include also better candidates. EAs therefore follow the performance gradients by sampling promising subspaces of the search space, converging to (local) optima.

The general EAs iteratively run in five phases, called – inspired by natural evolution – evaluation, selection, reproduction, recombination and mutation. Following this analogy,

the candidate solutions are often called *individuals*, the evolving solutions a *population*, and a set of competing candidates during one iteration a *generation*.

In the *evaluation* phase, the individuals of an (initially often randomly created) generation are rated for their efficiency to solve the problem. This requires a performance measure, often called a *fitness function*, that is problem specific and actually encodes the problem to be solved.

In the *selection* phase, the candidates are chosen to be either replaced or to be used as basis to create new candidates. In the latter case, such selected candidates are often referred to as *parents*.

A number of different selection methods have been proposed for this process (Jong 2006), commonly involving selection probabilities based on the measured performance of an individual. Usually, the better an individual is, the more offspring is created based on that individual.

During the *reproduction* and *recombination* phases the new individuals are derived from their parents. For this, the genomes of the parents (i.e. their parameter representations) are duplicated and (optionally) recombined during the so called *crossover*, where parts of the parents' genomes are exchanged. The individuals resulting from a crossover therefore share parameters of both parents. This corresponds to the local search around promising points in the search space.

During the *mutation* phase, the parameters of each individual are randomly modified, moving the represented point in search space slightly in different dimensions. For evolutionary algorithms without a crossover operator – which is common in the field of neuro-evolution – this is the only measure to alter the genome.

These five phases are repeated until convergence, which corresponds either to a solution of the problem or to a local undesired optimum. Accordingly, EAs are not guaranteed to solve a given problem, but have still proven to be quite efficient for many problem domains.

EAs have been used for many different applications (for examples see the *Applications of Evolutionary Computing* series, e.g. Cagnoni et al. 2000; Giacobini et al. 2009), because their general algorithms makes them suitable for a wide variety of problems, such as engineering, art, program design, function approximation and agent control. The use of EA in most domains is simple, because the problem specific parts of the algorithm are limited to the genome representation (with appropriate mutation operators) and the fitness function.

Despite their many successful applications, EAs have several problems that limit their usability. First, the characteristics of the problem, especially the genome representation and the used fitness function, strongly influence the performance of the EAs. Obviously, the more parameters have to be optimized, the more time-consuming the search becomes. But also the structure of the so called *fitness landscape* (Langdon and Poli 2002, chapter 2), i.e. the characteristics of the fitness space, has a major impact (Peliti 1996), as for all gradient based search methods. If the fitness landscape does not provide sufficient gradients, then the search performance can drop down to that of a random search. If the landscape provides too many local optima, the search easily gets stuck prematurely. Also, the mutation settings have to be suitably low to ensure that the mutated candidates still represent points in search space close to that of their parents. But on the other hand,

they should be large enough to allow an escape from local optima. Also, the crossover operators should only produce individuals that are still representing candidates close to their parents, which is often difficult to achieve (see section 3.3.3).

2.3 Mid-Scale Networks and the Scaling Problem

During the last decades, many interesting neuro-evolution experiments have been published in the evolutionary robotics (ER) community (for examples see the *From Animals to Animats* series, e.g. Doncieux et al. 2010; Meyer and Wilson 1990). The first experiments started out successfully with simple animats and very small fixed-topology networks. With the computers getting faster, more sophisticated experiments have been performed, but the reported sizes of the evolved neural networks (with and without topology evolution) are usually still small. To give a rough order of magnitude, non-developmental neuro-evolution (NE) algorithms work with less than about 30 neurons (developmental NE algorithms (e.g. Gruau 1994) can provide larger, but structurally limited networks, e.g. by structure repetitions). Using much more neurons increases the search space significantly, because the number of parameters grows quadratic with the number of neurons due to the many additionally available synapses. Most NE algorithms are not able to find appropriate networks in such large search spaces. Also, recurrent neural-networks are highly dynamic and therefore can change their overall behavior significantly as the result of a single small parameter change. The larger the networks are, the more severe such drastic effects of the mutations can become, because often such larger networks are neuro-dynamically much richer than small networks.

This severe difficulty to evolve larger neural networks – despite the many successes with small networks – is also known as the *scaling problem* of neuro-evolution (Hornby et al. 2003; Maja J. Matarić 1996).

Many different NE algorithms have been developed since the first approaches (see section 2.4). Such algorithms can solve the classical benchmark problems, like pole-balancing, the XOR problem, the parity function or simple navigation tasks, with ever increasing performance (e.g. Gomez et al. 2008; Gruau 1994; Moriarty 1997; Stanley 2004), but the overall number of involved neurons is still small.

At the current state we are able to evolve small neuro-controllers very fast. But the now really interesting neuro-controllers with novel (neuro-dynamical) properties, that are not already described and examined, are expected to be found in the domain of – here called – *mid-scale networks*, i.e. networks with about 50 to 500 neurons. The class of mid-scale networks has been introduced in this thesis to separate that complexity class from small networks, as well as from very large networks (with thousands of neurons), as they are used in different contexts of neural networks research (e.g. modelling of biological neural networks, data analysis, time series prediction).

The scaling problem and the lack of appropriate NE methods hinder the development of such larger, interesting networks of the mid-scale domain and hereby slow down the progress of ER (Hornby et al. 2003).

2.4 State of the Art in Neuro-Evolution

Neuro-Evolution for Control

The evolution of neural networks as controllers for autonomous robots has a long tradition in the evolutionary robotics community (Floreano et al. 2008a; Harvey et al. 1997, 2005; Kodjabachian and Meyer 1995; Lungarella et al. 2003; Miikkulainen 2010; Nolfi and Floreano 2004). Embodied neural networks utilizing the sensors of a robot (sensori-motor loop, see Pfeifer 2002) can result in astonishingly robust behaviors in noisy environments. Also, using neural networks for behavior control may give insights into the neural organization of biological organisms. However, the construction of such networks is – apart from simple behaviors like tropisms (Braitenberg 1984; Hülse et al. 2005; Salomon 1997) – difficult to design by analytical approaches. This requires algorithms and techniques to create and optimize neural networks in an efficient manner. Robot control tends to rely on *recurrent* neural networks, because behaviors are usually dependent on inner states and require the use of feedback loops in the network and through the environment. As a promising approach to create such networks, a number of different types of evolutionary algorithms have been developed over the last decades.

Types of Neuro-Evolution. Evolutionary approaches for artificial neural networks can be roughly divided into *fixed topology* algorithms and *topology evolving* algorithms. The former class of algorithms usually works with networks having a fixed number of fully interconnected neurons or with layers of neurons with a fixed interconnection pattern. One problem with fixed topology algorithms is that the chosen number of neurons and their interconnection structure may not be appropriate and thus may not allow a solution at all. Second, the search space is usually very large right from the beginning, because all potential neurons and synapses of the network in its maximal dimensions are always included. Accordingly, it is difficult to find the optimal balance between a minimal search space and a topology that in the first place includes solutions to the problem.

Topology evolving algorithms on the other hand mostly have the ability to start with a small network (Elman 1993) and to add neurons and synapses to gradually increase the search space slowly over time or when it seems necessary (Hülse et al. 2004; Stanley 2004). Such algorithms keep the search space small without restricting the size and topology of the network over the course of the evolution. So, if solutions in small networks cannot be found, then the evolution can continue to search in larger network topologies, so that the initial network topology can be chosen much less thoroughly.

Many topology evolving algorithms can also be used to shrink already well performing larger solutions to smaller, easier comprehensible and easier to generalize network structures (Giles and Omlin 1994; Hülse et al. 2004)).

Genome Encoding. The genome representation of neural networks differs widely between the various algorithms. Common representations are matrices of synaptic weights, object structures of parameterized neuron types, lists of weights and nodes, binary strings, graph descriptions, trees and lists of construction rules, and much more. In the literature,

the genome encodings are commonly separated into two classes: *direct encoding* and *indirect encoding*. Direct encoding genomes directly represent the weights of all synapses of the network. One advantage of this approach is its simplicity, which makes it easy to use and implement. Also, most directly encoded networks are, in principle, free of a representation bias, because all kinds of networks can be encoded with this method. The mapping between genotype and phenotype is simple and works fine in both directions. This is relevant, if one wants to evolve a network structure not from scratch, but instead on top of an existing, manually prepared network structure.

In contrast, indirect encoding genomes evolve parameters that are used to indirectly derive the network topology and/or synaptic weights. Often, this involves the application of grammars or rule sets to construct the network. Such algorithms are often referred to as developmental algorithms (Bongard and Pfeifer 2003; Downing 2007; Eggenberger 1996; Gruau 1995; Hornby and Pollack 2001; Inden 2007; Lucas 1995; Meyer et al. 2003; Nolfi and Parisi 1995) or algorithms with artificial embryogeny (Bentley and Kumar 1999; Bongard and Pfeifer 2001; Stanley and Miikkulainen 2003a). Developmental approaches have the advantage that the genome size is often much more compact compared to direct encoding if applied to larger networks. This can speed up evolution, but usually also induces a strong representation bias to the evolution, because often not all kinds of networks can be encoded (with similar likeliness) with this type of algorithm. Also, whereas the construction of a phenotype from a given genotype is relatively easy using a developmental algorithm, the definition of the genotype from a given phenotype is often difficult and complicated. In addition, even the construction of a phenotype from a given genotype can already be time-consuming. Some indirectly encoding NE algorithms, like simulated growing synapses (Nolfi and Parisi 1991) or those using compositional pattern-producing networks (CPPNs) (Stanley 2007), may even slow down evolution, because the mapping from genotype to phenotype is computationally expensive.

Crossover. Although crossover is generally a valuable operator of evolutionary algorithms (Herrera et al. 1998; Langdon and Poli 2002), it is rarely used for the evolution of neural networks. The (arbitrary) combination of two well performing neural networks most often creates networks that behave very different from their parents. One reason is that the same functionality can be produced by very different neural structures, so that two equally well performing networks may be structurally very different. This is called the *competing conventions problem* (Belew et al. 1992; Hancock 1992; Schaffer et al. 1992). Combining such networks usually destroys the functionality of both networks, so that the created network performs much worse than its parents. Algorithms, that used variants of this simple crossover include e.g. Angeline et al. (1994); Bongard (2003); de Garis (1991); Doncieux and Meyer (2003); Spears and Anand (1991)

An algorithm that introduced crossover to neuro-evolution again was the NEAT algorithm (Stanley and Miikkulainen 2002a, b) and it is used by all derivatives of that algorithm (e.g. HyperNEAT (Stanley et al. 2009), Modular NEAT (Reisinger et al. 2004), NEON (Inden 2007, 2008), NEATfields (Inden et al. 2010)). The crossover here successfully enhances the evolution performance, because the crossover does not combine arbitrary network areas

of two networks, but instead uses historical markings to find exchangeable sections with comparable evolutionary history. Exchanging such network sections is much less destructive than arbitrary crossover. Using crossover in NEAT has been shown to improve the evolution process measurably (Stanley 2004).

Diversity. The conservation of a certain level of diversity is very beneficial for any kind of evolution, including neuro-evolution. One reason is that networks with very different topologies and weight distributions can show a similar fitness rating. Furthermore, even small enhancements of a behavior may require large changes to the network structure. To find an improvement, evolution often has to cross subspaces of the search space with a lower fitness first to end up with a higher fitness, i.e. it has to cross from one local optimum to another. Therefore, a premature convergence to a single class of solution candidates easily leads to an invincible local optimum. Although the diversity can be influenced for most evolution methods at the selection level (Horn 1997; Mahfoud 1995; Sareni and Krähenbühl 1998) or with classical diversity measures (e.g. fitness sharing, niching, see Sareni and Krähenbühl 1998), some neuro-evolution methods provide specific features to increase the diversity. To protect different lineages during evolution, NEAT uses a niching method based on historical markings (Stanley 2004), which allow a dynamic grouping of the networks into similarity classes. So, only similar networks have to compete against each other. ESP (Gomez 2003) and SANE (Moriarty and Miikkulainen 1994) increase the diversity by not evolving complete neural networks, but instead by evolving single neurons, that are assembled to a full network for each evaluation (cooperative co-evolution). Because the neurons are always combined in other ways, the resulting networks remain diverse longer. In addition ESP makes use of fixed niches that further enhance diversity. Using a similar approach on the synaptic level, CoSyNE (Gomez et al. 2006, 2008) also uses niches to delay convergence.

Network Shaping. *Shaping* (also called *iterative* or *incremental evolution*) is a quite common strategy (Bongard 2003; Dziuk and Miikkulainen 2011; Hülse et al. 2005; Miikkulainen et al. 2006; Mouret and Doncieux 2008) to simplify the evolution of a complex task by subdividing it into multiple, easier to solve sequential experiments and to combine their results. This obviously influences the order of development and the organization of the evolved controllers through the experimental setup. Shaping can be used with almost all neuro-evolution algorithms, because it only requires the iterative conduction of separate evolution experiments and the ability to evolve new structures on top of previously evolved ones.

With *network shaping*, on the other hand, I refer to the ability of a NE algorithm to allow shaping on the network level. This means that the network itself, not only the experimental settings, can be influenced with external measures (based on domain knowledge) to evolve structures iteratively in certain orders and organizations. This is very useful, if one wants to examine very particular neuro-controller approaches or if one wants to guide evolution into promising regions of the search space.

Examples are HyperNEAT (Stanley et al. 2009) and HybrID (Clune et al. 2009), where the evolving structures can be influenced by different choices of the positions and distribu-

tion of (sensor and motor) neurons. The same holds for algorithms with simulated growing synapses (Nolfi and Parisi 1991). In these algorithms, an elaborated choice of the neuron positions can significantly influence the network topology and the evolution success.

A simple way of network shaping is provided by the ENS^3 algorithm (Dieckmann 1995; Hülse et al. 2004). It provides a protection mechanism that allows the prevention of all mutations from selected neurons, so that (initial) network structures can selectively be included or excluded from mutations over the course of a series of consecutive experiments.

More control over the evolving network structures is provided by the SGOCE method (Kodjabachian and Meyer 1998; Meyer et al. 2003). SGOCE is a developmental algorithm, where the user can choose the number of evolving structure construction programs and specify in an editor, at which positions of the network these programs start to construct their subnetworks. Also, the orientation of the initial seeds of these programs can be specified, which allows the definition of symmetric, modular networks, which is especially useful for repetitive structures, such as multi-legged walking machines. A different approach uses the modular evolution algorithm ModNet (Doncieux and Meyer 2004a). Here, networks are composed of non-hierarchical evolvable neuro-modules with one or two fixed input and output neuron. Whereas the network structure of these modules can be fully evolved, it is also possible to specify predefined neuro-modules to be used during evolution, so that the evolving network structure can be influenced (Doncieux and Meyer 2005). Also, ModNet supports *connectivity patterns* to influence the module connections (Doncieux and Meyer 2004b) in the initial generation.

Bootstrapping. One important problem with evolutionary algorithms and with neuro-evolution of controller design in particular, is the bootstrapping of the evolution. As for all gradient based search algorithms, the first few generations are especially important for NE algorithms, because until then, promising candidates should have been found that are within the range of an acceptable local optimum. If so, the gradient based search will have a good chance to converge to such a desired local optimum. If, however, there are no promising candidates in the entire generation, then the search will take very long or will not succeed, because new candidates are preferably generated close to the best, but in this case bad performing individuals. This is particularly severe the more low performing local optima exist in the fitness landscape. Then it becomes more and more likely that the evolution gets stuck in one of these undesired optima. Therefore, it is important to start with – and preferably to keep – a high diversity in the population. One approach to tackle this problem is shaping and iterative evolution (see above), because the simpler the experiments are, the smoother the fitness landscapes usually become. Finding a proper order for such iterative sub-tasks, however, is not always easy. To avoid fixed sequences of sub-tasks at all, Mouret and Doncieux (2008) perform a *multi-objective evolution* (van Veldhuizen and Lamont 2000) on all sub-tasks at once, leading to an increased diversity.

A second important approach is network shaping, notably the predefinition of network structures, so that it becomes more likely that random extensions of such an initial network results in promising candidates. In combination with suitable search space restriction measures, this increases the likeliness for good candidates.

Also, an effective strategy is to use very large populations to guarantee many different randomly initialized candidates. But then the performance of the algorithm drops down. A good compromise is the use of interactive evolution algorithms, such as ENS^3, where the population size and the probabilities for mutations can be adapted after the first few generations as a reaction to the observed progress.

A similar effect can also be achieved using an explicit initialization generation in the algorithm, so that the population in the first generation is larger than in the remaining generations. The additional computational effort then is only required in the first generation to find suitable initial candidates with a broad search space sampling, whereas all following generations only evaluate the normal, lower number of individuals.

With the same goal, the very different strategies *novelty search* (Lehman and Stanley 2011; Risi et al. 2009) and *behavioral distance measures* (Mouret and Doncieux 2009a, b) can be used to actively search for differently behaving controllers with non-zero performance. This helps to create an early and long-lasting diversity in the population.

Coping with Large Networks

As the experiments in evolutionary robotics become more difficult and the corresponding neural networks become larger, neuro-evolution approaches try to increase the success rate of the evolutionary search with different strategies. Three important strategies should be mentioned here: *structure reuse*, *incremental complexification* and *search space restriction*.

Structure Reuse and Modularity. Structure reuse means, that a single neural structure encoded directly or indirectly in the genome, can be reused in different parts of the network. This often relates to modular structures, or neuro-modules (Auda and Kamel 1999; Happel and Murre 1994; Pasemann 1995, 1996). The main advantage of reusing subnetworks is the possibility to develop functional building blocks, which can be reused in multiple parts of the network, without having to be reinvented by evolution multiple times. An example in biological systems are the cortical columns in the brain of mammals (Horton and Adams 2005)). Modular subnetworks are explicitly addressed in algorithms like Modular NEAT (Reisinger et al. 2004), ModNet (Doncieux and Meyer 2004a), CoSyNE (Gomez et al. 2008), Cellular Encoding (Gruau 1994, 1995) and its derivatives (e.g. Christoph M. Friedrich 1996), ENSO (Valsalam 2010; Valsalam and Miikkulainen 2009), ESP (Gomez and Miikkulainen 1997), in SGOCE (Meyer et al. 2003) and in Calabretta et al. (2000). Another approach to reuse structures is the generation of networks with CPPNs (Stanley 2007) and the related evolution method HyperNEAT (D'Ambrosio and Stanley 2007; Gauci and Stanley 2007; Stanley et al. 2009). HyperNEAT does not evolve neural networks directly, but evolves function networks (CPPNs) that generate the weights of a fixed network topology, resulting in often repetitive, symmetric structures similar to neuro-modules.

Incremental Complexification. Structure reuse is one efficient method to reduce the search space. Another common strategy is the incremental complexification of networks, as done in NEAT and its derivatives (Stanley 2004), ESP (Gomez and Miikkulainen 1997),

ModNet (Doncieux and Meyer 2004a), ENS^3 (Dieckmann 1995; Hülse et al. 2004; Pasemann et al. 1999), NEATfields (Inden et al. 2010) and most developmental approaches. Evolution is started with small or minimal networks – for instance only including the sensor and motor neurons – and explores this search space until no further improvement is observed. Then, repeatedly, the search space is extended by adding additional neurons and synapses, and the new search space is explored. The assumption is that if solutions with simple network structures exist, then it would be beneficial to find them first, because they are expected to evolve faster than complex networks with the same functionality. This strategy has been shown to accelerate the evolution process in a number of experiments (Stanley 2004).

Further Search Space Restriction and Topology Heuristics. Another way to reduce the structural search space is the consideration of symmetries and other topological peculiarities – like the positions of neurons – and the application of heuristics for the topology. Developmental approaches with growing synapses, for instance, usually implicitly implement the heuristics, that local, close-by neurons should have a higher probability to be connected than neurons further apart (Cangelosi et al. 1994; Meyer et al. 2003; Nolfi and Parisi 1991), leading to less densely connected networks with locally connected groups of neurons. Also strongly depending on the positions of the neurons is HyperNEAT, which can exploit topological neighborhoods. Symmetries can be used in many experimental scenarios, because the bodies of most animats have one or more axes of symmetry. This can be reflected in the network topology and strongly reduces the search space. ENSO (Valsalam and Miikkulainen 2009) is an example of an algorithm that systematically breaks down symmetries step by step, starting with a predefined, highly symmetric network, hereby exploring the more symmetric – and thus smaller – search spaces first.

Another – already mentioned – search space restrictions is the manual exclusion of certain neurons or synapses from being changed during evolution, as it is possible with the ENS^3 algorithm.

To summarize, the number and variations of neuro-evolution algorithms proposed during the last years is astonishingly high. But this is understandable, taking the anticipated impact on the fields of neuro-control, neuro-cybernetics and evolutionary robotics into account. Each evolutionary method has its own benefits for a specific application area, where it outperforms other methods. The neuro-evolution method proposed in this dissertation is in this spirit similar in that it focuses on a specific niche of neuro-evolution problems: the systematic search for neuro-controller variations in the domain of mid-scale networks.

Chapter 3

ICONE
Interactively Constrained Neuro-Evolution

This chapter introduces the **I**nteractively **Co**nstrained **N**euro-**E**volution (ICONE) method, developed to tackle the problems described in the previous sections. ICONE has to be understood as a general method, that can be combined with different selection methods, genome representations and mutation operators. Section 3.1 discusses the aim and application domain of the method to point out why and in which context the method is applicable and how it can enhance the network evolution process. That section also gives a brief summary of the method, so that the method details can be understood in context right from the beginning. Section 3.2 then describes the genome structure as it is used for the experiments of this thesis. The chapter is concluded by a description of the actual algorithm (section 3.3), giving details about the mutation phase, constraint resolving and the used crossover operator.

3.1 Overview

Application Domain. Despite the many available neuro-evolution approaches (compare section 2.4) there is still a lack of methods *practically* applicable for the evolution of *complex* neuro-controllers in the context of evolutionary robotics (ER), neuro-cybernetics and neurorobotics research. The aim of this research is to develop neuro-controllers for animats, such as robots or simulated creatures, and to understand the underlying neural and neuro-dynamical principles. In its quite long history (Floreano et al. 2008b; Harvey et al. 1997, 2005; Meeden and Kumar 1998; Meyer et al. 1998; Nolfi and Floreano 2004; Walker and Oliver 1997) this research field has made much progress using varieties of neuro-evolution methods. But due to the large search spaces, the controllers are restricted

to relatively small and comparably simple networks controlling simple agents. To gain *new* insights into the neural organization of control, larger, more complex networks are desired, preferably recurrent networks with non-trivial topologies. Also, the controlled agents should be more complex, providing rich sets of different sensors and actuators. This allows for a complex interplay of internal and external stimuli, affecting multiple, partially independent motor systems. Evolving neuro-controllers in this highly interesting domain is very difficult, because – as already mentioned – the search space exponentially increases with every new neuron in the controller. Consequently, evolving *interesting*, non-trivial neuro-controllers for animats requiring mid-scale networks (50 to 500 neurons, see section 2.3) is still a problem and the success rates are very low.

A second problem in the context of ER and neurorobotics is that it is often not just desired to find the most likely solution to a control problem, that usually comes up during neuro-evolution. Instead, any new distinct solution approach, even those not optimally solving the problem, are of interest due to their potential of providing new neuro-dynamical or organizational principles. As a result, the evolution should provide ways to bias the search in different directions to also explore less likely solution spaces. This is especially important for problems where theories have been proposed – e.g. by biologists or analytical reasoning – that should be *systematically* tested. This is very intricate to achieve with most neuro-evolution methods, because such problems require the compliance with given assumptions and restrictions. These somehow have to be enforced onto the evolving controllers, usually done through the fitness function or by writing specialized evolution algorithms for the investigated approach (e.g. von Twickel 2011). This is often difficult to do, restricts the flexibility of the evolution and makes it difficult to dynamically react to results or failures of the neuro-evolution rapidly.

To tackle these problems, the ICONE method has been developed. The basic idea of the method is to allow the supervised induction of arbitrary constraints and restrictions on the evolving networks, affecting the structure and the weights of the developing neural networks. With such constraints the search space can be greatly restricted based on domain knowledge, user experience and guessing, to render successful evolutions possible even with larger networks. So, although the evolving networks can be quite large, the corresponding search spaces are not larger than that of small, unconstrained networks, for which a successful evolution is feasible. With this, ICONE is suitable for the evolution of *mid-scale* networks, which today is still a difficult domain in the context of ER. In addition to a pure search space reduction, ICONE also allows to bias the search towards specific solution approaches, allowing the investigation or application of known or assumed neural control principles. The induced restrictions and constraints can be interactively controlled by the user, so that an adaptation as a reaction to intermediate results during the course of an evolution is possible.

Generality. In difference to other search space restricting methods (see section 2.4), the ICONE method was designed to be applicable to a wide range of scenarios without requiring specific prerequisites, such as inherent symmetries, special regularities, size restrictions or fixed topologies. However, with this evolution method many of such restricting features can be fully exploited where available. To achieve this, the method has been devised to be

easily extensible and adaptable to many different functional, structural and organizational conditions, such as different animats, research foci, control approaches and neural network models. Furthermore, the method was designed to minimize the structural bias originating from the algorithm itself (not to be mistaken with intentional biasing by the user), so that all kinds of networks are, in principle, evolvable.

The core of the ICONE method focuses primarily on the reproduction and mutation phase of the evolutionary algorithm, leaving all other parts of the algorithm open. Thus, the method can be combined with different selection methods, fitness measures, niching techniques, multi-objective evolution approaches and the like. This is important, because their individual strengths and weaknesses for the evolution progress strongly depend on the task. Therefore, they should not be fixed in the algorithm, but instead be allowed to be used wherever this enhances the evolution success for a particular experiment.

The overall generality of the method allows a wide application of the method, but comes with a price regarding performance. Accordingly, the focus of this method is not to provide yet another and faster method to solve the same (already solved) benchmark problems, but instead to allow the guided search of neuro-controllers in domains where most other evolution algorithms simply fail due to the large search spaces. This means, that performance is considered less important than getting varieties of complex solutions at all. Also, this method should not be expected to be an unsupervised universal problem solver, but merely an interactive assistant tool to support a user to find varieties of controllers in iterative, supervised steps. This is also the reason why this thesis does not try to directly compare and benchmark the method with other neuro-evolution approaches, because each evolution run is strongly biased by the ideas, experiences and preferences of the user. This makes a formal, objective comparison with other methods very difficult and vulnerable to criticism.

Method Overview. Evolution with the ICONE method usually requires the following steps. First, as for all evolutionary algorithms, the user has to design the experiment for the given problem, including an evaluation environment – such as a simulation – and a fitness function. The approached problem may be the search for a behavior controller, but also more elaborate questions, like the application of a specific control structure, the use of a neural paradigm or the confirmation of a given (control) hypothesis. As a second step, one or more initial networks have to be prepared, which is here called *modularization* (Rempis and Pasemann 2010). Hereby, the initial networks are structured into modules and groups to enhance comprehension, to restrict possible synaptic connections and to encapsulate and group functional and logical parts of the network. Also, predefined structures (here called *peripheral structures*, see section 5.1.1.2) can be added to the network to simplify the bootstrapping of the evolution and to bias the search towards specific solution approaches. As part of the modularization phase, the initial networks are further constrained by adding so called *functional constraints* to the groups and modules. These functional constraints can be used to enforce any desired structural feature on the network. They ensure that only compatible networks can evolve, i.e. networks that *do not* violate any of these constraints. This leads to a strong reduction of the search space and biases the search to very specific configurations. Such initial netoworks are called *constraint masks*, because they define the

restricted search space the evolution takes place in. The final step then is the actual – iteratively and/or interactively conducted – evolutionary search.

The described steps are, of course, not as sequential as described. Iterative evolution often requires the user to redesign the initial networks, the fitness function, the evaluation environment and sometimes even the experiment itself. Hereby, the user can learn by direct observation of the experiments, which modifications may be necessary to locate the promising subspaces of the search space that may contain the desired solutions.

Graphical Auxiliary Tools. Working directly with large neural networks, especially during the preparation of initial networks and during interactive evolution, requires graphical tools to do so efficiently. Without such a tool networks easily become inconvenient to work with, especially when modularizing and constraining the networks. Therefore, a graphical network editor has been implemented that allows the construction, modularization, constraint control and analysis of neural networks. Details on this editor can be found in appendix D.2. The availability of such a network editor should be kept in mind when reading the sections about the – at first glance seemingly quite complicated – topics of *constrained modularization* and manual network preparation.

The next sections describe the ICONE method and its related components in detail. The focus of this thesis is the method itself, not its actual implementation. However, to test the method and to perform the experiments described in chapters 7 and 8, the ICONE method was implemented as part of the NERD toolkit, which is described in appendix D. This open source project can be used to replicate the results and to start with own evolution experiments.

3.2 Genome

The genome encoding for the ICONE method may vary from implementation to implementation. This ensures that the method can be applied in combination with features from other evolution algorithms and problem-specific operators. However, some requirements are mandatory and are described in detail in the next sections. The overall genome encoding, that is used in this thesis, is then summarized in section 3.2.6.

3.2.1 Neural Network Encoding

Direct Encoding of Large Networks. As described in section 2.4 genomes can encode neural networks directly or indirectly. While the main opinion today is that indirect encoding is more effective and better scaling than direct encoding when used with larger networks (Gruau et al. 1996; Meyer et al. 2003; Yao 1999), ICONE uses a direct encoding scheme. For the research context ICONE is used in, the direct encoding has significant advantages over indirect encodings:

- The direct encoding does not add an encoding bias to the evolved networks, i.e. it does not restrict the possible (or likely) network structures that may evolve.

- Direct encoding allows a much better comprehension of the relation between genotype and phenotype, especially when a partial initial network should be designed manually. The mapping from an indirect genome encoding to its phenotype representation usually is easy, while the other way round is often not definite, very difficult or sometimes simply not possible. In the given context, this is a problem, because we want to seed the evolution with partially working networks, for which the indirect encoding may not be known in advance.

- To achieve small changes in a network phenotype, only small changes are required in a direct encoding schema, too. The same small desired modifications of the phenotype may require an entirely different indirect encoding. This further hinders the interactive evolution paradigm fostered by the ICONE method. Also, indirect encoding makes the definition of constraints (see section 4) very difficult, because constraints, seen from the user perspective, should constrain the phenotype. But they have to do this by adjusting the genotype. Automatically finding the proper indirectly encoded genotype that represents a certain change in the phenotype is difficult and time-consuming and thus is a problem for the proposed approach.

The major problem of direct encoding genomes for larger networks is its comparably large parameter space, which makes evolution more difficult. Indirect encodings usually have a smaller parameter space because they compress the genome at the cost of a reduced set of representable phenotypes (Yao 1999). However, ICONE uses explicit constraints to reduce the search space and therefore is not so much affected by that problem. Because of the relatively easy definition of constraints with direct encoding genomes, properly constrained genomes often provide even much smaller search spaces than unconstrained, indirectly encoded representations.

General Genome Requirements. Genome encodings for ICONE should be as expressive as possible to allow many different network architectures and accordingly the investigation of many research questions, including heterogeneous networks with mixed transfer functions, different activation functions, higher-order synapses, adaptive synapses and learning rules. The hereby increasing search space is not a problem per se, because ICONE allows constraints on the networks to pick only those components of the possible genome variations that are actually needed for a certain experiment. Moreover, the constraints also allow the restriction of any parameter variability only to differentiated, local areas of the network, so that the search space is only increased in a few, well defined areas.

A second requirement for all genome representations is that each network element (neuron, synapse, neuron-group, neuro-module; see next sections) has a unique identifier. This identifier is important, because constraints usually work with element IDs to refer to certain elements in the network. It is further important that these identifiers remain valid when network elements are replaced by others having differing unique IDs, for instance during a crossover. For this, the implementation of the ICONE method requires a mechanism to notify all constraints if such a replacement takes place, so that they can adjust their references accordingly to remain functionally valid.

Element	Attributes
Neuron	Bias Transfer Function Activation Function
Synapse	Weight Synapse Function
Neuron-Group	Member Neurons Constraints
Neuro-Module	Member Neurons Member Modules Constraints
All Elements	Position in the Network

Table 3.1: An overview of the basic network elements and some of their evolvable attributes. The element position is an optional, but very useful attribute, that is required for many connectivity heuristics and position dependent constraints.

3.2.2 Neurons and Synapses

The basis of the direct encoding schema of each ICONE genome is a representation for neurons and synapses. All parameters of these network elements, such as the synapse weight, neuron bias, activation function, synapse function and transfer function (see table 3.1), should be evolvable and parameters of the genome. These parameters should be adjustable independently of each other, so that arbitrary heterogeneous networks with mixed neuron models are possible. This increases the supported evolvable network structures and the scope of feasible experiments. The larger search space coming with these additional parameters can be reduced again during evolution via constraints, so that the search space is not necessarily blown up.

The bias value of the neurons hereby should be a separate parameter of each neuron, instead of a synapse connecting an always active *bias neuron* to the affected neuron, as many neuro-evolution algorithms do. This allows a separate control of the insertion and removal of bias terms and that of synapses. This is especially important for larger networks, because the probability to add a synapse between a neuron and such a *bias neuron* diminishes with the increasing number of available neurons, making bias connections more and more unlikely, the larger the networks become.

Neurons and synapses, like all other network elements, also provide a list of so-called *network tags*, that are described in detail in section 3.2.4, and the mentioned unique identifier to allow an unambiguous addressing of each network element. The details of the genome representations are summarized in figure 3.3 on page 34.

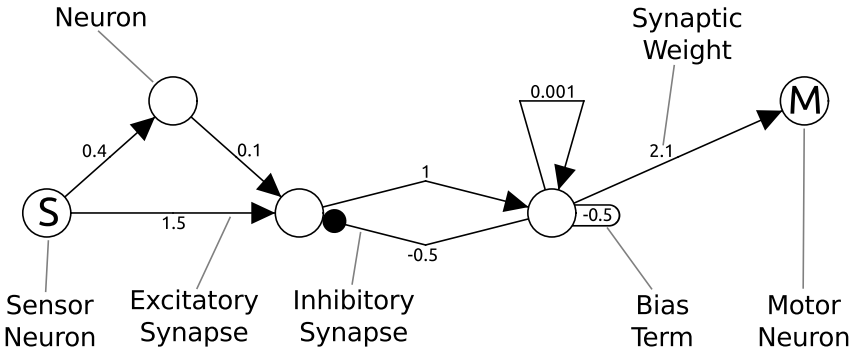


Figure 3.1: Basic neural network elements and their main properties. The figure also shows the symbols used in the neuro-controllers depicted in this thesis. A full overview on all symbols can be found in appendix A.

3.2.3 Neuron-Groups and Neuro-Modules

On top of the basic neural network an additional layer is defined, consisting of so-called *neuron-groups* and *neuro-modules*. These additional network elements are used to organize the network logically and hierarchically and to enable the definition of *constraint masks* by the user. However, the effects of these optional elements are limited to the mutations during evolution and do not have a direct effect on the overall *function* of the network. Hence, the removal of this layer always leaves a still intact, fully operational neural network. This allows an unproblematic use of evolved neuro-controllers in environments that only support the common neuron models, for instance when a controller is transferred to a physical robot or has to be evaluated in a simulator with a fixed neural network model.

Neuron-Groups. A neuron-group is simply a *set of neurons* with the following properties:

- Neuron-groups can be restricted by constraints (see section 3.2.5), that only affect the members of that group.
- Neuron-groups, like all network elements, provide a list of so-called network tags (see section 3.2.4).
- Neuron-groups make the represented set of member neurons addressable via a single, unique identifier, that can be used (e.g. by constraints and network tags) to refer to all member neurons of that group at once.
- Neuron-groups may provide a human-readable name for the represented group of neurons, which enhances the comprehensibility of the network and simplifies the handling of larger networks.

Neuron-groups are the basis and the contact point for the main search space restriction mechanism of ICONE: the *functional constraints* (section 3.2.5). Neuron-groups also allow

a logical grouping of neurons, for instance to define network layers, or they can be used in functional constraints to refer to the sets of corresponding neurons (e.g. to refer to all sensor neurons, all motor neurons, all neurons belonging to one side or a single body part of an animat, all neurons with a certain function or role, etc.).

A neuron-group is represented in the genome as a list of member neurons, a list or parameterized constraints, a list of property tags and a unique identifier (see figure 3.3 on page 34).

Neuro-Modules. Neuro-modules are extended neuron-groups. In addition to the features of neuron-groups, neuro-modules have the following properties:

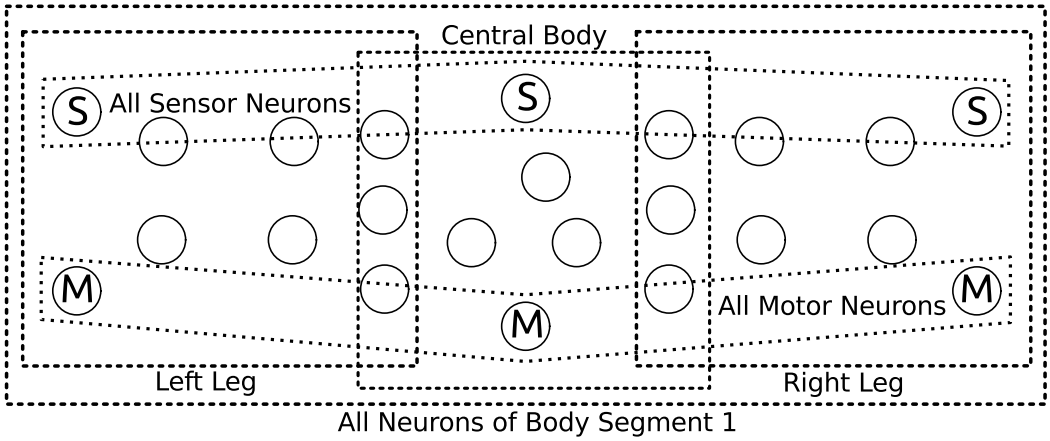
- In difference to neuron groups, neuro-modules cannot overlap with other neuro-modules, i.e. each neuron can only be member of a single neuro-module at the same time.
- Neuro-modules provide a well defined neural interface of input and output neurons.
- Neuro-modules can be stacked hierarchically, i.e. neuro-modules can be members of other neuro-modules and form hierarchies of *submodules*.

With these properties neuro-modules encapsulate distinct subnetworks and separate these structures from other parts of the network. Neurons inside of a neuro-module cannot be connected to neurons outside of that module. Exceptions are the interface neurons. These neurons are specially marked to be visible outside of the module, either for outgoing synapses (output neurons), incoming synapses (input neurons) or both. A normal interface neuron is only visible outside of its own module. If an interface neuron is a member of a submodule, then its visibility can be increased, so that a neuron additionally belongs to the neural interface of the parent modules of its own module. The depth of this visibility, i.e. the number of parent module levels the neuron is visible at, can be chosen separately for each neuron and direction. Put together, neuro-modules are especially useful to structure a network not only logically, but also hierarchically and spatially.

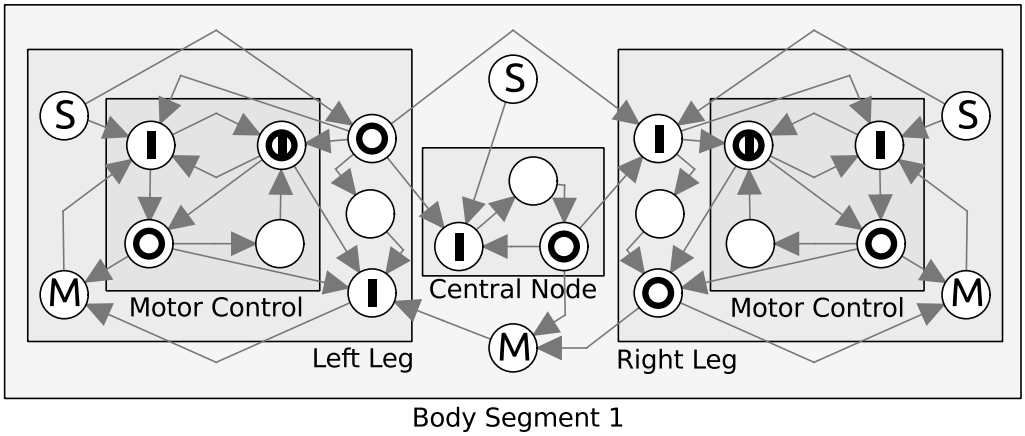
The extensive use of neuro-modules also biases evolution towards local synaptic connections, because most neurons only can connect synapses to a small set of local neurons or the interface neurons of neighboring neuro-modules. This is a powerful mechanism to bias the evolutionary search towards a local processing of related neurons, instead of allowing arbitrary synaptic connections. This is considered to be important because the larger the networks become, the more probable it becomes that global synapses will connect unrelated neurons and network areas, rather disturbing useful (local) processing structures than enhancing them. These global connections can be reduced by using neuro-modules to encapsulate and shield their member neurons and submodules. A properly modularized network (see section 5.1) does not only bias search towards a more local processing, but also restricts the search space by removing a significant number of possible, but most often obstructive connections. Furthermore, controllers are expected to evolve functional structures distributed over – and interacting with – a much smaller part of the network, which makes the isolation, analysis and comprehension of these structures easier. Of course, one

should not expect that functional structures evolve nicely within a single neuro-module, but nevertheless – due to the restricted connectivity – functional units have the tendency to be less interweaved.

In the genome, neuro-modules are represented as a list of member neurons, a list of parameterized constraints, a list of property tags, information about the interface neurons and a unique identifier (see figure 3.3 on page 34).



(a) Examples of neuron-groups, showing the logical grouping of neurons according to their role, their associated locations on the animat body and their type. The figure also shows the human-readable names of the overlapping neuron-groups.



(b) Examples of neuro-modules illustrating the hierarchical and spatial separation of the same network. Interface neurons of the neuro-modules are indicated with an I (input) and an O (output). The shown synapses illustrate the visibility of interface neurons in the submodule hierarchy and the resulting tendency for a local processing.

Figure 3.2: Examples for neuron-groups and neuro-modules and their different use on the same neural network. Both, the neuron-group and neuro-module definitions can coexist on top of the same network.

Advantages of using Neuro-Modules and Neuron-Groups. Neuro-modules – as they are used with ICONE – are functionally different from those used in other NE approaches, where the modules are usually used to represent repeatable or exchangeable network structures and to compress the genome. Although, with suitable constraints, neuro-modules in ICONE can also serve a similar purpose, the here proposed neuro-modules (and neuron-groups) are much more flexible and allow very diverse ways of reducing the search space and of controlling the evolution process:

- Neuro-modules structure a network to enhance comprehensibility and to reduce the search space by reducing the number of valid synaptic connections.
- The use of submodules organizes a network hierarchically leading to an increased control of the interaction of functional structures by the user.
- Neuro-modules are a suitable way to *reuse* previously evolved network structures during evolution (see section 5.1.3).
- Neuro-modules allow a crossover operator that is less destructive than standard crossover (see section 3.3.3).
- Neuro-modules clearly separate functional neural structures behind well defined interfaces, which helps to preserve the function of a module during evolution and to avoid harmful synaptic connections to 'internal' processing neurons.
- Neuro-modules and neuron-groups allow very detailed, complex restrictions and a shaping of the desired network structure with constraints (see section 4 and 5.1).

Limitations of Neuro-Modules and Neuron-Groups. Structuring networks with neuron-groups and especially with neuro-modules naturally is not without certain limitations and potential drawbacks. The most obvious one is that a good structuring of a network requires domain knowledge and therefore has to be done manually in advance. Although modules – in principle – can also be defined automatically during evolution, their expressiveness concerning the above advantages would be quite limited. Also, as a problem of the strictly distinct sets of neurons in neuro-modules, the length of synaptic pathways can increase. To connect two separate neuro-modules, an additional synapse between an output of the first and an input of the second module is required, that could be avoided by merging the two neuro-modules instead (such as in the SGOCE method (Kodjabachian and Meyer 1998)). However, such merging can lead to side-effects and makes the use of functional constraints more difficult. As a countermeasure for a longer signal processing time caused by these longer synaptic pathways (i.e. the number of update steps required to propagate a signal through the network), selected synapses may be replaced by special order-dependent neurons (section 6.9) to increase the responsiveness of a neuro-controller, if this turns out to be a considerable problem for a specific experiment.

3.2.4 Network Tags

A unique characteristics of ICONE networks is that all network elements (neurons, synapses, neuron-groups, neuro-modules and also the network itself) can have arbitrary lists of so-called *network tags*. These tags are simple string-string pairs that can be used to add additional information to each element. Adding or changing such a network tag is here called *tagging*. The names and meanings of the tags may differ from implementation to implementation, because the meaning and function of each tag is defined solely by the operators, scripts or constraints that read and interpret the tags. Because of this, tagging is a universal way to add arbitrary information to network elements. It allows the introduction of new tags at any time without the need to change the genome implementation. This simplifies the rapid introduction of new mutation operators, constraints and scripts, that often require additional information embedded to the network. A list of useful network tags, as they are used with the NERD Toolkit for the experiments of this thesis, is given in appendix C.

Network tags influence the evolution, function and appearance of neural networks in many ways:

Simple Constraints and Property Protection. Network tags can be used to directly limit the number of evolvable parameters of an element, which results in a simple reduction of the search space. This is similar to the protection mechanism of the ENS^3 algorithm (Hülse et al. 2004), but is much more detailed, because any parameter of the network elements can be protected separately. These *protection* tags are read by the corresponding mutation operators (section 3.3.1) leading to an exclusion of accordingly tagged elements with respect to certain mutations. For instance, a bias term of a neuron can be fixed, a synapse may be protected against removal, a module may be protected against the insertion or removal of neurons, or any change to an entire network sub-structure may be prohibited.

Hints for Mutation Operators. Tags can also be used to influence mutation operators, e.g. to define preferred synapse targets for new synapses, to limit the number of synapses for certain neurons, to define preferred connection pathways between modules, to mark a new neuron to be initially connected to the network, to influence the interconnectivity density within a module or to restrict the number of neurons of a module. Special network tags are also required to configure the modular crossover operator (section 3.3.3) for a network.

Fine-Control of Evolution. Global parameters of the evolution, such as mutation rates and change deviations for bias terms and synaptic weights, can be locally overwritten with network tags. Also, the valid range of such network attributes can be locally redefined, for instance to keep a synapse weight within very specific limits. These tags are very useful to realize a mix of evolution operator settings throughout the network. This is important, because in large networks, some network areas are much more fragile to variations than others. So having only a single parameter setting for all network elements always leads to a conflict between such fragile areas and such that require very large changes to have an

effect at all. Tagging such network areas based on domain knowledge reduces this problem. In the NERD implementation of ICONE such local settings can be given in absolute values (e.g. fixed probabilities) or as a percentage rate relative to the current global setting. The latter preserves the important (interactive) adjustments of the evolution settings during the course of an evolution experiment, whereas the required relative differences of the evolution settings between network areas of different mutation sensitivity is kept intact.

Function Control of the Network. Some network tags can also be used to change the behavior of a network element. An already mentioned example are *order dependent neurons* (section 6.9), whose update order is influenced via network tags to improve the reactivity of neuro-controllers. Another important tag is used to flip the activation range of neurons, which means, that their output is reversed. The flipping of activation ranges of motor and sensor neurons is an important measure during modularization (section 5.1) to make networks compatible with symmetries and regularities.

Identification and Configuration of Elements. Network tags of this kind provide tracking and tracing information (such as creation date, origin of elements), that can be used to analyze the evolution progress. Some constraints also require special identifiers in addition to the common unique identifiers to automatically collect affected elements according to a role or type, that can be provided as network tags.

Auxiliary Tags. Such tags are used for instance by a network editor or during the export of networks into other formats (e.g. machine code for a robot). They may hold the location and size of an element in an editor, the hardware addresses of sensor and motor neurons on a target robot, and the like.

Temporary Tags. Mutation operators, constraints and scripts can also use tags to temporarily leave information during the mutation phase, so that a complex interaction of the operators becomes possible. Examples are markers to identify newly inserted elements or hints left by operators to affect subsequent operators. Temporary tags are automatically removed after the mutation phase and therefore are rarely noticed by users.

3.2.5 Constraints

Functional constraints are the primary feature of ICONE used to restrict the search space and to bias the search towards specific network structures. A constraint hereby is a hard requirement of the network structure, its organization or its weight distribution that cannot be violated during evolution. In difference to other constraint based algorithms, that influence the selection of individuals based on their degree of violation of constraints (Coello 2002; Deb 2000; Michalewicz and Schoenauer 1996), ICONE does not allow any constraint violations at all (see section 3.3.2). Networks not meeting all given constraints are removed from the evolving population.

However, evolution does not have to find the proper network configurations that fit all constrains simply by trial and error of mutations, which would be much too time-

consuming. Constraints in ICONE are called *functional constraints*, because they do not only describe certain limitations and requirements, but also provide functions to enforce these requirements on a network by actively changing it. This means that a single mutation may trigger a cascade of constraint function activations to restructure the entire genome to enforce the compliance of all given assumptions. As a result, any arbitrary mutation usually automatically results in a valid network configuration, after the functional constraints of the network have been executed. If there are conflicting constraints in the network and the constraints cannot be resolved, then the network is discarded. Networks with conflicting constraints therefore cannot evolve, but due to misconfigurations during the modularization phase by the user, such networks are still possible. Because networks outside of the constraint limitations *cannot* evolve, the choice of the constraint mask is very important for a successful evolution to avoid irresolvable constraint configurations or too restrictive constraint masks that prevent a solution for a certain control problem.

Functional constraints are of algorithmic nature, therefore they have to be implemented in the evolution software. Standard constraints (see section 4 for an overview on the implemented constraints) can be implemented natively to the software, while other, more specific constraints should be made possible via scripting or other dynamic extensions of the software. Otherwise the power and applicability of constraints is reduced. Constraints usually are parametrized, so that their function can be adapted to a given problem.

The search space restriction with constraints can be very effective, because many degrees of freedom in a networks become dependent on each other. This means that, in principle, all attributes of the network can still be mutated during evolution, but because each mutation may lead to the adaptation of dependent attributes, in the end only a limited set of attribute combinations are possible. To effectively use this search space restriction during evolution, domain knowledge has to be applied to the networks. These domain knowledge based constraints however can not only be used to restrict the search space, but also to allow the direction of evolving controllers towards very distinct topologies or organizational principles.

Constraints in ICONE are limited in scope to simplify their application and configuration in larger networks and to avoid side-effects between different network areas. Therefore, constraints only operate on neuron-groups and neuro-modules, usually only affecting the network elements of their corresponding group. This simplifies the usage and configuration of constraints in large networks, because only a subset of the network is affected. It also allows the definition of constrained neuro-modules as building blocks (see section 5.1), that – equipped with suitable sets of constraints – allow mutations within well defined limitations. Because each constraint belongs to a specific module or group, all constraints are automatically moved and adjusted whenever a module or group is relocated, removed, copied or exchanged.

3.2.6 Final Genome Encoding

The complete genome representation of a neural network is schematically summarized in figure 3.3.

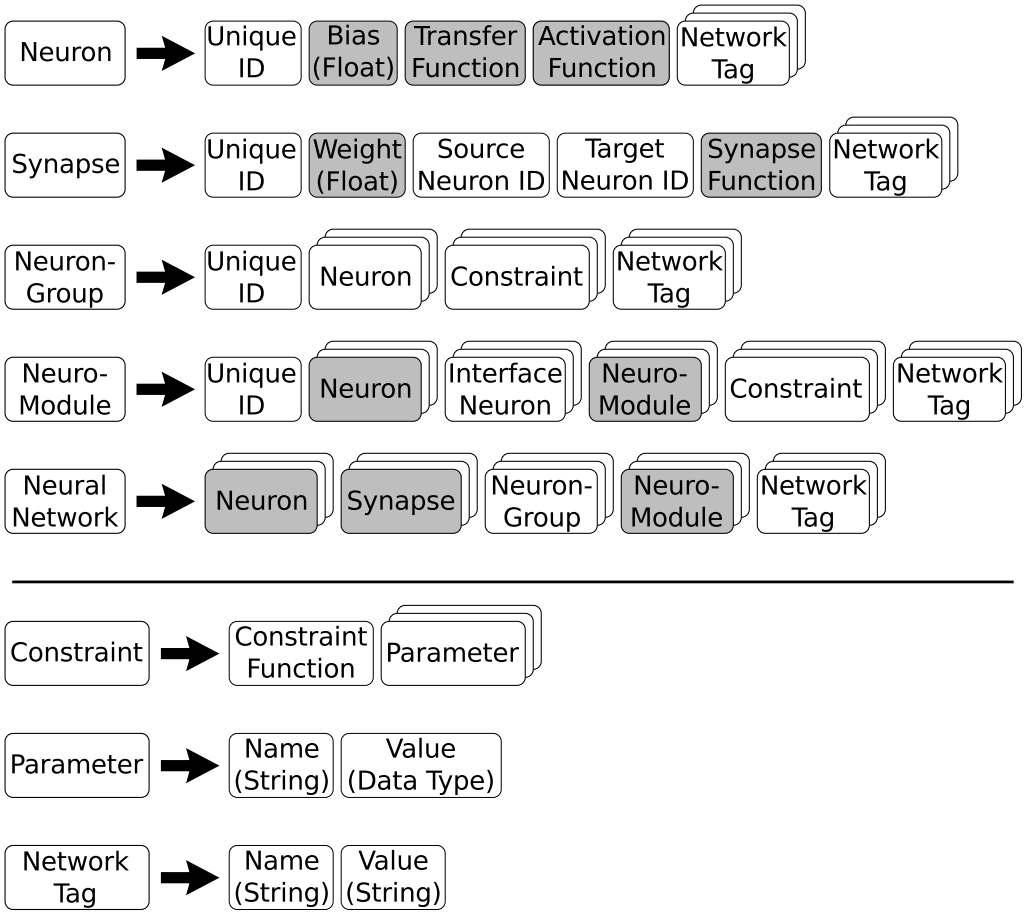


Figure 3.3: Overview on the object structure of the genome. The elements highlighted with a gray color indicate parts of the genome that actually are mutated during evolution.

Here, it also becomes clear, which parameters typically can be varied (inserted, removed, exchanged or modified) by mutation operators during evolution (gray elements) and which parameters can only be modified by a user, by functional constraints or – in case of the IDs – not at all. The constraint function of a constraint object represents the actual implementation of the functional constraint, that has to be realized according to the used programming language, e.g. as a script, a function pointer or a function object. Similarly, the synapse functions, activation functions and transfer functions refer to corresponding functions implemented in the programming language.

3.3 General Algorithm

The general algorithm of ICONE is - similar to most evolutionary algorithms - a cyclic application of evaluation, selection, reproduction and variation to a population of individuals (Algorithm 1). The evaluation and selection of the individuals is not a rigid part of

the ICONE method and can be varied according to the performed experiments. Many evolution experiments can be significantly enhanced by using suitably selected evaluation and selection methods, like shared fitness, multiple-try evaluation, niching, multi-objective selection, migration models, and the like. The ICONE specific part of the algorithm is the reproduction and variation phase. This phase is composed of a module-based crossover operator during offspring generation, a variation chain with an extensible number of mutation operators, and a constraint resolver as part of that variation chain, used to apply constraints on the mutated networks. The details on these specific parts are described in the remainder of this section.

Algorithm 1: Main Evolution Loop

Data: I_{init} a set of domain specific, constrained initial networks

```
1
2  begin
3      P ⟵ I_init
4      if sizeOf(P) < popSize then
5          P ⟵ P ∪ completeInitialPopulation(popSize − sizeOf(P))
6      repeat
7          foreach individual p_i ∈ P do
8              Evaluate(p_i)
9          P_best = keepBestIndividuals(P)
10         P_new = CreateOffspring(P, popSize − sizeOf(P_best))
11         foreach individual pn_i ∈ P_new do
12             VariationChain(pn_i)
13         P ⟵ P_best ∪ P_new
14     until sufficient solution found
```

3.3.1 Offspring Generation and Variation Chain

Initial Generation. The population of the first generation is created based on a number of predefined initial networks. These networks have to be prepared manually by the user (see section 5.1) and usually provide – in addition to the basic network structures – an elaborated constraint mask , that defines the search space of the experiment. Alternatively, the initial generation may also be composed of the individuals of a previous evolution experiment, which is a common case when performing iterative evolution experiments (see section 5.3). If the number of the given initial networks is lower than the desired size of the initial generation, then that generation is filled up with slightly varied versions of these initial networks. For this, the affected networks are passed on to the variation chain even in the first generation, so that these networks are slightly mutated. Accordingly, the initial generation usually contains both, the unchanged initial networks and variations of these networks.

Selection. In all further generations, the individuals of a new generation are determined by a separate *selection method*. A small fraction of the new generation is comprised of the best individuals of the previous generation (elitist selection). These individuals are not mutated and pass on to the next generation unchanged. The number of preserved individuals can be adjusted as a parameter of the algorithm. Elitist selection can prevent the loss of good solutions by inauspicious mutations, especially when the mutation rates are very high. In such cases, it can happen, that all new individuals are worse than their parents and good (partial) solutions are lost, so keeping the best individuals helps to carry the best individuals through the generations independently of the mutation success. However, elitist selection is not a guarantee to preserve already good working approaches, especially in randomized evaluations with a low number of tries per individual, where the best individuals can represent local optima of a specific evaluation instead of the global problem. Consequently, the settings of the elitist selection have to be chosen problem dependent.

All other individuals of the generation are discarded and replaced by newly created ones. For this, the selection method chooses individuals of the previous generation as

Algorithm 2: VariationChain(P)

```
   Data: P a set of individuals
   Data: M an ordered list of mutation operators
 1 begin
 2     foreach individual p_i ∈ P do
 3         if hasNoGenome(p_i) then
 4             setGenome(p_i, CloneFirstParent(p_i))
 5         if hasTwoParents(p_i) then
 6             ModularCrossover(p_i)
 7         n ⟵ maxIterations
 8         repeat
 9             valid ⟵ true
10             foreach mutation operator m_j ∈ M do
11                 executeOperator(m_j, p_i)
12                 if isNotValid(m_j, p_i) then
13                     valid ⟵ false
14             execute ConstraintResolver(p_i)
15             if ConstraintResolver failed then
16                 valid ⟵ false
17             n ⟵ n − 1
18         until valid = true or n = 0
19         if valid = false then
20             remove p_i from P
```

parents for each new individual. The underlying selection strategy hereby depends on the used selection method. Usually, individuals with a higher fitness have a higher probability to be chosen as a parent. The details of the selection strategies, however, are manifold (for examples, see Hancock 1994; Jong 2006). ICONE does not restrict the used selection method, so that the user can choose the most appropriate strategy for each experiment.

Variation Chain. After the selection phase, the variation chain (Algorithm 2) is applied to any new individual of the new generation to create and vary their genome. The variation chain comprises an ordered sequence of evolution operators and filters, that are specifically implemented for a given genome encoding. Each individual of a generation is passed through that operator chain, allowing all operators to apply mutations and variations to its genome.

Genome Rejection. As a unique feature of ICONE, evolution operators can also *reject* a genome, i.e. operators may indicate that an individual should *not* be evaluated with its current genome configuration. This is useful to realize filters that prevent costly evaluations of networks known to be faulty or to violate requirements, for instance when essential sensors are not properly connected so that a proper behavior is impossible. Such a prevention of superfluous evaluations is especially useful when the evaluation of individuals is computationally expensive, as is true for most non-trivial animat experiments.

To allow further mutations after a rejection and herewith a potential correction of the detected flaw, the execution of the variation chain is repeated as long as at least one operator still rejects the genome. To avoid infinite loops, operators can only be executed a limited number of times per individual, which can be adjusted by the user depending on the problem. If the maximal number of executions for an individual is exceeded and there is still an operator rejecting the genome, then that individual is removed from the generation. Therefore, the choice of this setting is always a compromise between worst case performance and resolvability of the filters and constraints.

Evolution Operators. The evolution operators of the mutation chain are genome specific functions, which can mutate arbitrary attributes of the genomes, such as the insertion of network elements or variations to synaptic weights. Such operators can be quite complex, for instance implementing specific insertion heuristics for synapses. Because of their importance and specialization, the set of used evolution operators can be chosen freely for each evolution experiment. Appendix B gives an overview on the mutation operators used with the NERD Toolkit implementation of the ICONE method.

Among the evolution operators of the variation chain, three operators are special: the genome *clone* operator, the *modular crossover* operator and the *constraint resolver*. Although these operators are – in principle – not different than other evolution operators of the variation chain, they are separately shown in algorithm 2 to underline their important role for the ICONE method.

The *clone* operator is required to create a genome for any new individual by cloning the genome of its first parent. The subsequent *modular crossover* operator exchanges parts

of the individual's cloned genome by genetic material of a second parent (section 3.3.3). This, of course, only works with selection methods that provide more than one parent per individual. After this, all further mutation operators and filters are executed in a fixed, adjustable order. The final operator of the variation chain is the *constraint resolver* (section 3.3.2), which triggers the execution of all constraints of the mutated individual to readjust the genome to fit the given constraints. If this is not successful, then the constraint resolver rejects the genome. Such networks are then further mutated with repeated applications of the variation chain until they are either matching the constraints, or are – after the maximal number of mutation attempts – removed from the generation.

3.3.2 Constraint Resolver

The constraint resolver is a relatively simple operator required to apply the functional constraints of a genome after all other mutations have taken place. Therefore, this operator is usually the last one in the variation chain. The resolver operator simply executes all constraints of a network one by one in a non-deterministic order. The constraint functions hereby verify the compliance of the required genome properties and – if necessary – adjust the network to repair detected constraint violations. If after the execution of all constraints one or more of them had to do any modification to the genome (or was not satisfied with the current state of the genome), then the constraint resolver is rerun. This is important, because when modifying a genome, a constraint may violate other constraints, that may already have been satisfied. These other constraints then require an additional run to correct the new constraint violations again. Hence, structure changes often have to be

Algorithm 3: ConstraintResolver(p)

```
   Data: p a single individual
   Data: C the set of all constraints used in the network of individual p
 1 begin
 2     n ⟵ maxIterations
 3     repeat
 4         modified ⟵ false
 5         foreach constraint c_i ∈ C do
 6             applyConstraint(c_i)
 7             if c_i made changes to the network then
 8                 modified ⟵ true
 9         n ⟵ n − 1
10     until modified = false or n = 0
11     if modified then
12         report resolver failed
13     else
14         report success
```

propagated through the network over several iterations before all constraints are satisfied. This *resolved state* is reached when all constraints could be executed without any of them having to change the genome.

The advantage of this algorithm is that its implementation is very simple and does not require a complicated backtracking or coordination (for classical approaches see Coello 2002; Dechter 2003). Also, functional constraints are easy to implement for such a resolver, because each constraint can be implemented as an independent entity. This matters if (scripted) constraints should also be implemented by users with limited programming skills. On the other hand, such a system can lead to infinite loops when a network comprises conflicting constraints. For that reason, the constraint resolver has an adjustable maximal number of repetitions per run, that has to be chosen as a compromise between worst case performance and resolvability of complicated, stacked constraints. In practice this number can be kept low (< 10) in most cases, which does not stress performance much. If the resolver has to rerun more often than the chosen maximal number of runs, then the genome is considered to be incompatible with the constraints and thus is rejected. As described in the previous section, a rejected genome is passed to the variation chain again, so that the network gets further mutated and possibly can finally reach a resolved state.

With this approach, genomes with violated constraints cannot evolve, because they are either 'repaired' by further runs of the variation chain, or removed from the generation if even additional mutations did not lead to a resolved state. This guarantees that all evaluated individuals are fully compatible with the chosen constraint mask.

3.3.3 Inheritance and Modular Crossover

The genome of a new individual is created by cloning the genome of its first parent. Without a crossover operator, this genome is then mutated to obtain a variation of the parent network. With this widely used technique the solution of an experiment is approached in many iterative, consecutive steps. An obvious disadvantage of this is, that there is only a transfer of information from one parent to its direct children and thus only separate lines of development in the population. Specifically, this means that any partial solution to a problem has to be found independently by each direct lineage. A direct exchange of different approaches or partial solutions between individuals of a population is herewith not possible. For small networks with a single task or a low number of subtasks this is not so much of a problem, but when we try to evolve larger networks comprising necessarily the interaction of functional subnetworks and the combination of subtasks, this is a limiting factor for evolution. Consequently, in the domain of mid-scale networks, the exchange of information between different lineages is considered beneficial, and can be realized with a crossover operator.

Crossover. In the field of genetic algorithms crossover denotes the combination of genetic material of two parents when a new child is created. This is inspired by genetics where this happens during the reproduction of bi-sexual animals. Hereby, during the meiosis, the compatible female and male chromosomes exchange equally long parts, usually longer continuous segments. In genetic algorithms, this mechanism is adopted and used with

varieties of different combination approaches. The most prominent ones (Mitchell 1996) are the one (two, multiple) point crossover, where the chromosomes are exchanged at one (two, multiple), in both chromosomes similar position(s), and the bitwise crossover, where each bit of the chromosome is either exchanged or not. In the field of neuro-evolution, crossover has been used only in a few methods, for instance by exchanging sub-matrices in matrix representations of neural networks. However, most neuro-evolution algorithms ignore crossover for a simple reason: the uncoordinated, random exchange of genetic material does – in case of neural networks – mostly produce networks performing much worse than their parents, because the exchanged attributes often alter the entire network dynamics. Evolution therefore does not perform well, because the created networks do not work similar enough like their parents to benefit from the local search. The larger and more diverse the parent networks are, the more severe this problem is. This is why this simple crossover can often be used successfully with small and topologically identical networks, but fails with larger, topologically heterogeneous networks.

Modular Crossover. ICONE approaches this problem by making use of the modularized networks (section 5.1) and the herewith used neuro-modules. Neuro-modules are already used to structure and constrain the network. Since the network's modules represent very specific network areas, often already comprising a specific function or organizational role, they are ideal candidates for the exchange of genetic material. This approach follows the heuristics, that compatible modules should have similar function in both parents, which makes them more likely to be exchangeable.

Conditions for Modular Crossover. To exchange two modules in ICONE, both modules have to be compatible. This means, that they have a similar neural interface (input and output neurons) and have a compatible type. The neural interface is important, because during the exchange, all connections to and from the replaced module are kept intact and are rerouted to the new module, which is only possible when the new module provides

Algorithm 4: Modular Crossover(p)

```
   Data: p a single individual
   Data: N_1 the neural network of individual p
   Data: N_2 the neural network of the second parent of p
1  begin
2      M1 ⟵ all unprotected modules of N_1
3      M2 ⟵ all unprotected modules of N_2
4      foreach module m_i ∈ M1 do
5          if rand() < crossoverProbability then
6              M_match ⟵ all modules of M2 matching type and interface of m_i
7              m_flip ⟵ randomly chosen module from M_match
8              replace m_i in M1 with a copy of m_flip
```

the same interface as the replaced one. The type compatibility is defined via property tags (section 3.2.4). A special tag contains a list of types the module belongs to, and another tag defines a list of types the module is compatible with. Both lists can be defined freely by the user. A module A is compatible for a replacement by module B, if its compatibility list contains a type name found in the type list of module B.

Optionally, modules can have a role tag, that defines the role(s) of the specific *place* a module is located at. If present, module B in our example must have a type name listed in the role tag of module A to be compatible. If a replacement takes place, then the role tag is copied to the new module and removed from the replaced one, because it belongs to the specific place the module is used at. Using a role tag avoids *type drifts*, which may occur after several successive replacements with modules having very different compatibility lists. In the course of such successive replacements previously incompatible modules can become valid for a module location. The role tag that always remains static for a specific module location, hereby guarantees that the specific module location is occupied only by modules of types compatible with that role tag.

Selection and Exchange of Modules during Crossover. In the crossover phase, the crossover operator decides for each new individual randomly whether crossover takes place or not. If so, the operator decides for each neuro-module at random, whether it should be exchanged with a compatible module of the individual's second parent. The module exchange probability can be chosen globally by the user, but can also be locally overwritten via property tags at the module level. Therefore, neuro-modules can have different probabilities of being exchanged, which allows a fine control of exchange preferences. Once a module is chosen for exchange, a compatible partner from the second parent is selected according to the described compatibility rules. From all matching candidates, one is randomly selected. This new module now replaces the old module with all of its submodules, keeping its position in the network and module hierarchy intact. All synapses connected to the old module are hereby rewired to the new module.

Limitations. Modular crossover only works in properly modularized networks. Crossover without an experiment-specific definition of compatibility tags is not recommended and will not lead to better results than random crossover. Therefore, all networks have to be prepared properly by the user. The better the modules are typed and organized, the more crossover may enhance evolution.

In some cases, crossover can destroy or remove constraints, e.g. when replacing constrained modules with unconstrained ones. This should be avoided by the user by proper settings of compatibility lists and roles. Also, conflicts of constraints may be induced, for instance when incompatible constraints are exchanged together with their modules. However, the worst that may happen in this case is that the constraints of the genome cannot be resolved any more, which leads to a removal of this genome. As a consequence, faulty networks still cannot evolve.

Advantages. The main advantage is obviously, that crossover is possible at all, which leads to more diversity and the possibility to exchange partial solutions between different lineages. This crossover strategy may not be optimal, because of its need of supervision and preparation, but it is still better than no crossover at all.

The network preparation is quite simple using network tags. It often can be done with little extra effort along with the network modularization (section 5.1), that has to be done anyway. With the compatibility lists, modules can also be excluded from crossover, if an exchange of specific modules is not desired. Crossover therefore can be limited to such modules where an exchange may potentially be useful, once again allowing the application of domain knowledge to the evolution.

Ludwig (2011) has shown in a first evaluation of the modular crossover operator that it can enhance the efficiency of the evolution for suited experiments – especially those comprising independent functional modules – and that the performance never suffered from the use of modular crossover. As a result, it is recommended to enable modular crossover when a modularized network provides modules suitable for an exchange.

Exchange of Modules from a Module Library. With little modifications, modular crossover can also be used to allow the exchange of modules between a network and separate module pools, instead of a second parent. This allows the exploration of compatible modules from a predefined module-library (section 5.1.3) or the co-evolution of neuro-controllers and separate populations of specific types of neuro-modules.

Chapter 4

Functional Constraints

This chapter describes the main functional constraints that are implemented and tested for the ICONE method. The understanding of these constraints is important to follow the experiment descriptions in sections 7 and 8. The constraints described in this chapter are only a subset of the realizable functions. They form a basic set of standard constraints that are useful for many experiments. Additional constraints, however, can also be added when needed to fit more specific needs.

Constraint Masks. Functional constraints are the main search space restriction mechanism of ICONE. Such functional constraints allow the definition of constraint masks for the evolving networks to enforce the compliance of specific topological and parametric demands. With a proper constraint mask the parameter space of the evolutionary search can be significantly reduced and the search can be biased towards desired solution approaches. Furthermore, the evolving networks can be forced to evolve within limitations given by the desired target systems, for instance by the network capabilities of a target robot. The network size, layout, general organization, neuron distribution and neuron model can be accordingly constrained so that only networks evolve, that are actually compatible with the target system and therefore can be transferred.

Active and Passive Constraints. Constraints can be roughly separated into *active* and *passive* constraints. Passive constraints are very easy to implement, because they only check for certain violated conditions. If these conditions are not met, then such a constraint simply reports a failure, without trying to adapt the network to fit its need. In contrast, active constraints can also actively modify the network to make it compatible with the constraint again. Passive constraints should only rarely be used to express difficult limits that cannot be resolved in an automatic, reasonable way. The main problem with passive constraints is that a detected constraint violation can only be resolved by suitable additional random mutations of the variation chain, which is in many cases un-

likely. Consequently, networks violating a passive constraint are very often removed from the generation. This implies, that passive constraints should be carefully used, because if such constraints are easily violated, many individuals of a generation are removed, which may lead to a significant loss of diversity. Active constraints are preferred to be used wherever possible, because they can solve a constraint violation instantly without the need of further mutations.

Dynamic and Static Constraints. Another distinction can be made between *dynamic* and *static* constraints. Static constraints are constraints that do not affect the function of the network during evaluation. These constraints are the default in ICONE and are exclusively used in the experiments presented in this thesis. However, dynamic constraints are possible (and implemented) as an interesting future feature, for instance to add additional search heuristics to the networks. Active constraints are executed as part of the neuro-controller during its evaluation and can alter the network at runtime. This allows many new scenarios, for example the application of different kinds of neural learning algorithms on a module, or the collection of information about the network activations during the evaluation, which both can be used as a base for heuristics to change the genome later during the mutation phase. Like passive constraints, active constraints are not essential for the function of the finally evolved network and as a result can be removed safely with the rest of the constraint mask, leaving a fully functional network. Such networks therefore can still be exported to target systems, that only support the standard network models, which is important for many robotic systems.

Most constraints in ICONE should be implemented natively in the programming language of the evolution program, because the execution of constraints requires access to all parts of the genome and a high performance. Because the constraint resolver may execute a single constraint multiple times per individual, the constraint functions should be suitably fast to prevent a time-consuming mutation phase. However, many custom constraints cannot be defined in advance, because they are very specifically designed for a single, sometimes unique purpose. Therefore, the definition of constraints via (often slower) script or plug-in should be possible to increase the utility for the user and to enhance the expressiveness of the constraints.

In the following, all constraints used for the experiments of this thesis are briefly explained. All constraints have been implemented as part of the NERD Toolkit (see appendix D). In the experiment descriptions in the application sections (chapters 7 and 8), only the names of the used constraints are given to avoid duplications of descriptions.

4.1 Cloning

TargetId	The ID of the master module or group that is to be cloned
Mode	The execution mode
References	A list of ID pairs describing the reference pairs between the neurons of the master module and the cloned module

Table 4.1: Parameters of the **cloning** constraint

The clone constraint can be used to replicate a network structure in different places of the network. The neurons and synapses of the affected neuro-module or neuron group are restructured to fit the cloned neuro-module or neuron group.

The cloned master group is specified with the `TargetId` parameter. The cloning behavior can be influenced with the `Mode` parameter: Different settings lead to an identical copy of the master group, to a copy of the structure only (not controlling the weights), or to a copy with *slight* weight variations between the master and the copied group (*Offset Symmetry* constraint, section 4.4). Also, the sign of the copied weights can be reversed. Other modes control how incoming, outgoing and mutual synapses of the groups are treated. Each direction can independently be set to symmetric, reversed sign, variable sign, variation offset or unaffected.

The `References` parameter contains pairs of unique IDs of the member neurons of the groups, so that for each neuron of the master group there is a corresponding neuron in the copied group. This reference list usually is created and maintained automatically by the constraint. In some cases, for instance if motor and sensor neurons are part of the groups, some of the references have to be given manually to ensure the correct reference relation between the neurons.

The clone constraint is useful if the same network structure is expected to be used in different places of the network. This reduces the corresponding search space to a single prototype (master) group. The parameters of all cloned network elements are excluded from the search space, because their structure, bias terms and weights are not independently evolved any more, but instead are fully (or depending on the selected mode only partially) specified by the master group.

4.2 Symmetry

TargetId	The ID of the group this module / group is symmetric to
Mode	The execution mode
Layout	The axes of the symmetry
References	A list of ID pairs describing the reference pairs between the neurons of the master module and the symmetrized module

Table 4.2: Parameters of the **symmetry** constraint

The symmetry constraint is very similar to the clone constraint, with the major difference, that this constraint allows the specification of axes of symmetry. The symmetrized neuron group can have a horizontal axis, vertical axis or both axes of symmetry. These axes are defined with the `Layout` parameter.

Like the clone constraint, the symmetry constraint often reduces the search space significantly, especially when the target animat has a symmetric motor-sensor layout and the evolved behavior is to be expected to be symmetric as well (walking, squatting, pointing, etc.). With the symmetry constraint, such networks often require only half of the usual search space during evolution.

4.3 Connection Symmetry

TargetId	The ID of the group the connections of this group are symmetric to
Mode	The execution mode
References	A list of ID pairs describing the reference pairs between the neurons of the target and the dependent group

Table 4.3: Parameters of the **connection symmetry** constraint

Connection symmetry is a symmetry constraint that only affects the synapses between the member neurons of the affected neuron-groups. The involved groups hereby may overlap, so that a neuron may be part of both the master group and the affected group. This overlap allows for interesting configurations, for instance to realize motifs (Bullmore and Sporns 2009; Wang and Chen 2003) in a grid network or to realize overlapping connectivity structures in multi-segment animats.

The parameters are similar to the other symmetry constraints, apart from the fact, that the `References` property accepts reference pairs where the same neuron ID is used both in the role of the source and that of the target.

4.4 Offset Symmetry

Offset Limit	The maximal offset a weight or bias term may diverge from its master element
Excluded Type	Exclude either neurons or synapses from the effects

Table 4.4: Parameters of the **offset symmetry** constraint

Offset symmetry is a variation of cloning, symmetry and connection symmetry and can be activated as part of these constraints. With this addition, the structure of the master module is still enforced on the affected neuron group, but the weights and bias terms

are not simply copied. Instead, the weights and bias terms of the affected synapses and neurons can be mutated freely with the standard mutation operators. However, if the weights or bias terms of the master module are mutated, then all neurons affected by the offset symmetry are adapted as well. The bias or weight is increased or decreased by the same amount the reference neuron or synapse was changed. This strategy has similarities to delta encoding (Gomez and Miikkulainen 1997; Whitley et al. 1991) and comes in handy, when a module should be cloned or be symmetric, but requires a slight local adaptation. The local offsets should not be too large (depending on the application), because if the weight deviations become too large, then the probability increases, that changes of the master module destroy the functionality of at least one of the dependent modules. In this case the performance drops down and these mutations do not get a chance to prevail during evolution. Therefore, this constraint provides a parameter to limit the offset to a fixed amount or to a proportion of the master element. This keeps the variations small to ensure that the cloned modules remain similar to their master modules.

4.5 Network Equations

Scope	The scope of the equation constraint

Table 4.5: Parameters of the **network equations** constraint

The network equation constraint can be used to calculate neuron bias terms and synaptic weights depending on other bias terms and weights in the neuron-group. For this, property tags can be added to neurons and synapses to define variables and equations. The `variable` tag defines a variable name that refers to the current value of the bias term or the weight of the tagged neuron or synapse. These variable names can be used as part of equations to calculate the bias term or synaptic weight of other neurons and synapses by tagging these network elements with an `equation` tag. Such a tag defines an equation that may use any arithmetic expression, including all variable names defined with the `variable` tag that are visible in the scope of the constraint.

The scope is defined by the `Scope` parameter of the network equation constraint. By default the scope includes only neurons and synapses of the same neuron-group, module or of its submodules. The scope however can be influenced, e.g. to exclude all submodules of a module or to exclude only those submodules that provide their own network equation constraint. As a result, it is possible to define multiple, independent modules with network equation constraints, that do not affect each other, even if the defined variable names are identical within their scope. This allows multiple instances of the same module equipped with equation constraints (e.g. from the module library) to work independently of each other during evolution.

The network equation constraint is a powerful way to define all kinds of relations between synaptic weights and bias terms. Hereby, all affected synapses and neurons become dependent on other synapses and neurons and thus are not freely evolvable parameters of the search space any more.

4.6 Prevent Connections

This constraint prevents all connections between the neurons of a group. This constraint can be used to avoid known ineffective connections between individual neurons. A variation of this constraint can also be used to avoid only self-couplings for the member neurons.

4.7 Enforce Directed Path

TargetGroup	The group containing the neurons to connect the own member neurons to
PathLength	A range specifying the minimal and maximal allowed path length
ModuleTransitions	A range specifying the minimal and maximal number of modules that are crossed on the path
Mode	The execution mode
References	An optional list of neuron pairs that have to be connected

Table 4.6: Parameters of the **enforce directed path** constraint

In many cases domain knowledge demands that there should be certain directed connection paths between specific neurons to have any chance to solve the control problem. For instance, if the sensors in a network are not influencing the motors, then a sensor-driven behavior simply is not possible. Or if an experiment is designed to prove that certain subnetworks can work together, then these subnetworks should be connected, because otherwise, the question underlying the experiment is not really addressed. Evaluating such networks would most likely be a waste of resources and, worse, often leads to local optima that prevent a further development towards a more desired connectivity structure. In both cases, giving adequate connections manually in advance often biases the outcome of the experiment too much and hence is often avoided.

To solve this problem, this constraint can be used to enforce directed paths between neurons. In this context a directed path means any directed connection chain from a source neuron to a target neuron, allowing any number of neurons in between. The minimal and maximal path length and the number of traversed modules can be specified separately, for instance to avoid too short directed connections, that may lead to trivial solutions only. This constraint is almost passive, i.e. the network is not directly modified by the constraint. However, the constraint leaves control tags as hints for the mutation operators on the affected neurons, that increase the probability for these operators – for instance the synapse insertion or removal operators – to do the proper changes during the next execution of the variation chain.

The constraint allows for several execution modes. In the default setting, any neuron of the affected neuron group requires at least one directed connection to any neuron of the target group, without further restrictions. In many cases it also makes sense to specify the desired neuron pairs for a connection, which can be done using the `References` parameter. This parameter contains a list of ID pairs that have to be connected, the first ID belonging

to a neuron of the source group and the second ID belonging to a neuron of the target group.

4.8 Restrict Number of Neurons

Maximum	The maximum number of neurons in this module
Minimum	The minimal number of neurons in this module

Table 4.7: Parameters of the **restrict number of neurons** constraint

Often it makes sense to restrict the number of neurons in a module to avoid too complicated and incomprehensible modules. Furthermore, size restrictions are sometimes mandatory because the target system (e.g. the robot hardware) has size limitations for the networks running on the controller boards. In these cases this constraint can be used to limit the number of neurons for a module. Also, the minimal size of a module can be adjusted to avoid trivial modules without any potential function. This constraint is an active constraint that adds missing neurons and removes the most recently added neurons if the size limitations are exceeded.

4.9 Enforce Connectivity Pattern

Pattern	The connectivity pattern to be enforced
Parameters	Optional parameters of the selected connectivity pattern

Table 4.8: Parameters of the **enforce connectivity pattern** constraint

This constraint can be seen as a connectivity pattern generator that creates, maintains and repairs a certain standard connectivity pattern on the members of a group. Patterns include feed-forward structures, chains (with or without self-connections), layered network structures with fixed layer sizes, or fully connected subnetworks. When neurons or synapses are inserted in or removed from a module having this constraint, the corresponding connectivity pattern is restored, integrating the new network elements or fixing the occurred gaps in the pattern. The modules therefore are still open for mutations concerning their size and neuron distribution, whereas the desired pattern always remains valid. The constraint can be extended by additional patterns when needed.

4.10 Connect Groups with Pattern

Target Group	The ID of the group to connect to
Pattern	The connectivity pattern to be enforced
Parameters	Optional parameters of the selected connectivity pattern

Table 4.9: Parameters of the **connect groups with pattern** constraint

This constraint is quite similar to the *enforce connectivity pattern* constraint, with the difference, that the pattern is not maintained between the member neurons of a single group, but for all connections between two groups. Therefore, the ID of the target group has to be given as a parameter. The connectivity pattern is created only between visible neurons according to the module visibility rules, i.e. only between interface neurons of modules, if modules are involved. The patterns also only create direct connection patterns without inter-neurons between the neurons of both groups. Possible connectivity patterns are fully connected (unidirectional, bidirectional), single connections only (unidirectional, bidirectional) or arbitrary connections unidirectional. Additional patterns can be added on demand.

4.11 Restrict Weight and Bias Range

Mode	Indicates, whether neurons or synapses are restricted. For synapses, the constraint can be restricted to incoming, outgoing or group-internal synapses
Scope	A list of group IDs. Restricts the scope of the constraint to synapses to and from certain external groups only
Range	The valid range (min, max) for the weights or bias terms
Recursive	Applies constraint also to submodules

Table 4.10: Parameters of the **restrict weight and bias range** constraint

With this constraint, the valid ranges of synaptic weights and bias terms can be enforced for all members of a neuron-group. Such a constraint is useful to limit certain synapses to have only excitatory or inhibitory effects, or to allow only dampening (negative) bias terms in certain neurons. This active constraint automatically corrects violations. If possible, the new bias term or synaptic weight is calculated as

$$w_{new} = \begin{cases} 2min - w & \text{if } w < min \\ 2max - w & \text{if } w > max \\ w & \text{else} \end{cases} \tag{4.1}$$

If w_{new} is still out of range, then w_{new} simply is set to min (if $w < min$) or max (if $w > max$). With this strategy, it becomes less likely that the weights and bias terms get

saturated at the given minimum or maximum over time, so it helps to increase variations. The constraint supports different modes, that determine, which network elements are affected. The constraint can selectively affect the bias terms, the synapses of the member neurons of the constrained group, or also synapses going to and coming from external neurons. In the latter case, the affected synapses can be further limited with the `Scope` parameter, which holds a list of groups. All synapses that come from these groups (input mode) or go to these groups (output mode) are affected. The constraint only restricts network elements of the group itself, so in case of a module, submodules are treated like external modules. This behavior can be changed with the `Recursive` parameter to include all submodules.

4.12 Synchronize Network Tags

Network Tags	A list of parametrized network tags that are automatically added to each affected network element, or the ID of a prototype network element to synchronize with
RequiredTags	A list of parametrized network tags that have to be present at an element to include it to the list of affected elements
Mode	Indicates, whether neurons or synapses are affected. For synapses, the constraint can be restricted to incoming, outgoing or group-internal synapses
Scope	A list of group IDs. Restricts the scope of the constraint to certain external groups only
Recursive	Applies constraint also to submodules

Table 4.11: Parameters of the **synchronize network tags** constraint.

This constraint is useful to add network tags to newly inserted network elements, for instance to affect their mutation during evolution (see section 3.2.4) or to add tags required by other constraints to work properly, e.g. to add equations for the *Network Equations* constraint. The network tags are specified either manually as a list of parametrized tags, or by specifying a network element as prototype. In the latter case, all tags of the affected network elements are synchronized with this prototype, following the tag prefix rules described in appendix C. The affected network elelements can be further restricted by specifying a list of required network tags. If this list is present, then only network elements are synchronized, that provide the network tags in that list. The constraint supports different modes that determine, which network elements of the group are affected. The constraint can selectively affect the neurons, the synapses of the member neurons, or also synapses going to and coming from external neurons. In the latter case, the affected synapses can be further limited with the `Scope` parameter, which holds a list of groups. All synapses that come from such groups (input mode) or go to such groups (output mode) are affected. The constraint only restricts network elements of the group itself, so in case

of a module, submodules are treated like external modules. This behavior can be changed with the `Recursive` parameter to include all submodules.

4.13 Connectivity Density

Min Density	The minimal accepted density
Max Density	The maximal accepted density
Mode	Allows the application of density restriction to each neuron or on the whole group
Removal Policy	The removal policies include *newest*, *oldest*, *random*

Table 4.12: Parameters of the **connectivity density** constraint

Influencing the connectivity density of a group of neurons is a valuable feature. Using structure evolution, it is often difficult to get networks that are neither too sparsely, nor too densely connected. Highly connected networks are often difficult to understand and provide many, usually highly dependent degrees of freedom, whereas too sparse connections often simply do not provide the desired functionality. With the connectivity density constraint, it becomes much easier to choose the density of the entire network or of separate neuromodules or groups of neurons. The constraint provides parameters to choose the desired connectivity density, either as a proportion of all possible connections of the neuron group, or as a fixed number of synapses. The desired density is specified as a range [Min, Max], so that the density can be defined quite flexibly. The mode parameter chooses whether the connectivity density is enforced per neuron or for the whole group of neurons. The removal policy specifies how synapses are removed in case the connectivity has become too dense. Policies support the removal of the oldest, the newest or a random synapse. The constraint does not insert synapses itself, but rather leaves a temporary hint for the *insert synapse* mutation operator (appendix B) as network tag on the neuron group. The constraint then signals the mutation chain algorithm that the individual has to pass the variation chain again. The insert synapse mutation operator then adds the missing synapses according to the validity rules of module boundaries and synaptic pathways.

4.14 Custom Constraints

The standard constraints described in the previous sections cover large parts of the practical limitations needed for evolutions with ICONE. Though, in some cases very special constraints are needed, because without such constraints, an expressive constraint mask would be too complicated or even impossible to be constructed with a combination of standard constraints. Such *custom* constraints are therefore explicitly supported and should also be supported by implementations of the ICONE method, e.g. by allowing the definition of constraints as scripts or plug-ins.

Custom constraints are very useful, if specific regularities and dependencies should be described. In the following some examples of such constraints are described:

Neural Fields. Dynamic neural field theory (Coombes 2005; Spencer and Schöner 2006; Venkov 2009) is an approach to model working memories and task coordination with a biological vastly plausible approach. Neural fields – networks with a special connectivity pattern of short-range excitation and long-range inhibition – are used to sustain temporary events as prolonged activations in a field of neurons, preserving a (spatial) relation between these events. Neurons in such a field have a very specific connectivity pattern towards their neighbor neurons: at a short range, synapses are excitatory, getting weaker with increasing distance, finally turning to inhibition for longer range synapses. The maximal distance for such connections hereby is restricted, so that only a limited field area around a neuron is affected by the long-range inhibition. This distribution pattern, whose underlying function looks like a Mexican hat, allows multiple self-sustaining activation bumps. Such neural fields can be valuable for neuro-controllers, where a dynamic short-term memory is beneficial or where noisy, uncontinuous signals have to be stabilized. To use such networks with ICONE, a special custom constraint can be designed, that creates and maintains the neural field. As a result, new neurons inserted during evolution can directly be integrated into the field structure by adding adequate synapses with corresponding weights. Gaps resulting from removed neurons can automatically be repaired. And the parameters of the underlying weight distribution function (radii of the excitation and inhibition) can be derived from three reference synapses, that can be regularly mutated (but not removed) within limited ranges during evolution.

A neuro-module implementing a neural field cannot be described as a combination of standard constraints. But with a suitable custom constraint, the usage of neural fields becomes simple, because a module representing such a field with arbitrary size can be defined with a single constrained module. The advantage of using such a constraint compared to the predefinition of its structure as peripheral structure, is, that the size and properties of the field can still develop during the evolution, without the risk of destroying its function. This, in the first place, allows a proper focusing on experiments using neural fields.

SO^2 and Other Oscillators. Oscillations are an important neuro-dynamical feature that can be used in many contexts, for instance as pattern generators, as internal clock or for a behavior coordination. Oscillations (periodic and quasi-periodic) can be generated with many different structures and neuron models, each providing advantages and disadvantages with respect to size, reactivity, frequency or dynamical attributes. SO^2 oscillators (Pasemann et al. 2003a), for instance, are small and provide two phases of the oscillation (with a shift of 90 degrees), which is useful to shape a repetitive motion by combining the two phases. Limitations of that oscillator are the relatively high frequency and its inability to be adjusted dynamically. Consequently, the frequency and shape is determined by its weights, which have to be evolved to be changed. An unconstrained network easily loses its oscillation capability during evolution, because the oscillation requires specific ranges and relations of the weights to work properly. A custom constraint here can help to ensure, that no mutations of the neuro-module can destroy its oscillation capabilities, whereas the re-

sponsible synapses still can be altered to influence the frequency. Such custom constraints can be defined for most kinds of oscillators, so that it becomes easy to provide a whole set of specialized, mutable, but functionally stable oscillators in a module library to be used in forthcoming experiments.

Chapter 5

Neuro-Evolution with the ICONE Method

This chapter describes details on how to evolve neuro-controllers with the ICONE method. The primary focus is on the most essential requirement, the so-called *constrained modularization* of neural networks. In this preparation step constraints, limitations, heuristics and domain knowledge are combined to define a constraint mask as a base for the initial population of the evolution experiment. With this step, the search space is restricted and the search is biased towards specific classes of solutions. The chapter also addresses possible problems in that phase and how to cope with them. As part of the modularization, this chapter also describes the refinement of evolved neuro-modules, so that results of experiments can be reused and shared through a neuro-module library. The remainder of this chapter then describes the application of the ICONE method in iterative, interactive experiments.

General Practice. Evolving neuro-controllers with the ICONE method requires domain knowledge and user interaction to restrict the search spaces of the quite large evolving neural networks. This restriction is very difficult to achieve automatically and therefore has to be done manually. A typical evolution experiment with ICONE involves the following steps:

1. Plan experiment
2. Prepare the evaluation scenario(s) and the fitness function
3. Prepare the initial networks (with *constrained modularization*)
4. Test the initial networks for plausibility
5. Iteratively evolve the (partial) solutions to the problem in a usually interactive way

---

Phase 1: Planning of the Experiment. The first step of any evolution experiment is its overall planning: What should be shown and how can this be achieved? With ICONE this also requires the decision whether the experiment can and should be broken down into multiple, successive experiments and with this the order and scope of the single sub-experiments. For such successive, simpler experiments it is usually easier to successfully find solutions, because these experiments often comprise much smaller search spaces compared to a single-step search for the overall problem.

Also, by stepwise approaching the overall problem, the evolution 'path' and with this the final approach can be influenced better.

Another benefit is that all successful intermediate experiments can be used as fallback positions from which one can search systematically into different directions, especially if some initially planned experiments turn out to be unsuccessful. So instead of failing with the whole experiment, only a small step towards the desired goal fails and allows a reconsideration of the approached subsequent experiments.

Solutions to intermediate experiments are also interesting with respect to the search for variations to a problem, because they allow, starting from such intermediate solutions, to approach the problem with different varied successive experiments, leading to more diverse solutions to the overall problem.

Therefore, iterative evolution often simplifies the evolutionary search, gives more control over the evolved solution approach and increases the likeliness of successful varieties of solutions. Iterative evolution (section 5.3) as such can be performed with most evolution methods. However, with ICONE the single consecutive experiments do not only shape the evolution by choices of different experimental scenarios, but additionally by the definition of different constraint masks (network shaping, see section 5.1) and herewith of different, deliberately chosen subspaces of the overall search space. This strongly increases the effectiveness of the iterative approach.

Phase 2: Preparation of the Evaluation. Once the course of an experiment is roughly planned, the evaluation methods have to be created. In the context of neuro-control, this usually is the preparation of a simulator with proper scenarios. However, in some cases, for instance for the evolution of separate functional modules, also simple function-based evaluations will do, which may lead to a better evaluation performance compared to a computationally expensive physical simulation.

The evaluation scenarios should not be realized in detail before all required previous experiments have succeeded. Experience shows that during interactive, iterative evolution, the initial plan of the overall experiment often changes as a result of the observed progress of the evolution experiments. Often, undesired local optima and unforeseen problems are only identified during the execution of an evolution experiment. As a result, experiments scheduled far ahead may become unnecessary or turn out to be unsuitable for the task long before they could be performed, superseding their design effort. As a result, the planning and the design of the evaluation scenarios should always allow flexible adaptations with short iterations instead of a fixed experiment plan.

Phase 3: Modularization of Initial Networks. The major measure with ICONE to control the search space for the evolution and to guide the evolution is the preparation of suitable constraint masks for the evolving neuro-controllers. In accordance with each evaluation scenario, a (set of) initial network(s) has to be prepared. These initial networks are used as initial seed for the population of the first generation and to restrict the search to a limited subspace of the search space. Therefore, it often makes sense to run a single evaluation scenario multiple times using different constraint masks. This allows the search for different controller variants within a single evaluation scenario, which reduces the effort for the experiment design and often leads to different intermediate alternative solutions, of which the best can be chosen for the subsequent experiments. The experiment in chapter 8 is such an example, where numerous experiments have been conducted within a single evaluation scenario, but with very different constraint masks, which lead to a large number of partially very distinct solutions to the approached problem.

Because of its importance to reduce the search space, to bias the search towards a specific solution approach and to bootstrap the evolutionary search, the preparation of the initial networks with their constraint masks should be done particularly thorough. This preparation is called *constrained modularization* and is described in detail in section 5.1.

Phase 4: Testing of Initial Networks. Modularized networks are often equipped with numerous, interweaved functional constraints, which define the constraint mask for the entire experiment. Because of this exclusive selection of the search space, it is particularly important to ensure that the resulting search space really is the desired one. The larger the initial networks are and the more constraints are involved, the easier it becomes to make mistakes and accordingly to focus on the wrong parts of the search space. This may accidentally prevent the evolution of a desired approach, blow up the search space by including undesired network properties or prevent the creation of sufficiently diverse valid individuals due to constraint conflicts. Therefore initial networks should be tested for plausibility before starting any evolution (see section 5.1.1.6).

Phase 5: Interactive Evolution. Most evolution methods run evolution experiments in an unsupervised way. Hereby, the evolutions are started with a fixed set of parameters and the result of the experiment then is collected when the algorithm terminates. This approach is profitable especially when successful results are known to exist and are likely to be found within the search space. If the existence of such controllers is not known, then many of such experiments fail, so that the experiment has to be performed multiple times with slight variations. For the evolution of mid-scale networks in the field of neuro-robotics, where the evaluation of individuals is computationally expensive, this can significantly slow down evolution.

Therefore, ICONE facilitates the conduction of *interactive* evolution experiments (section 5.2). With such supervised evolution experiments, the user can guide evolution directly by adapting the evolution parameters, constraint masks and the evaluation at runtime. This allows not only a prompt reaction to unforeseen problems, but also to explore and experience the dynamics of the evolution for the particular experiment, so that the choice of good parameter settings for the evolution becomes easier. Furthermore, frequent local

optima and undesired developments can be identified rapidly and be avoided promptly with countermeasures.

For ICONE, bot approaches are considered beneficial. The interactive approach is useful in the early stages of an experiment, when the potential problems and the evolution dynamics are still unknown. Once, the first (partial) controllers have been found, the experience gained from the interactive experiments (parameter settings, countermeasures to undesired local optima, revised constraint masks) can be transferred to unsupervised experiments, that then can explore the identified promising search spaces automatically to find variations to the interactively found solutions.

Backtracking. All described steps do not necessarily have to be performed in this fixed order. As indicated by the suggested *rough* planning of the experiments, changes in the evaluation scenarios, the fitness function, the initial networks and the overall experiments often are necessary to react on the evolution progress. In many cases, the evolution and its (intermediate) results reveal unexpected problems or suggest new ways to approach a problem that may not have been considered before. This then requires a backtracking to previous steps to account for these new aspects.

Evolution Results. Iterative neuro-evolution with the ICONE method provides different kinds of results. One result, obviously, should be a solution to the overall problem, which was the target of the experiment. However, using iterative, constrained evolution usually provides additional results as byproducts. First of all, every partial solution, i.e. a solution to an intermediate experiment, is a result on each own, that may lead to new insights. Furthermore, with network shaping, it is much easier to systematically search for variations of solutions, which often helps to find much more, even unlikely to develop variations, compared to an unconstrained evolution. The identified controllers additionally are often easier to understand, because the evolved *focus structures* are – despite the size of the whole network – often comparably small and less interweaved. Therefore, it is also easier to identify the underlying principles of these structures and to isolate and reuse them. This reuse of developed network structures is facilitated with ICONE by deriving functional neuro-modules from these structures (section 5.1.2). Such neuro-modules can then be collected in a module library and be reused in future experiments. Consequently, such refined neuro-modules are especially valuable results of the experiments.

5.1 Constrained Modularization and Network Shaping

Working with mid-scale neuro-controllers involves large search spaces for evolutionary search algorithms. For that reason, the evolvable degrees of freedoms, i.e. the number and range of network parameters, have to be reduced to increase the chance of getting proper results. The larger the networks, the more important this search space restriction becomes. Hereby, the application of domain knowledge is required to limit the evolvable parameters in a suitable, problem dependent way. That is why the search space restriction in ICONE is done manually by the user. The term *constrained modularization* here denotes

the task of creating a suitable *constraint mask* for the initial networks. This constraint mask should exclude all superfluous and avoidable parameters and limit the search space towards explicitly chosen parameter domains, hereby intentionally biasing the evolutionary search. This process is also called *Network Shaping* because the networks are outlined and 'shaped' in very specific, experiment dependent ways.

The actually used constraints depend on the goals, preferences and experiences of the user. Highly constrained networks, that eliminate major parts of the search space, can lead to results in shorter time, because – if a solution is within the search space – it is more likely to be found. On the other hand, such highly constrained networks only allow a low variation of solution networks and therefore often already partly contain the (assumed) solution. Having fewer constraints on the network opens up evolution to more diverse, less predetermined solution approaches, that are, for instance, not so intuitive for the user. Such networks, however, easily comprise search spaces that are too large and prevent the development of proper results. So, the user has to carefully decide from case to case which and how many constraints are necessary to succeed evolving the desired neuro-controllers. Examples of different constraint masks are described in chapters 7 and 8.

5.1.1 Preparation of Initial Networks

The major step when setting up an evolution experiment with ICONE is the preparation of a suitably constrained initial network or of a set of such networks. This step has to be done carefully, because it strongly influences the evolution progress, the kind of evolvable solutions and the probability of getting a suitable result. Creating an initial network is equivalent to the induction of domain knowledge to the evolving population of networks. The domain knowledge is used to form a (functional) *constraint mask* on top of the initial network to force the adherence of certain, user specified criteria. In addition, forming an initial network also means to provide (*peripheral*) network structures in advance (see section 5.1.1.2). That way, evolution is relieved from evolving network structures that are already known and are assumed by the user to be needed in some places.

Put together, a detailed control of the search space and the possible resulting networks is achieved. The creation of initial networks involves the following steps:

5.1.1.1 Basic Modularization

The search space for neuro-controllers grows quadratic with every new neuron, because in an unconstrained network, every neuron can have connections to any other neuron in the network. So, keeping the number of neurons low, and therewith avoiding processing neurons, is a major strategy in many evolution approaches to keep the search spaces feasibly small. Though, the number of neurons can obviously not arbitrarily be reduced, because even if no processing neurons are involved, the animat still needs all motor and sensor neurons to connect the controller with the body. So, in the anticipated domain of mid-scale networks involving *non-trivial* animats with rich sets of motors and sensors, even minimal networks can already comprise a large number of possible synapses and therefore a large initial search space. Also, the more sensors and motors an animat has,

the more likely disruptive connections between unrelated sensor and motor neurons become. These connections are unlikely to contribute to a successful behavior or even hinder the development of efficient controllers. Such unrelated connections are in many experiments quite easy to predict by experience and reasoning. So by preventing such connections based on domain knowledge, the search space can be reduced to a subspace that is more likely to contain working controllers. One step to do this and to bias evolution towards local processing units is to structure the network with modules. This groups related motor and sensor neurons together and shields them from potentially harmful connections with unrelated network areas. This heuristic is often helpful, because related neurons are more likely to produce expedient structures together than unrelated ones. For example, letting the sensors of a finger of a humanoid robot connect to the motors of its knees, will in most experiments not make much sense and – if at all – have a disturbing effect. In difference, if that knee motor is connected to the sensors of the knee itself and the sensors of its close-by joints, it is much more likely to end up with a meaningful behavior. Such considerations are often not necessary when evolving small neuro-controllers for *simple* agents, because these animats are usually designed to provide only related sensor and motor neurons (e.g. inverted pendulum experiments, tropism behaviors with differential-drive robots).

In larger networks, however, a grouping into related sets of neurons is necessary and has to be done manually based on domain knowledge. This can be realized at the network level by separating the network into distinct neuro-modules. So, modularization can be used to group related neurons behind fixed interfaces to prevent undesired connections and to favor local, related connections.

A special case, where neuron grouping with neuro-modules also has a technical relevance, are target animats that require a certain topological network distribution, for instance if the network is executed on a distributed processing system with limitations of the connectivity and the neuron visibility. Such network distributions can be enforced with a suitable segmentation of the network into distinct neuro-modules that reflect the distribution of the network on the processing system, so that only compatible networks can evolve (for an example see chapter 7).

5.1.1.2 Preparation of Peripheral Structures

In the context of neurorobotics and neuro-cybernetics it is particularly interesting to investigate specific network properties, organizational principles or the interaction of functional network areas. In the majority of such experiments, large parts of the networks are just *peripheral structures*, that are – with respect to the examined properties – relatively uninteresting, yet not omissible. The hereby really interesting parts of the networks – the remaining, evolving network structures – are here called *focus structures*, because these structures are a (usually previously *unknown*) solution to the problem the experiment actually focuses on.

Peripheral structures are often needed, for instance as motor controllers, to derive high level information from sensors or to use known structures to generate oscillations, working memories or signal filters. In an experiment with a focus on a certain neural coordination hypothesis, for example, it usually does not make sense to let evolution – in addition

to the herewith interesting coordination structures – also search simultaneously for the required motor controllers and the coordinated functional structures. This is especially true if structures for these required functions are already known. And even if the required structures are not yet known, then it is often easier to develop these structures first in separate, simpler experiments. This is expected to be much more successful than evolving a large network with multiple unknown functional areas in a single experiment.

Giving peripheral structures in advance does, of course, not only limit the search space, but also the possible results. However, it should be kept in mind that the main goal of evolutions in the domain of mid-scale networks primarily is to find results at all. This often demands a narrow focus on a limited functionality of the network and to find limited solutions first. Then, the related search space can be further explored with variations of the experiment to find alternatives to the already identified solutions. Otherwise the search spaces remain too large and successful evolutions are entirely prevented, which is obviously worse than limitations of the evolvable networks. With ICONE, the desired variations of the networks can still be evolved, but this has to be done systematically by purposely focusing the search on very specific parts of the search space. In fact, intentionally focusing the search on slightly different search spaces even increases the number of evolvable variants, because different search space configurations often have different solutions to be the most likely to evolve.

So, all structures, that are not of primary interest for the evolution experiment, but are nevertheless needed for a proper function of the network (the *peripheral structures*), should be given in advance to achieve a number of advantages: Evolution does not have to reinvent already known structures, the search space is reduced significantly, the bootstrapping of the evolution experiment is sped up and the experiment is easier to be biased towards a desired solution approach.

Forcing evolution to reinvent already known structures never seems to be a good idea, except if new solutions to already solved problems are the target of the evolution experiment. In all other cases, letting evolution recreate such peripheral structures only blows up the search space and reduces the chance to solve the actual evolution problem. If variations in the peripheral structures are desired, then this can still be achieved, for instance by providing multiple initial networks with different peripheral structures, by letting evolution replace known structures by other known structures from a neuro-module library (section 5.1.3), or by allowing evolution to choose from predefined alternatives (section 6.3). Following the modularization approach, peripheral structure should also be encapsulated into separate modules. This enables the use of specific constraints on these structures and allows a fine control of the valid connections between the peripheral and the focus structures. Peripheral structures that are encapsulated into neuro-modules are also especially suitable for hierarchical architectures, where modules are placed as functional submodules within other modules to control the visibility of neurons and their connectivity.

Peripheral structures also include network elements that define the frame for the focus structures, for which the experiment is designed to evolve. Such framing structures often include 'empty' neuro-modules only equipped with fixed interface neurons, to control the connectivity of these new structures with the remaining peripheral structures and to influence the evolving structures via constraints.

5.1.1.3 Low-Level Constraints with Network Tags

The search space should be restricted by excluding all evolvable properties of the network that are already known or that should not change. This can be achieved by adding suitable network tags (section 3.2.4 and appendix C) to those network elements. Network tags allow for a detailed selection of evolvable parameters on each separate network element. In addition to these protection measures, network tags should be used to specify plausible ranges for selected synapse weights or bias terms, if such ranges are known or assumed in advance. This, for instance, can avoid an evolutionary search in parameter domains with too large weights for synapses that need to be small to contribute reasonably, or to limit synapses to be excitatory or inhibitory.

Along with such range limitations for network attributes, the global settings for evolution operators (probabilities, rate of change) can be overwritten locally with network tags, so that distinct network areas can have different mutation rates. This allows a fine-control of the mutations throughout the network. In larger networks, it is common, that some synapses are very sensitive to even small weight changes whereas other synapses require large changes to have an effect at all. This contradiction is difficult to match with a global setting of the mutation rates. A reduction of the global rate of change for all synapses would lead to the desired small changes in the sensitive regions, but would also only allow such small changes in areas that would need large mutations. So, whenever such discrepancies are known in advance, then this should be specified in the initial network with network tags, so that each region has its own suitable settings. A suitable ICONE implementation should allow the definition of such settings not only by plain numbers, but also as proportion of the global setting. With this, the settings can still be adapted during (interactive) evolution, whereas the required relative setting differences between such network areas is kept intact.

5.1.1.4 Preparation for Modular Crossover

The modules of the network also should be prepared to be compatible with the modular crossover operator (section 3.3.3). For this, the required network tags should be added to the modules that defines their type, role and compatibility lists. Properly configured, the exchange candidates for the modules become more reasonable and the crossover less destructive. The same tags are also useful to make the modules compatible with extensions, like module insertions from a module library (section 5.1.3).

5.1.1.5 High-Level Constraints

Functional constraints (section 3.2.5) are the major measures used to restrict the search space. One reason for this is that constraints affect many neurons and synapses at once in an often complex, but well defined way. Because of their potentially strong impact on the network elements of a neuro-module or neuron-group, many degrees of freedom of the network are reduced to a few remaining evolvable parameters. The resulting network structures, however, can be organized in quite complex ways (examples are shown in chapters 7 and 8). Therefore, the user should impress functional constraints to the initial network to

clearly outline the desired network topologies. Functional constraints are a flexible way to restrict the search space and the network topology without losing the potential for large mutations, network extensions and topological modifications. Based on domain knowledge, functional constraints can much better and less restrictively describe the desired network topologies than network tags and peripheral structures alone.

5.1.1.6 Testing

Mid-scale networks with many interacting functional constraints and network tags can become complex and sometimes also confusing. In such cases, it easily happens that the outlined network topology diverges from the one the user wanted to describe. Furthermore, poorly configured or wrongly combined constraints can lead to irresolvable constraint masks. Such initial networks never produce valid offspring, because irresolvable networks are removed from the population.

To avoid such problems, the preparation of initial networks should always include a testing phase. All constraints should be manually resolved to prove their resolvability and to check if the resulting networks are within the expected and desired set of topologies. For this, brief test-evolutions with high settings for insertions of neurons and synapses (and later with high settings for their removal) should be used to check if the maximal network (minimal network) still comprises the desired topological features.

5.1.2 Module Refinement

When using neuro-modules with given structures to create the peripheral structures for an experiment, these neuro-modules should have a high quality, be small and allow adaptive mutations without increasing the search space too much. Many of such modules are the results of isolated evolution experiments and consequently often are not optimal. Before such structures can be reused as building blocks they should be refined to significantly enhance their quality and usability. In the following the main measures to refine neuro-modules are described in detail.

Functional Separation and Minimization. Evolved structures are often interweaved and not neatly separated from each other. To use the structures as reusable building blocks, their functional network elements have to be isolated and separated from each other. Superfluous neurons and synapses should be removed to minimize the structure.

Generalization, Completion and Repair. Evolved structures are often not working perfectly under all circumstances. Therefore, it makes sense to examine the functional properties of a module and to create a generalized version that works properly in all application domains. Also, many structures can be further optimized according to size or performance, for instance by using *order dependent neurons* (see section 6.9) to reduce the processing time of a module.

Footprint Reduction. Because most refined neuro-modules provide a very specific function, most of their internal neurons and synapses are not of interest for other parts of the network. Connections with these neurons would either disturb the function of the module, or are unlikely to lead to a useful interaction. Therefore, only a few neurons are required to be accessible from outside of the module. These neurons should form its neural interface. As a result, adding a module with many internal neurons, but with a small neural interface, extends the possible synaptic connections only by these few interface neurons and therefore has a much smaller footprint than allowing connections to all internal neurons.

Search Space Minimization. Adding a fully mutable neuro-module with many neurons still can significantly increase the search space. Therefore, refined neuro-modules should use network tags to protect all attributes of all network elements that are not required to be mutated during evolution. Network tags can also be used to restrict certain synaptic weights or bias terms to plausible ranges, which avoids search in undesired parameter domains. In addition, functional constraints should be used to induce dependencies of network elements to further reduce the search space to the essential evolvable parameters only.

Functionality Protection. Mutations of functional neuro-modules are often required to allow an adaption of the module to the special context. But such mutations can easily destroy the function of the module. Therefore, refined modules should be equipped with functional constraints that protect the functionality of the module. Such constraints can be used to automatically repair the module if mutations destroy or violate the function or organization of the module.

Enhancements for Evolution. In refined neuro-modules the function of the module and the parameter domains of its network elements are usually known and well understood. For that reason, such networks can be tagged with suitable network tags to optimize the evolution on and with these modules. For instance, mutation probabilities for crucial parameters may be tagged to be more rough or more fine than for the rest of the network. Also, a proper setting of the module type and compatibility lists simplifies the use of the refined neuro-modules with modular crossover.

Refining neuro-modules pays off soon, because once refined, the modules can be reused in any forthcoming experiment. There, the initial networks get more compact, are easier to compose and provide a smaller search space. A well refined neuro-module can extend a network instantly with a fully working function composed of many network elements, but extend the overall search space only as much as the insertion of one or a few neurons would do. Another advantage is that anyone can reuse such a module, even without fully understanding its internals, so that persons with different backgrounds can work together and share results more easily. And finally, refined neuro-modules allow the use of a functional neuro-evolution with algorithms that compose networks from building blocks during evolution (see next section).

5.1.3 Neuro-Module Library

Refined neuro-modules should be collected in a *module library*. Such a library can be maintained by a work-group or community to collect new useful neural building blocks. With such a module library it becomes easier to build initial networks, because the peripheral structures can largely be composed from modules of the library. Most of the neuro-modules are already equipped with constraints, hints for evolution and proper search space reduction measures, so the user can fully concentrate on the constraints and limitations of the desired *focus structures*.

With future extensions of the algorithm, operators may also use such a module library to insert neuro-modules from the library to extend the networks with already working, functional subnetworks. Algorithms like that would allow evolutionary experiments similar to genetic programming (Banzhaf et al. 2001; Koza 1992, 1994; Koza et al. 1999), but with neural networks as functional elements. Another related algorithm extension is the initial insertion of compatible modules from the library as replacements for *placeholder* modules to increase the variations in the initial population. This also allows the spontaneous replacement of existing modules by compatible ones from the module library as a new type of mutation. Rapid changes of a neural structure are then possible without the need to convert a structure slowly into another by successive, random mutations, which seldom happens successfully during evolution.

The neuro-module library therefore is not only a valuable way to collect, refine, preserve, store and share neuro-modules, but also the basis for future enhancements of the evolution algorithm.

5.2 Interactive Evolution

Evolution with the ICONE method is not meant to be (only) an unsupervised evolution method, that – once prepared – is simply started and run until convergence. Although the method can be used in that manner, especially when used in combination with short, well defined iterations (section 5.3), the method is merely meant as an assistance tool to support the designer of a neuro-controller to find suitable results. This means that the method is used interactively while the network is supervised and guided by the user. At any point in time, evolution experiments can be altered, stopped and influenced to react to the current evolution progress. This includes adaptations of the parameters of the evolution operators, changes of the fitness function and modifications at the evaluation system.

Advantages. The advantage of interactive evolution is that the dynamics of the evolution, i.e. the impact of the evolution operators and their parameter settings, becomes more transparent to the user. This is important, because with the constraint masks on the evolving networks, the evolution can take much different routes in search space than expected or desired by the user. Interactive evolution allows an early estimation of whether the parameter settings and constraint masks are suitably configured, before time-consuming, long-running evolutions have been conducted. Experience suggests the strategy to run a new evolution experiment interactively for a while to find the proper settings, and then

reduce the interactivity (up to fully unsupervised evolution runs) over time. Experience collected from interactive evolution for a specific experiment can then also be transformed to control scripts that apply the found heuristics for parameter changes automatically in unsupervised evolution runs. Such an approach makes sense especially when a successful experiment is repeated multiple times with slight variations to search for network variants for an already solved problem. Such variation searches can then run on the computer equipment without supervision.

A second advantage of interactive evolution is, that necessary changes of the parameter settings during an evolution experiment can be induced in a highly flexible way at the 'correct' time, i.e. when the user recognizes that the search focus should be shifted to a slightly different area of the search space. Such readjustments are helpful, for instance to start with relaxed constraints and high mutation rates and then get to more specific constraints with lower mutation rates over time, or vice versa. Interactive adjustments can also help to detect and overcome observed local, undesired optima. Such interactive modifications can also give hints on how to change the experiment for the next run to avoid such a local optima right from the beginning, based on the strategies examined and refined during the interactive evolution.

Limitations of Interactive Evolution. Supervising an evolution experiment can be very time-consuming, especially if the evaluation involves the simulation of complex animats. Interactive evolution requires a quite fluent evolution progress so that the user can interact in real-time. If the evaluation of a generation requires many hours, then interactive evolution becomes difficult to do. Therefore, the evaluation method should be designed to allow a fast evaluation, if necessary by using a computer cluster. The evaluation phases of evolutionary algorithms are highly suited for parallel processing, which makes the implementation of a cluster evaluation quite simple.

Because the evolution progress of interactive evolution is strongly influenced by the guidance of the user, that user should have a good understanding of the evolution dynamics and experience in using constraints and evolution operator parameters. Consequently, the evolution success is very dependent on the user.

Interactions can also lead to performance decrease, because the user may lead the evolution badly. This can, for instance, be the case when constraints or evolution operator settings are poorly chosen, so that the desired results are simply not possible or very unlikely to occur. Also, interference by the user may prevent evolution from finding solutions, when undesired behavior is prematurely terminated or excluded via constraints, although that undesired behavior was a necessary step towards the desired behavior. Premature adaptations can easily prevent evolution from overcoming a local optimum on its own or to elaborate the networks over several generations. Therefore, user experience should be gained patiently and different intensities of interaction should be tested for each experiment.

Performing Interactive Evolution. The software for interactive evolution requires two main features: observation facilities and adjustment mechanisms. The evolution program should provide multiple ways to observe and analyze a running evolution, for instance vari-

ous statistics about fitness, network properties and the population diversity. Furthermore, it should be possible to view intermediate results, i.e. to display the networks graphically and to observe the resulting behaviors in a simulator. This observation should, for performance reasons, not be done for every single individual, but only for a representative subset of the individuals, for instance the best individual or the individuals with the highest number of offspring. For adjustments, the user should be able to work interactively with the parameter settings of the evolution operators, with the fitness function and preferably also with the evaluation method (e.g. the simulation scenario).

Interactive evolution then is the ongoing observation of the statistics developments, the examination of intermediate networks and, if appropriate, the adaption of parameters. In the reference implementation of the ICONE method (Appendix D) all interactive changes are, optionally extended by user comments, logged along with the evolution statistics and intermediate networks. With this, interactive evolutions become easier to analyze afterwards, to learn for future evolutions and to probe different paths through the search space. The latter can be achieved by restarting the evolution with different settings using any generation of a previous experiment as initial generation, for instance to try different settings for the same generation until the desired progress is achieved.

5.3 Iterative Evolution and Shaping

Evolving large networks comprising functionally complex subnetworks from scratch is – even with constraint masks – difficult to achieve, because the search space still can be very large. One way to overcome this problem is called *shaping* (Dorigo and Colombetti 1998; Gomez 2003; Karpov et al. 2011) or *task decomposition* (Hülse 2006; Kassahun et al. 2009; Perkins and Hayes 1997). The idea is to break the overall problem down into smaller, easier to evolve subtasks. This decomposition is done by the experimenter and requires domain knowledge on the problem. The subtasks then can be evolved separately with simpler experiments and correspondingly simpler, lower dimensional search spaces. In each iteration, subnetworks from previous experiments can be given as peripheral structures, so that the dimensionality of the evolved focus structures remains low. This approach is often called *iterative* or *incremental* evolution.

The approach has several advantages. First, the search spaces of the single experiments become smaller and the probability of success increases. Second, evolved subnetworks can be analyzed separately before they are used in forthcoming experiments. So, the overall analyzability of the networks increases, because even in larger networks combining several previously evolved subnetworks, most of the network structures are already understood. Evolved subnetworks can also be *refined* (section 5.1.2) before they are used as peripheral structures, which additionally reduces the search space and helps to keep the networks close to the desired topology. The module identification and refinement also is simplified by this approach, because it is much easier to rework a relatively small network with a single, known function, compared to the disassembling of a large network with many interweaved functions.

The evolution experiments also become simpler, including the evaluation scenario and the fitness function. This is, because a single task is easier to describe with a fitness function than a complex task with many subgoals. The fewer parameters are required for a fitness function, the smoother the fitness landscape can get, which increases the chance to find solutions in shorter time.

Iterative evolution also automatically provides *fallback positions* for the overall experiment. From each evolution step several distinct evolution attempts with different constraints and parameter settings can be started, always with the safe fallback to the previous experiments, in case the experiment did not succeed. So, each successful partial experiment already provides reusable results, even if the overall experiment finally fails.

Evolving controllers iteratively also has the advantage that the evolution can be biased strongly towards a very specific *development path*. Therefore, it is often beneficial also to break the evolution of a single-function network down into a sequence of consecutive experiments. This allows a fine control of the evolution focus for the same network over time. In fact, the user should plan an *evolution story*, starting with the evolution of required, and still unknown peripheral structures up to the final experiment. Such a *story* does not have to be fixed right from the beginning, so that each iteration can flexibly react on the results of the previous experiments. Intentional variations of the development paths can also lead to very different results, because each different partial experiment can result in new ideas on how to proceed from the currently evolved point.

Chapter 6

ICONE Extensions

The basic ICONE method, as described in the previous sections, can be extended by a number of useful additional algorithmic features, that allow a better adaption of the method to specific experiments. Such extensions can easily be implemented and added to the method. These algorithmic extensions should not be confound with custom constraints, that already allow a functional extension on the constraint mask level. Algorithmic extensions have a larger impact on the ICONE method itself and modify the basic algorithm by new strategies and features. Such extensions include additional evolution operators, heuristics for network element insertions, extensions of the neuron model and changes of the evaluation and selection phase, just to mention a few applications. This section briefly describes some of the ICONE extensions, that have been implemented for and used with the ICONE method. Note, that not all of these extensions are actually used for the experiments shown in this thesis. The thorough investigation of the effects of some of these extensions are considered future work.

6.1 Synaptic Pathways

In large or mid-scale networks, that comprise multiple distinct functional subnetworks, the network areas that should be interconnected are often known in advance. This also means that many possible interconnections are known to be superfluous or even harmful. The hierarchical structure of the modularized networks (section 5.1) already prevent many of such undesired connections by hiding local neurons behind well defined module interfaces. The interface neurons of the modules, however, can still be connected arbitrarily. This can be limited in more detail using the *synaptic pathway* extension.

Synaptic Pathways. Synaptic pathways are established by allowing or forbidding connections between certain modules explicitly. The direction of the synaptic connections hereby can be considered as well, so that connections from one module to another can be

allowed, whereas synaptic connections in the opposite direction are forbidden. Synaptic pathways can be described in different ways to make it convenient for the user. A simple way is to give the ids of modules explicitly to define lists of desired or forbidden module pairs. This, however, only works for fixed modules, that are not replaced or added to the network during evolution. A more flexible way is to specify such lists based on the network tags of the modules, such as the *type* tag or custom *permanent* tags (Appendix C).

The definition of synaptic pathways allows very clear descriptions of the desired network interconnections without the need to give all desired synaptic connections in advance. Evolution therefore still can search for the 'right' connections within the connection frame specified by the user.

Probabilistic Synaptic Pathways. A variant of the simple synaptic pathways is an extension that allows the definition of probabilities for connections between certain modules. With this extension, new synapses are not uniformly distributed between all modules within the pathway descriptions. Rather, the location of each new synapse depends on an additional probability that is attached to each approved module pair. A user can therefore express a preference for certain synaptic pathways and make other connections less likely, without preventing these connections completely.

6.2 Typed Connections

This extension is a way to further influence the interconnectivity of neurons and modules. The synaptic pathways extension (see above) is easy to use and allows the simple exclusion of certain inter-module connections. However, the valid connections between the specified modules can be arbitrary. If more control is required, then the typed connections extension may help.

This extension requires the definition of *type* tags on neurons and modules. Such type tags group network elements in type classes, that are specified by simple names. Then, the valid connections can be specified by lists of such class names, to which – or from which – connections are allowed. Such lists can be added as network tags to single neurons, but also to entire modules. This, especially in combination with the synaptic pathways extension, allows a very detailed control over the connections between neurons. As an example, a neuron input of a module may be restricted to get only inputs from neurons of type *sensor* or *processed sensor* to make sure that this special input receives a sensor signal, and not an arbitrary input such as a motor signal. Another input of the *same* module may be configured to get input only from special *control* outputs of specific modules.

As the example shows, the typed connections extension allows to add *meaning* to a neuron, i.e. what type of input it is supposed to process or what kind of output it produces.

6.3 Hidden Elements

Constraints can effectively limit the possible neurons and synapses evolution may add to a network and thus reduce the search space to a well defined subspace. However, the use of constraints may be too unspecific or complicated in some cases where – based on domain knowledge – most optional network variations are already known or assumed. This is useful, if only a few alternative configurations should be tested. To avoid to give all these optional network elements in advance and accordingly starting with a large and potentially functionally redundant initial network, one would like to specify all potentially new neurons and synapses in advance without forcing the network to use these elements right from the beginning. The hidden elements extension can achieve this.

Hidden Elements. The hidden elements extension allows the specification of a network with all of its neurons and synapses in its maximal dimension, and then to hide a subset of these network elements. These hidden elements are no longer functional parts of the network. However, during evolution a special operator can uncover hidden elements (or hide visible elements), with the same effect a simple insertion or removal of a neuron or synapse would have. The difference is only, that the uncovered elements are not randomly inserted, but have previously been placed and configured by the user. This allows an evolutionary search on given alternatives for the problem without considering other, more random options, that are not of interest for the user. Note, that hidden elements are fully compatible with constraints and the other search space restriction features of ICONE.

Hidden Sets. In addition to neurons and synapses, also entire modules or neuron groups can be marked to be considered as hidden elements. This leads to an activation and deactivation of (functional) subnetworks that are always uncovered or hidden together. This makes it easy to specify complex structural alternatives for a problem in advance.

Probabilistic Hidden Elements. A variant is the introduction of probabilities for each hidden element. That way, the likeliness to uncover a network element is different for each network element, so that elements assumed to be more important are uncovered with a higher probability than elements assumed to be less important.

Mixing Hidden Elements with Standard Operators. Hidden elements do not necessarily replace the standard operators for insertion and removal. Both types of operators can coexist. One way to do this is to apply constraints on selected modules to prevent the insertion and removal of network elements. Then, hidden elements can be defined in these modules. As result, these modules only allow the use of combinations of hidden elements, whereas in other parts of the network neurons and synapses are inserted and removed as usual.

6.4 Expiring Constraints

Interactive (and iterative) evolution has the advantage, that constraints can be changed and influenced over the course of an evolutionary search. Whenever it seems appropriate, constraints can be removed, added or adapted to give the network more or less evolvable degrees of freedom. Especially the stepwise increase of the search space over time seems to be useful. It allows a narrow search in the beginning to get roughly working solutions fast, and then stepwise allow more variations to optimize these rough solutions. Adding the many degrees of freedom required for an optimization of the controller already in the beginning often slows down the search for networks that do anything useful and thus prevent solutions at all.

In some special cases, this stepwise constraint release can be automatized, so that slowly extending search spaces can also be achieved in unsupervised experiments. For this, groups and modules can be *tagged* with a (probabilistic) *constraint release plan*. Such a plan can be realized with a simple list that specifies for each removable constraint, when it should be removed. The event for the removal of a constraint may be triggered when a specific generation is reached or a threshold of the fitness is exceeded. The release plan may also contain probability information, so that constraints, when the trigger event is reached, are not directly removed, but instead only increase their removal probability with the excess of the given thresholds. The decision of their removal is then done by an additional evolution operator.

Expiring constraints allow interesting experiments, for instance the stepwise, systematic break-down of symmetries, as shown with the ENSO method (Valsalam 2010; Valsalam and Miikkulainen 2009), or the stepwise *individualization* of previously cloned modules during the optimization phase of the evolution. Due to the diversity and extendability of constraints, this approach allows very complex and experiment-specific release scenarios.

6.5 Automatic Parameter Scheduling

Similar to the expiring constraints, this extension allows to automatically influence the evolution process. During the evolution, the 'ideal' settings of the evolution operators (mutation rates, population size, selection pressure) are not constant and depend on the current progress of the evolution. In the early generations, for instance, mutation rates are usually higher than in later phases, in which controllers are primarily optimized rather than constructed. Also, higher rates may be required occasionally to escape local optima.

In interactive evolution such adaptations of the evolution parameters are easy to do. In unsupervised experiments, in contrast, the experimenter has to decide for a single set of parameters that are used throughout the entire evolution experiment. This limitation can be overcome using this extension. The automatic parameter scheduling allows the definition of events at which settings of the evolution operators are changed. So, the evolution can be divided into different phases or states, in which different settings are used. The transitions between the states can be defined based on the generation count, the fitness development or the attributes of the evolving neural networks.

6.6 Mutation Plans for Individuals

The creation of the initial generation is of a high significance, because the first generation is used to probe the search space broadly to locate promising regions of the search space. This first generation therefore should be as various as possible. With an extension like the previously mentioned automatic parameter scheduling, the first generation can be created using exceptionally large mutation rates (especially for the topology manipulation operators) to promote the creation of a wide spectrum of initial network configurations. However, probabilistic mutations lead to a Gaussian distribution of the results, so most initial networks will be of a certain mean complexity, and only a low number of networks will be particularly small or large. In many highly constrained networks this is not desired, especially not in the first stages of a new experiment. In that phase, the ideal topology (network size and connectivity density) is usually not known and has to be explored with many separate experiments with different settings of the parameter settings. Here, it can save time when a non-Gaussian distribution is used, that covers the search space more uniformly and allows more variants of a desired kind in the first generation (or throughout the evolution). For this, the mutation plan extension can be used. Such a mutation plan describes multiple sets of mutation settings, that are alternated during the creation of the *same* (initial) generation. For each separate individual, one of these sets is chosen at random and used during its mutation phase. Individuals therefore will be created based on different mutation settings, which leads to a broader, but still fully definable distribution of the (initial) individuals.

6.7 Evolving Constrained Learning Rules

The evolution of static neuro-controllers, i.e. neuro-controllers whose synaptic weights and topology remain static during their evaluation, requires evolution to optimize all relevant synaptic weights in a series of many, consecutive mutations. Therefore, the number of generations needed to find and optimize a controller requires relatively many generations, because for these required changes accordingly many mutation steps are needed to converge to the desired behavior.

An alternative can be the evolution of neuro-controllers with learning abilities in form of adaptive synapses (Floreano and Urzelai 1999; Gruau and Whitley 1993; Stanley and Miikkulainen 2003b; Urzelai and Floreano 2001; Zahedi and Pasemann 2007). Such adaptive synapses are often based on learning rules that describe the strategy used to adapt the synapses during the network evaluation. The variety of possible learning rules is very wide, so that this extension is suitable to investigate learning rules for different types of experiments. The learning rules can be treated as simple properties of neurons, synapses and neuro-modules, so that heterogeneous networks can be evolved, mixing static synapses with different types of adaptive synapses. This makes evolution quite flexible and avoids the problem of having adaptive synapses in the entire network. Furthermore, the mutation and distribution of learning rules is fully compatible with constraint masks, so that the search space of networks using learning rules can be restricted like any other network. By allowing heterogeneous networks that mix different learning rules and static synapses, evo-

lution can choose – within the constrains and limitations given by the user – the appropriate locations for adaptive synapses to find the best learning strategy for each subnetwork.

Adaptive synapses have many potential advantages. With suitable learning rules, evolution merely has to evolve the structure and the rough starting configuration for a network, instead of optimizing all detailed synaptic weights. The fine-tuning then can be learned during the evaluation phase. This can speed up evolution, because, at least in theory, less generations should be needed to find the optimized controllers. The evolution of adaptive networks may also benefit from the Baldwin effect (Baldwin 1896; Downing 2009, 2010; Mitchell and Forrest 1994). Hereby, it is assumed, that in the beginning of an evolution, highly adaptive networks evolve, that are capable of solving the problem by learning during the evaluation time. In later generations the networks continue to evolve such that the required learning effort with respect to the problem is minimized, so that the final networks do (almost) not need a learning phase any more.

In addition to the Baldwin effect, also Lamarckian learning strategies (de Lamarck 1819) can be implemented. In such an evolution one could translate the evaluated network after its learning phase back to the genotype, so that the learned strategy can be inherited to the children. A less drastic strategy would be to evaluate a network and then change all the synaptic weights of the original network towards the direction of the synaptic weights of the learned network, ideally combined with an adjustable learning rate parameter. This would shift the networks slowly into the direction of the learned synaptic weights, avoiding a too strong adaption to outliers and special cases of the evaluation.

In combination with the constraint masks of ICONE, these effects can be exploited by evolution even for larger networks, because the affected network areas can be systematically specified based on domain knowledge, leading to smaller search spaces and a better focus of the learning features to those network structures that are expected to benefit most.

6.8 Morphology Co-Evolution

The co-evolution of animat morphologies and control networks is a promising and often applied approach (Bongard and Paul 2001; Floreano et al. 2008b; Hornby and Pollack 2001, 2002; Paul and Bongard 2001; Pollack et al. 2000; Sims 1994a, b) to evolve neuro-controllers for robots with variable morphologies. Many control problems can significantly be simplified by using especially suited morphologies (mass, center of gravity, arrangement of body parts) for the robot. The problem is, that in most cases, these optimal morphology configurations are not known. Therefore, it makes sense to evolve the morphology of the animat as well.

ICONE supports the evolution of morphology parameters via special networks, whose nodes represent the adjustable morphology parameters. These networks can be constrained like any other neural network. Thus, it is easy to evolve morphologies with symmetries, cloned limbs and the like. Because the morphology parameters are simply described by networks with nodes similar to neurons, both types of networks can be represented and managed together in the same network (but in separate modules), so that neuro-controller and morphology are evolved together in the same genome. Mutation operators, modular

crossover and even constraints (e.g. symmetries) then can affect the animat morphology, as well as the corresponding neuro-controller, as a single unit.

6.9 Order Dependent Neurons

The reactivity of a synchronously updated neural network, as it is used as neuron model for this thesis (section 2.1), is limited by the number of synapses a signal has to cross until it has an effect. This crossing of pathways results in a delay between an (input) signal and its effect. For highly reactive controllers, such as motor controllers of a robot, this delay has to be preferably short, so that the state of the overall system (e.g. the animat) at the time of the effect is still close to the state when the causative input signal was received. Otherwise, the reactions always come too late and can destabilize the system. This is a common problem of signal processing in general (Proakis and Manolakis 2007).

One source for long synaptic pathways is the structure of a controller itself. So when superfluous, avoidable neurons and synapses are part of a controller, then the delay becomes inappropriately long. Such longer pathways can easily develop during evolution. But when a neuro-module is later refined as building block (section 5.1.2), the pathway lengths can often be minimized manually.

A second important source for long pathways is the extensive use of neuro-modules. Each neuro-module provides its own set of input and output neurons as part of its interface. If modules are connected, an additional synapse is required, that connects the output of one module with the input of a second module. This can lead to quite long delays, that are not necessarily required for functional reasons (apart from connecting the modules). In other module based methods, e.g. in Doncieux and Meyer (2004a), such delays are avoided by merging the input and output neurons of two neighboring modules. This, however, is not possible in ICONE because of potential side effects with constraints.

A countermeasure for long synaptic pathways is the *order dependent neurons* (ODN) extension. Neurons can be tagged with the ODN network tag to add an order information (execution level) to that neuron. Neurons without an ODN tag are assumed to be on level 0. In a single update step, the neurons are then not updated synchronously, but instead depending on their ODN level. Only neurons with the same ODN level are updated synchronously, starting with the lowest ODN level, until all neurons on all levels have been updated. Thus, neurons on higher levels already use the updated state of the neurons of the lower levels. Signals therefore can be propagated over several neurons of ascending levels in a single step. A network without ODN tags still behaves exactly like a normal synchronous network, because all neurons are on the same level and thus are updated synchronously. With ODN levels, synaptic pathways can be reduced greatly, not only to compensate for the connective synapses between modules, but also to further reduce the delay of complex functional modules. Many multi-neuron modules can be refined to calculate their output signal in a single step, which increases their reactivity.

Order dependent neurons are primarily suited for peripheral structures, because there the user can fully control the properties of each neuron. An automatic definition of update levels during evolution is not easy, because the choice of the levels is dependent on the

function and context of a neuron. The peripheral structures are the only structures where the required context knowledge is available. An exception may be the direct connections between modules. These can, if desired, be automatically set to a high ODN level when inserted to avoid the additional delay of these purely module-connecting synapses.

Another way to use ODN levels in a semi-automatic way is the inclusion of such levels during the refinement of neuro-modules. This is possible, because in this phase the function and context of all neurons is known. If such modules are then instantiated during evolution, then the ODN levels are already provided and the controller can benefit from the shorter delays instantly.

Chapter 7

Application: Walking with a Humanoid Robot

7.1 Experiment Description

7.1.1 Approach

The goal of this experiment is the development of a neuro-controller for a walking behavior of a physical humanoid robot. The target platform is the so-called A-Series humanoid (Hild et al. 2007, figure 7.1), a 45 cm high robot that was developed at the Neurorobotics Laboratory of the Humboldt University of Berlin. The robot is based on the commercial BIOLID robotics kit[1]. Each robot is about 45 cm high and equipped with 21 servo motors,

[1] http://www.robotis.com

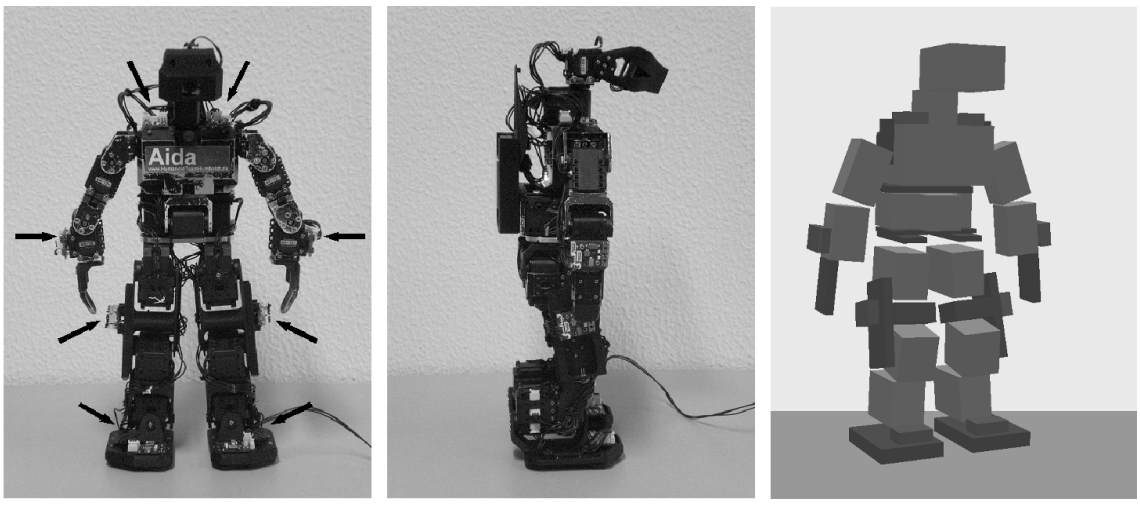


Figure 7.1: The A-Series humanoid robot and its simulation. The arrows in the first picture show the locations of the eight AccelBoards.

21 angular sensors and 16 acceleration sensors. Eight so called *AccelBoards*, small electronic boards responsible for the control of the motors, for reading the sensors and for calculating the activity of the artificial neural network controllers, are distributed on the body. The boards communicate over a synchronized bus with an update rate of 100 Hz. During each update step, each board receives information about the state of all motors and sensors of the robot. Because of the limited capacity of the *AccelBoards*, the neural controller networks have to be distributed over the available boards. Accordingly there is no central control, but rather a number of interacting autonomous networks. To allow the exchange of state information between the networks and accordingly synaptic connections between the networks, each *AccelBoard* can communicate the state of up to four of its neurons to all other boards.

Evolving a controller for such a robot is a challenging task. One reason are the limitations of the supported network structures due to the distributed architecture of the *AccelBoards*. Furthermore the robot provides a comparably large number of sensors and motors. Each motor is controlled by two motor neurons (desired torque and desired angular position), which results – including the 21 angular sensors and the 16 acceleration sensors – in a minimal network with 79 neurons. In principle, additional sensors, like a gyroscope or foot contact sensors, can optionally be added to the robot, so that different control strategies can be realized. This is a considerably large search space, even without additional neural structures to realize the behavior.

This experiment is a suitable problem to be solved with the ICONE method: The search space of the problem can be significantly reduced by domain knowledge, for instance by choosing, which sensor and motor neurons are actually required at which step of the experiment. Furthermore, the limitations of the *AccelBoards* can be forced on the network with functional constraints, so that all evolvable networks automatically are suitable for an upload to the physical robot. Those parts of the network, whose function is already understood from previous experiments, can be given as peripheral structures in advance so that the focus can remain on the evolution of the structures for the walking pattern. With different choices of peripheral structures and neuron exclusions, the control approach can also be influenced to examine different variations.

The overall experiment is divided into several, iterative evolution steps, to shape the evolution process, to get more control over the evolution and to increase the likeliness of successful evolutions. The first iteration evolves different controllers in simulation to make the robot march in place. Based on one of the solutions of that experiment, the actual walking behavior, still in simulation, is evolved. The final evolution step increases the behavioral robustness of the controller to allow the successful transfer to the physical machine.

For this experiment, evolutions have been run interactively first. Once successful settings have been found, the experiments have also been repeated as unsupervised batch processes. This was done, besides of the chance of finding variations of the interactively developed controllers, to collect data to provide a rough impression on the method's performance. To account for the 100 Hz update rate of the controller boards and to allow a later transfer to the physical robot, the update rate of the simulation as well as that of the neural network updates have been set to 100 Hz for all experiments in this section.

7.1.2 Involved Techniques

The experiments in this section apply the ICONE techniques listed in table 7.1. Details on these techniques can be found in the corresponding sections given in the table.

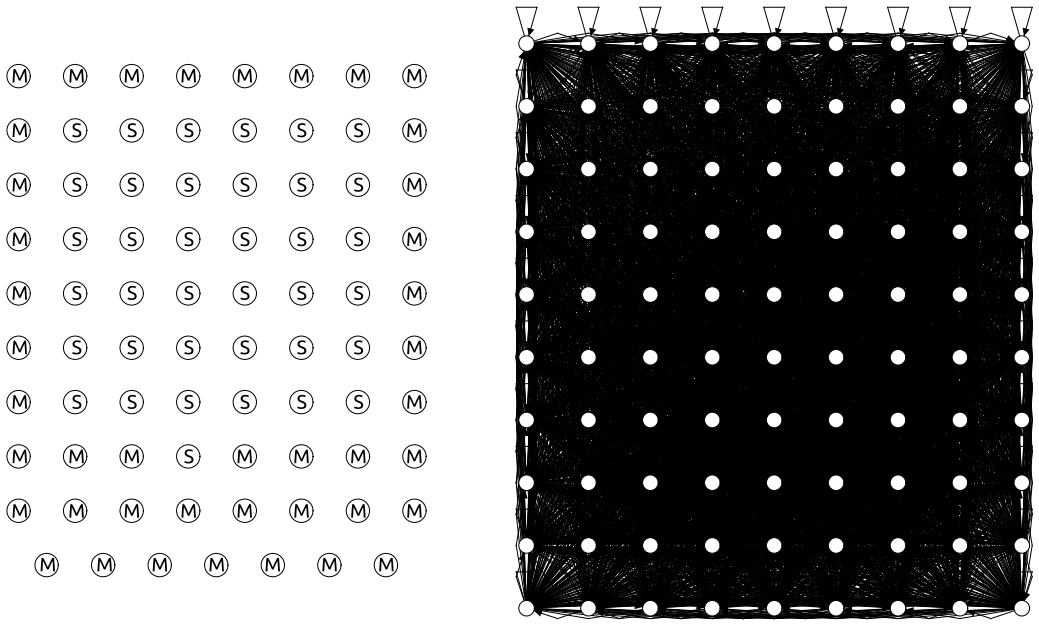
Type	Technique	Section	
Evolution	Interactive	5.2	p. 65
	Iterative	5.3	p. 67
	Unsupervised for Variation Exploration	5.2	p. 65
Crossover	Active	3.3.3	p. 39
Constraints	Cloning	4.1	p. 45
	Symmetry	4.2	p. 45
	Connection Symmetry	4.3	p. 46
	Restrict Number of Neurons	4.8	p. 49
	Restrict Weight and Bias Range	4.11	p. 50
	Maximal Number of Outgoing Synapses (as a part of Connection Density)	4.13	p. 52
Extensions	Synaptic Pathways	6.1	p. 69
	Order Dependent Neurons	6.9	p. 75
Property Tags	Protection	3.2.4	p. 31
	Network Symmetrization (Neuron Flipping)	3.2.4	p. 32
	Order Dependent Neurons	3.2.4	p. 32
	Mutation Control	3.2.4	p. 31
	Mutation Hints	3.2.4	p. 31
	Module Types	3.2.4	p. 32
	Auxiliary Tags (Code Export Information)	3.2.4	p. 32

Table 7.1: ICONE techniques involved in the humanoid walking experiments.

7.1.3 Search Space

To better comprehend the involved search space, figure 7.2 shows the plain, unconstrained network of the full A-Series humanoid robot. The network shown in figure 7.2(a) is the minimal network that can be uploaded to the hardware. The plot also shows that it is quite difficult to handle such an unconstrained, arbitrarily arranged network, even if the names of the motors and sensors would be given. As a contrast, a network after the constrained modularization phase (figure 7.4 on page 83) is much easier to understand due to the inherent layout and the ordering of neurons according to their function and location on the robot (although that network has twice the number of neurons compared to figure 7.2(b)). Figure 7.2(b) also shows that the unconstrained search space even in a minimal network is already very large, so that successful evolutions become unlikely for many non-trivial experiments.

In the forthcoming experiments, the search spaces of the constrained networks are in each case given as comparison, so that the differences of the involved search spaces between the constrained and the unconstrained networks becomes clearer.



(a) Unconstrained Initial Network
Parameters: B: 42 W: 3318

(b) Search Space (+11 Neurons)
Network: N: 90 S: 4770
Parameters: B: 53 W: 4770

Figure 7.2: Search space of an unconstrained network for the A-Series humanoid robot. In (b) the search space of a network with 11 additional neurons is shown. This number of neurons is comparable to the networks evolved with the constrained networks in the forthcoming experiments.

The detail figures of each controller also gives information about its complexity, in particular the number of neurons (N) and the number of synapses (S) of the *entire* network, and the number of actually mutable parameters of the network, i.e. bias terms (B) and weights (W) (compare figure 7.2).

7.2 Marching in Place

Due to its physical constraints the physical A-Series humanoid robot cannot stand stable on a single leg. The motors of the A-Series, especially in the ankles, are too weak to simultaneously counteract the weight of the robot and to do fine control of a desired angular position. Therefore, walking is not assumed to be a static movement from a stable one-legged standing position to another, as often seen with humanoid robots that are controlled by zero-moment-point (ZMP) approaches (Peterka 2009). Instead, a dynamic, pendulum-like walking is implemented. The stability for the A-Series robot has to originate primarily

from the dynamics of the pendulum-like lateral swinging of the robot while staggering from one leg to the other.

As a starting point for walking it is assumed that the robot should be able to lift its legs in a regular, periodic way. To achieve this, the first task for the robot is to step from one leg to the other on the spot without falling over.

7.2.1 Experiment

Simulation Environment. The environment of the simulated robot consists of four balks at a height of 10 cm that restrict the operational range of the robot (figure 7.3). These balks support the development of stepping behaviors that keep the robot near its starting position. All collisions between the robot and the balks or the ground (except with its feet) stops the evaluation immediately. The approach to stop evaluation at the violation of hard constraints has shown its benefit in many other evolutions (Pasemann et al. 2003b; Rempis 2007; von Twickel et al. 2011). It speeds up evolution by preventing wasteful evaluation time on controllers, that do not fulfill all requirements on the behavior. Combined with a suitable fitness function that prefers individuals with longer evaluation time, the evolution of controllers outside the desired specifications can be avoided right from the beginning.

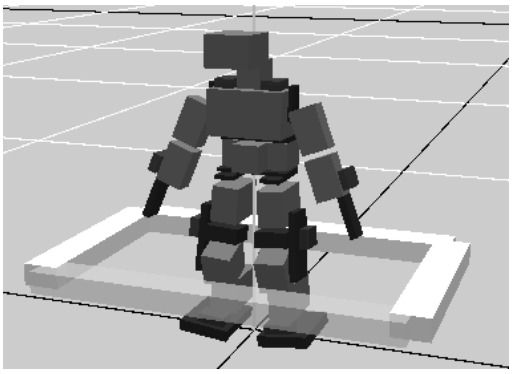


Figure 7.3: The simulated A-Series robot in its evaluation environment for marching in place with constrained operational range.

Fitness Function. The fitness function combines several aspects of the stepping motion, that can be weighted separately with parameters δ and γ. The first aspect is the maximal stepping height h during one step. This favors controllers that lift the legs as high as possible. The second aspect is the step duration d. The longer a footstep takes, the better, because for walking large steps are desired, which require some time.

To use footsteps j as the measuring unit in the fitness function, and due to missing foot contact sensors, footsteps are detected monitoring the height of the feet. A new footstep is assumed to take place, when the difference between the minimal height of both feet changes its sign. To avoid the false detection of footsteps caused by noise or *vibrating* behaviors,

the feet have to reach a minimal distance $|d_{min}|$ after the sign change. d_{min} hereby can be adjusted as a parameter of the fitness function. The current footstep count at time step i is s.

The fitness f at time step i is then given by

$$f(i) = \sum_{j=0}^{s-1} (\delta h_j + \gamma d_j), \tag{7.1}$$

that is, $f(i)$ is the sum of the maximal heights h_j at footsteps j and their duration d_j up to the previous footstep $(s-1)$. Only summing up to the previous footstep avoids controllers that try to maximize duration or height in an uncorrectable one-time attempt, such as lifting the leg very high by falling over to the side. Fitness therefore is only gained for motions that lead to another footstep.

7.2.2 Modularization

The starting network for this behavior is modularized and constrained with the techniques described in section 5.1. As can be seen in figure 7.4 the basic neural network with its 38 sensor and 42 motor neurons would be difficult to understand without structuring the network.

All modules with names starting with AB (abbr. for *AccelBoard*) represent the hardware controller boards on the robot. Using modules to structure the network according to the hardware boards allows an intuitive understanding of the location of the sensors and motors. Furthermore, constraints on these modules enforce evolved networks to satisfy all constraints originating from the hardware, like the maximum number of neurons and synapses per board (limited by memory and execution time), and the maximum number of neurons visible to other boards (limited by the communication bus). Therefore, the number of neurons with synapses leaving each AB module, and the number of neurons per AB module have been restricted via constraints. Obviously, neurons outside of the AB** modules are prevented, because otherwise the network could not be transferred to the hardware.

To restrict the search space for the evolutionary algorithm, all motors and sensors not needed for the transversal movements, have been *protected* and hence cannot be targeted by mutation operators. This affects all motors and sensors, except the transversal hip and ankle motors, including their corresponding angular sensors, and the transversal acceleration sensors at one shoulder and the feet. Arms and knees have been fixed with bias terms at suitable angles to support the task statically. The motor torque neurons have been fixed with a bias of 1.5, so the networks in this experiment are forced to control the motor angles with maximum torque. As stated in chapter 5 this biases the search towards a certain solution approach, here to control the motion with the angular motors, based on acceleration sensors. Different constraints would lead to very distinct solutions, e.g. when the motors would be forced to be controlled by torque or by including other sensors (see section 7.2.5).

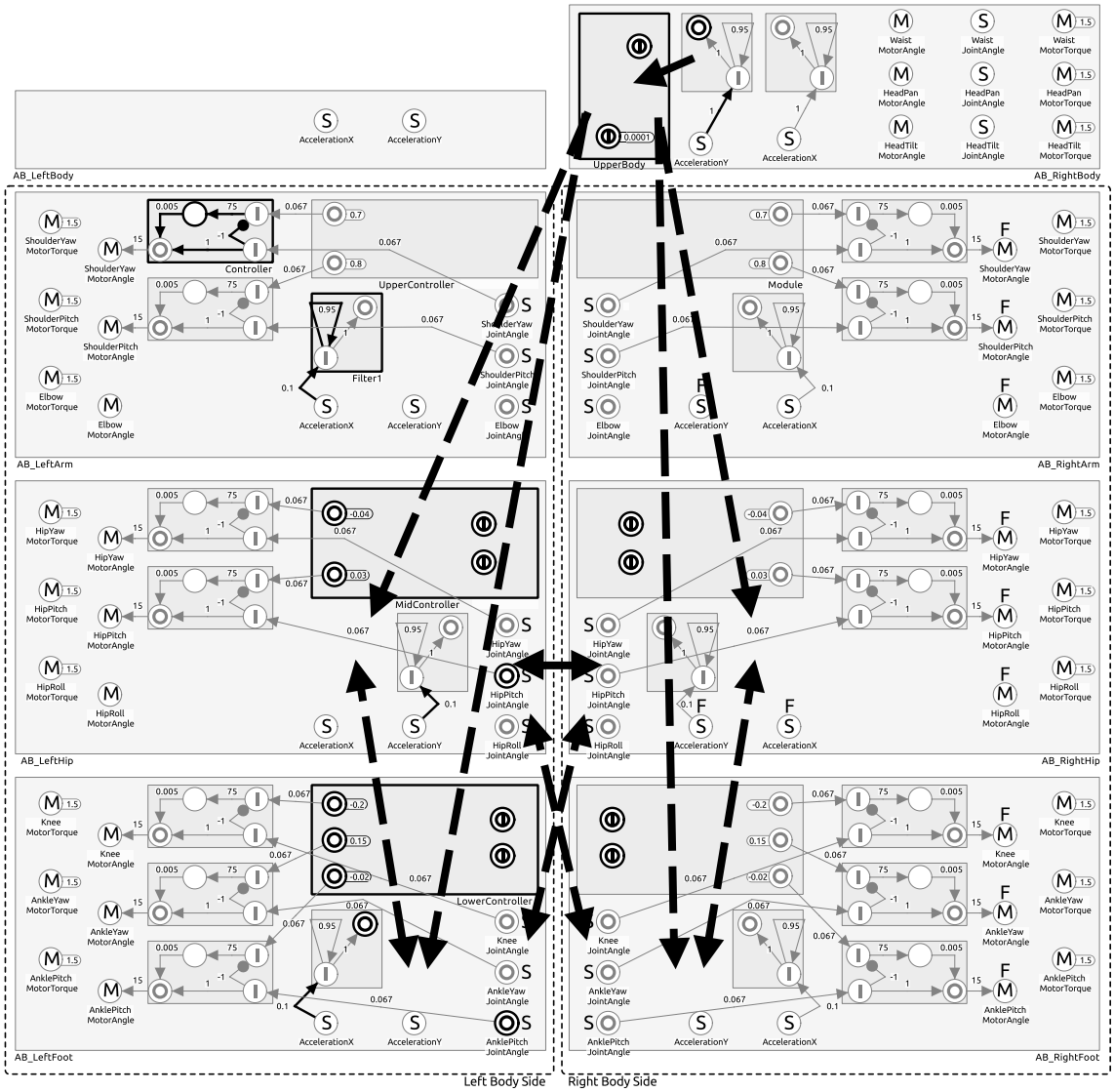


Figure 7.4: The modularized initial network for marching in place. Neurons with names are interface neurons of the robot (motors and sensors). Neurons marked with an F have been flipped to symmetrize the network.

As an additional constraint the lower six AB modules have been organized into two larger groups to support a symmetry constraint between the left and the right side. The elements on the left side are horizontally mirrored to the right side. All incoming synapses of these two modules have been chosen to be anti-symmetric, i.e. they get synapses coming from the same external neurons, but with reverse signs. All mutual synapses between both sides have been chosen to be symmetric. The physical robot is not fully symmetric with respect to the motor and sensor arrangements, because the motors and sensors have been assembled with focus on a suitable positioning, rather than on the activation ranges. Thus,

when actuating, for instance, both arms with the same activation, then one arm moves forwards and the other one backwards. For a symmetric behavior, both arms should do the same thing when activated similarly. Therefore, all asymmetrically assembled motors and sensors have been flipped with a network tag (neurons marked with an F in figure 7.4) to reverse their activation range, so that the robot behavior again is symmetric for symmetric neuron activations.

The `MotorAngle` neurons of the A-Series represent the desired joint angles of the motors. Accordingly, no additional controller is required to hold a given angle. Nonetheless, it makes sense to connect each motor neuron with a controller, that limits the rate of change of an angle setting to smoothen the motions, to protect the motors and to simplify the transfer of the controllers to the hardware later on. The latter is the case because the motors behave less predictably near their operational limits and hence are difficult to simulate adequately for this case. The structure of these controller modules is given as peripheral structures in advance, but the synapse weights are open for mutations to manipulate their reactivity and characteristics. Because each motor neuron should be equipped with such a controller, it makes sense to use only a single mutable controller prototype in the network (module `Controller`), and a clone of this prototype in each place where a controller is needed. That way, only a single controller module is part of the search space. The same holds true for the filter modules used at each acceleration sensor. The signals of the acceleration sensors are not smooth and consequently difficult to use. Filtering the signal reduces the effect of spikes, but induces a delay. Therefore, the filter properties of one prototypic filter module should be open for evolution to find the best suitable filter behavior, while every other acceleration sensor is filtered by a clone of this mutable module. In both cases, the function-relevant synapses have been tagged to restrict their weights to plausible ranges and to reduce the mutation variance to only 50 percent of the global setting, which ensures that the mutations of these weights are comparably small.

The reactivity of these predefined modules has been increased with *order dependent neurons* (section 6.9) to compensate for the longer synaptic pathways involved with these modules. This includes also the synapses connecting these modules with the main focus structures and the sensors and motors of the robot.

The main focus structures are forced to develop within separate submodules (`Upper-` , `Mid-` , `Lower-Controller`). These modules can be exchanged by the modular cross-over operator and additionally define a neural interface limiting the number of connections between the control modules. The interface neurons have been extended in their visibility depth so that direct connections between these six control modules and the upper body module are possible.

Finally *synaptic pathways* (section 6.1, black dotted arrows) have been introduced to restrict the possible connections between the specified modules. These pathways force all new synaptic connections to be added only between the specified modules, including visible interface neurons of their submodules. Here, only connections from the shoulder sensors to the hip and feet modules, and between the feet and the hip modules, are allowed.

The evolvable neurons and modules are illustrated in figure 7.4: All modules that can be modified during evolution are highlighted with thicker, black lines. All other modules are either fixed (*protected*), or depend on one of the other modules due to a constraint.

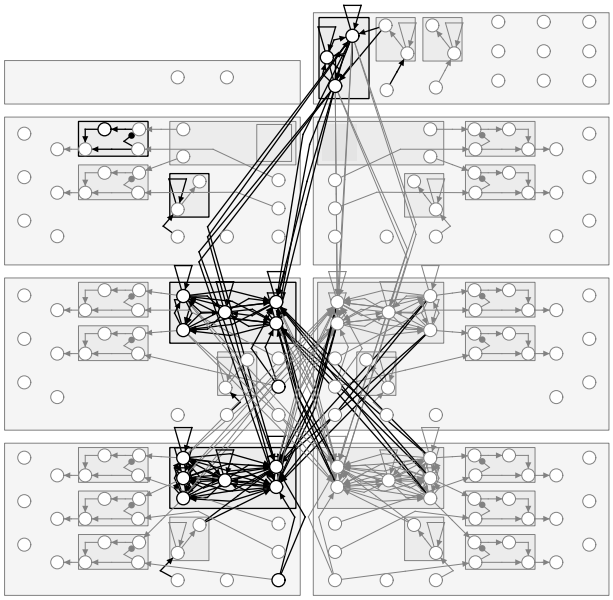


Figure 7.5: Parameter Space. The bold network elements are subject to mutations.
Search Space (+1 Neuron per Module)
Network: N: 180 S: 319 Parameters: B: 17 W: 113

However, in this graphics, not all degrees of freedom can be visualized, but it gives a good impression on the involved search space. Details on the maximal search space with 1 additional neuron per processing module are shown in figure 7.5.

7.2.3 Parameter Settings

The experiments have been conducted with a variety of parameter settings for the evolutionary algorithm (see table 7.2). These parameters were interactively adapted during the evolution according to the convergence behavior of the algorithm. Therefore, instead of exact parameter settings for each experimental run, only general ranges are given, in which the parameters have been varied during each evolution run.

As a rule of thumb, all operators mentioned in table 7.2 have been active, starting with very low probabilities for search space extensions (*Add Neuron*, *Add Synapse*) and with a relatively high potential for changes of synapse weights and bias values (*Change Bias*, *Change Weight*). During evolution the probabilities for structural changes were increased when the results converged to an undesired local optimum, so that more complex structures could evolve to overcome the local optimum. The probabilities of weight and bias changes and their average amount of change were decreased when a promising area of the solution space was reached, so that the behavior of the network could be fine-tuned.

The probability for modular crossover was 0.5 to support the transfer of genetic material between lines of ancestries. The number of trials indicates how often each individual is evaluated with slightly randomized environments, e.g. alterations of the starting angles.

Operator	Parameter	Interactive	Unsupervised
General	Population Size	[100, 200]	150
	Max Simulation Steps per Trial	[500, 3000]	3000
	Number of Trials per Evaluation	[2, 5]	3
Tournament Selection	Tournament Size	[3, 5]	4
	Keep Best Parents (Elitist)	1	1
Modular Crossover	Crossover Probability	0.5	0.5
	Crossover Probability per Module	0.5	0.5
Remove Neuron	Probability	[0, 0.005]	0.002
	Number of Removal Trials	[0, 2]	1
Remove Synapse	Probability	[0, 0.01]	0.002
	Number of Removal Trials	[0, 5]	5
Remove Bias	Probability	[0, 0.01]	0.004
	Number of Removal Trials	[0, 3]	2
Add Neuron	Probability	[0, 0.005]	0.002
	Number of Insertion Trials	[0, 2]	2
Add Synapse	Probability	[0, 0.02]	0.01
	Number of Insertion Trials	[0, 5]	2
	Init. Insertion Probability	[0.01, 0.1]	0.01
Add Bias	Probability	[0, 0.02]	0.005
	Number of Insertion Trials	[0, 3]	2
Initialize Synapses	Min	[-20, -1]	-5
	Max	[1 - 20]	5
Initialize Bias	Min	[-5, -1]	-2
	Max	[1 - 5]	2
Change Bias	Change Probability	[0.005, 0.015]	0.01
	Deviation	[0.005, 0.2]	0.1
	Reinitialization Probability	[0, 0.005]	0.001
Change Weight	Change Probability	[0.01, 0.2]	0.05
	Deviation	[0.005, 0.2]	0.1
	Reinitialization Probability	[0, 0.005]	0.002

Table 7.2: Settings of the main evolution operators. The settings are given as ranges in which the parameters have been varied during interactive evolution, and as fixed settings for the unsupervised evolution runs. The functions of the operators are listed in appendix B on page 177

As the selection method an implementation of the standard *Tournament* selection (Miller and Goldberg 1995) was used.

After evolving several networks with the interactive evolution approach, the experiment has been re-run on the computer cluster for approximately 30 times in unsupervised mode. The evolutions have been run with fixed parameter settings given in the last column of the parameter table. These unsupervised evolution runs have been performed to search for varieties of networks and to produce a simple form of performance analysis, that should give a rough indication of the algorithm performance.

7.2.4 Results

The evolution was performed 33 times for about 100 to 300 generations per evolution, depending on the observed progress.

The evolution experiments have been terminated either manually, or automatically when the fitness did not increase further for at least 50 generations. In 21 cases, networks

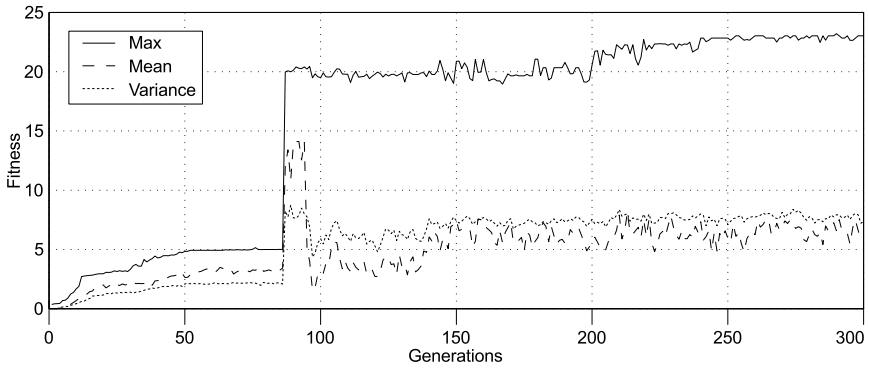


Figure 7.6: Maximal and mean fitness of the best evolution run for the march-in-place task.

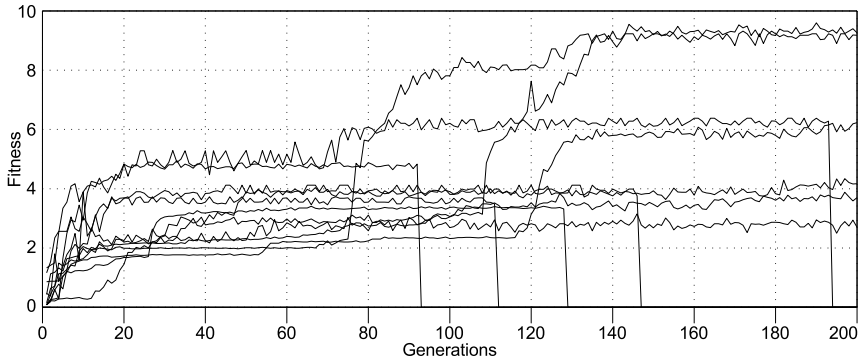


Figure 7.7: Maximal fitness of the 10 best evolution runs not including the very best run (figure 7.6) for scaling reasons. Fitness curves dropping to zero indicate evolution runs that were interactively stopped due to stagnation.

have been found that solve the task and provide a valid starting condition for the next evolution scenario. The fitness progress during the evolution (maximal and mean fitness, variance) for the best solution is shown in figure 7.6. The progress of the maximal fitness for the next best 10 evolution runs are shown in figure 7.7.

Because of the constrained network the implemented strategies are not too surprising (figure 7.9). In the networks driven by the acceleration sensors, the main strategy was to destabilize the robot with the transversal hip or ankle motors according to the swing phase. Once swinging, the transversal acceleration sensors provide an oscillatory signal, that is used to control the hip or ankle motors. An example of such an acceleration sensor driven behavior is shown in the motion sequence in figure 7.8.

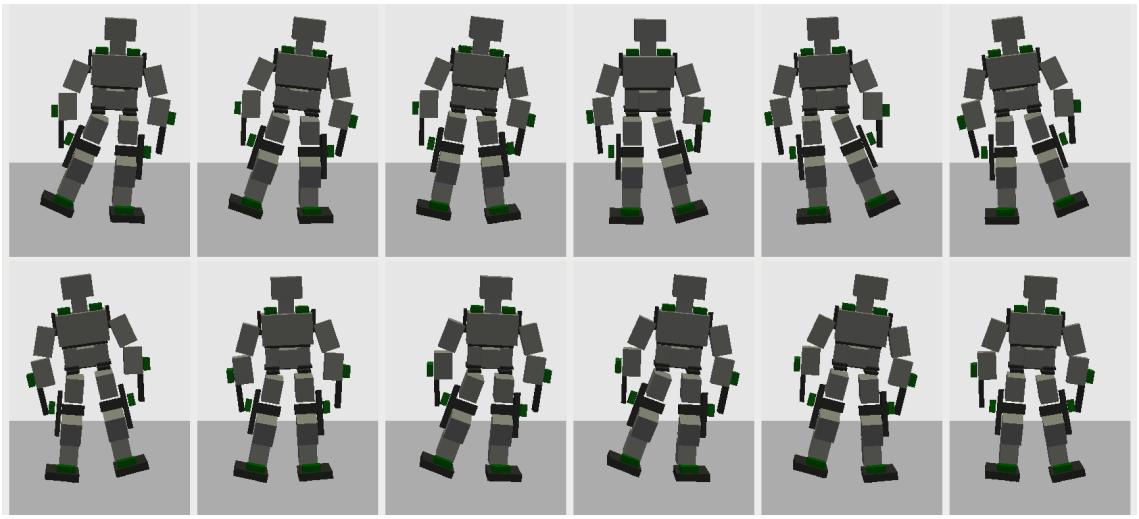


Figure 7.8: Motion sequence of the robot controlled by the network of figure 7.9 for the march-in-place task. The picture series shows every 100th simulation step.

However, 12 evolutions did not come up with satisfactory solutions. Even with the symmetry constraints between the left and the right side, many behaviors resulted in unstable, irregular motions. This was especially the case when the acceleration sensors of the feet were connected to the ankle motors. As the movement of the ankle motors directly influences the acceleration sensors on the feet, this leads to an isolated, unsynchronized swinging behavior locally to each leg, which could not generate a stable global behavior. This suggests that the feet sensors may only be useful if the feet remain at fixed angles while their sensor signal is used to control the hips. Such solutions can be avoided interactively during evolution by preventing these kind of connections and therefore by excluding this type of frequent local optimum. This example also indicates, that, if even a related sensor can disturb the behavior so significantly, such disturbances can be expected to be much worse for connected unrelated sensors, which underlines the importance of restricting the valid connections between the distinct body parts.

Figure 7.7 gives a rough overview on the performance of the algorithm for this experiment. As comparisons with other algorithms are difficult due to the involved constraint

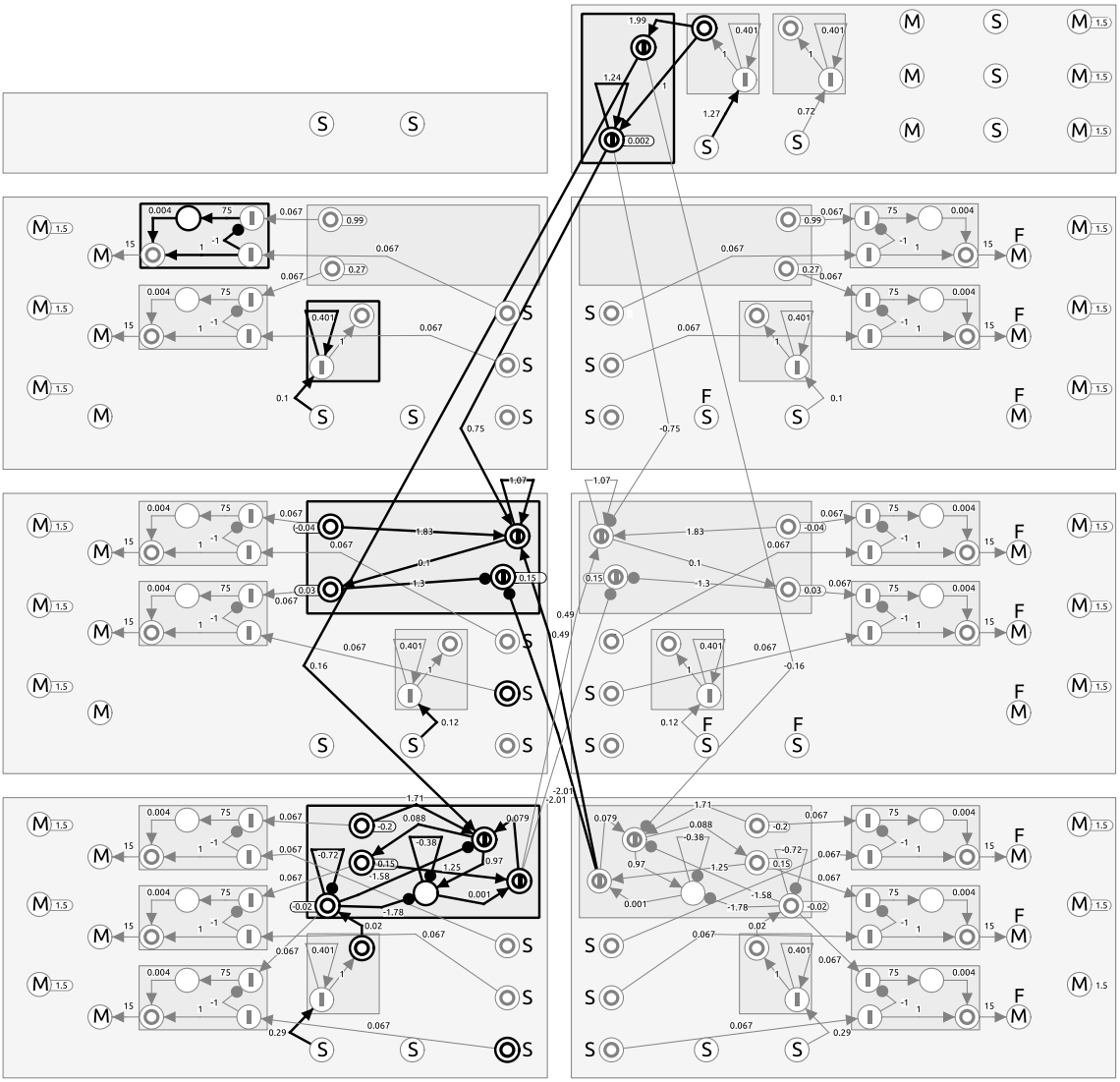


Figure 7.9: Network of a successful marching-in-place network using the acceleration sensors at the shoulders to control primarily the hip motors and slightly the feet motors to destabilize the robot during the swing phases.
Network: N: 177 S: 163 Parameters: B: 9 W: 31

masks and the interactive approach – and therefore due to the involved user experience –, these graphs should give a general idea about the usability of the method. Nevertheless, one observation can be made: applying the ENS^3 algorithm or NEAT to an unconstrained, empty starting generation with networks of this size, solutions for this problem have not been found at all.

7.2.5 Experiment Variants

The experiment has been conducted in several variants to find different controllers to solve the task. Because the identification of controller variants using *network shaping* is the focus of chapter 8, no details on these variations are given in this section to avoid a disruption of the sequential, iterative evolution approach. However, a short list of conducted variants should give an impression of how the search space can be explored systematically.

One series of experiments examined whether an oscillator can be used as a central pattern generator to let the robot swing from one leg to the other. For this, the initial network provided several, exchangeable types of oscillators, that could be integrated by the evolution. Some of the oscillators had a fixed frequency, which could be adjusted during evolution, while others had a more flexible configuration that allows the change of frequency and amplitude at runtime as part of the network dynamics. All these approaches could indeed make the robot swing, but in all cases the swinging did not last very long. This was due to the difficulty to react on small deviations of the swinging phase and hence the resulting desynchronization of the oscillation with respect of the body swinging. Such solutions therefore are not suitable for the next evolution step.

A second variant was the examination of different sensors and motors enabled for the evolution. This includes different sets of enabled angular and acceleration sensors, but also the introduction of new sensors. The A-Series robot is designed to be extensible by foot contact or force sensors in the feet, that allow the robot to detect when the foot touches the ground. Evolutions with these sensors also led to well performing controllers. Another alternative sensor was a gyroscope sensor that was attached to the body. With this sensor, the robot can detect the orientation of its upper body with more accuracy compared to the acceleration sensors, but also with a different characteristics curve of the sensor, leading to different networks compared to networks based on acceleration sensors. These additional sensors, although being nice case studies for future robot configurations, cannot be used with the current version of the robot hardware and therefore are also not suitable for the next evolution step.

Figure 7.10: Motion sequence of a controller that maximizes the height of the feet during each step. The motion sequence shows the maximal deflections of the legs, so the pictures are taken in irregular intervals. The elapsed simulation steps between the pictures of this series are therefore given in the pictures.

A last variant was the extension of evolved controllers with respect to the height of the steps, allowing additional motors and sensors to use the hip and knee motors to raise the legs further. In these experiments, also a mixture of different neuron models (a mixture of linear and sigmoidal transfer functions) was tested with the expectation, that the suppression of neurons would become easier. Due to the dominating approach to lift the entire, straight leg like a soldier (figure 7.10) in irregular heights and durations, and because of the use of neuron models not available on the physical robot, solutions of this evolution have also not been considered as a base for the forwards movement.

Other variants, that have not yet explicitly been performed, include the examination of different symmetries, different motor controllers (e.g. torque driven), varying peripheral structures, alternative synaptic pathways to enforce a specific signal flow, and other variants of the neuron models.

This list should briefly inspire the reader, that even in simple experiments, many variations are possible, that all can lead to interesting results. With the network shaping features of ICONE, such variants can be explored in a systematic, stepwise and uniform way (see chapter 8).

7.3 Humanoid Walking

Based on the results of section 7.2 the next iterative step towards walking was conducted: modifying a stepper network to move forwards. Adding forward movement is only one next possible step. A possibility could have been first to stabilize the stepping behavior to make it more robust to small bumps on the ground or a shaking floor. Another next step could have been an optimization of the leg lifting by involving the motors and sensors of the sagittal plane of the knees, ankles and the hip. However in this document the next step is to continue directly with walking forward due to space limitations.

7.3.1 Experiment

Simulation Environment. The environment for the walking experiment (figure 7.11) gives the robot space to walk forwards, but still restricts its operational range to the sides and backwards. In these directions balks obstruct the path of the robot. As in the experiment before, collisions with these balks immediately stop evaluation. In consequence, undesired behaviors, like moving backwards or in circles, can be avoided efficiently. To avoid a common local optimum during evolution, namely moving by *vibrations* instead of steps, obstacles have been introduced in regular intervals. To overcome these obstacles, the robot has to lift its legs high enough to get the feet over the obstacle. To avoid the robot from tilting when the obstacle gets below the feet, the obstacles are implemented as sliders, that are lifted to their target height with a soft spring. Stepping on such an obstacle just makes it slide back below the ground without resulting in a bump. Therefore the obstacles hinder the feet only when colliding horizontally, not vertically.

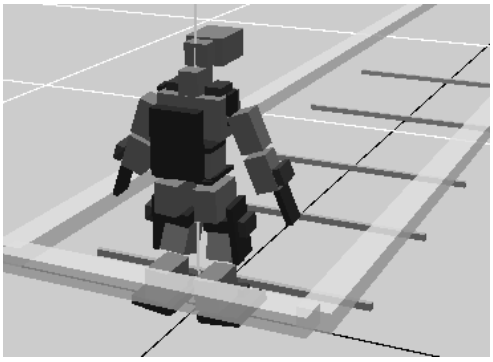


Figure 7.11: The simulated A-Series robot in its evaluation environment for walking with constrained operational range and obstacles.

Fitness Function. In the fitness function a new term is introduced: x_{max} is the maximally reached distance along the x-axis during the entire trial. The fitness function extends the one from section 7.2.1 by a weighted factor that rewards moving along the x-axis:

$$f_i = \sigma x_{max} \sum_{j=0}^{s-1} (\delta h_j + \gamma d_j)\,, \tag{7.2}$$

that is, f_i is the sum of feet height and footstep duration multiplied by the weighted distance x_{max}. The distance from the origin at time step i, x_i, is the minimum of the distances of the head, waist and both feet from the origin at that time step. Taking the minimum distance of several parts of the body prevents the robot from becoming easily trapped in a common local optimum, where the robot catapults the single relevant body part – e.g. the head or one of the feet – as far as possible. Such optima have to be avoided right from the beginning, because they are often dominant in the beginning of the evolution and in general do not provide a suitable path towards desired solutions.

7.3.2 Modularization

The initial network for the walking experiments has been derived from a solution network of the previous task (figure 7.12).

It was chosen to use one of the accelerator sensor driven solutions. Here, three simple hysteresis neurons are used to sustain the signal from the acceleration sensor of the shoulder, so that the two swing phases are more persistent and can influence the motors more enduringly. The swinging is solely generated by motions of the lateral hip motors and thus a corresponding shifting of the center of gravity. The base network was pruned to get the smallest working version of the network, so that evolution may start with a minimal network again. That network is not the best evolved solution to the march-in-place task, but it was chosen because of its simplicity. And because now, additional joints (ankles, knees, arms) can be used, this simple swinging mechanism seems to be a better basis than a complicated, difficult to extend network with many synapses.

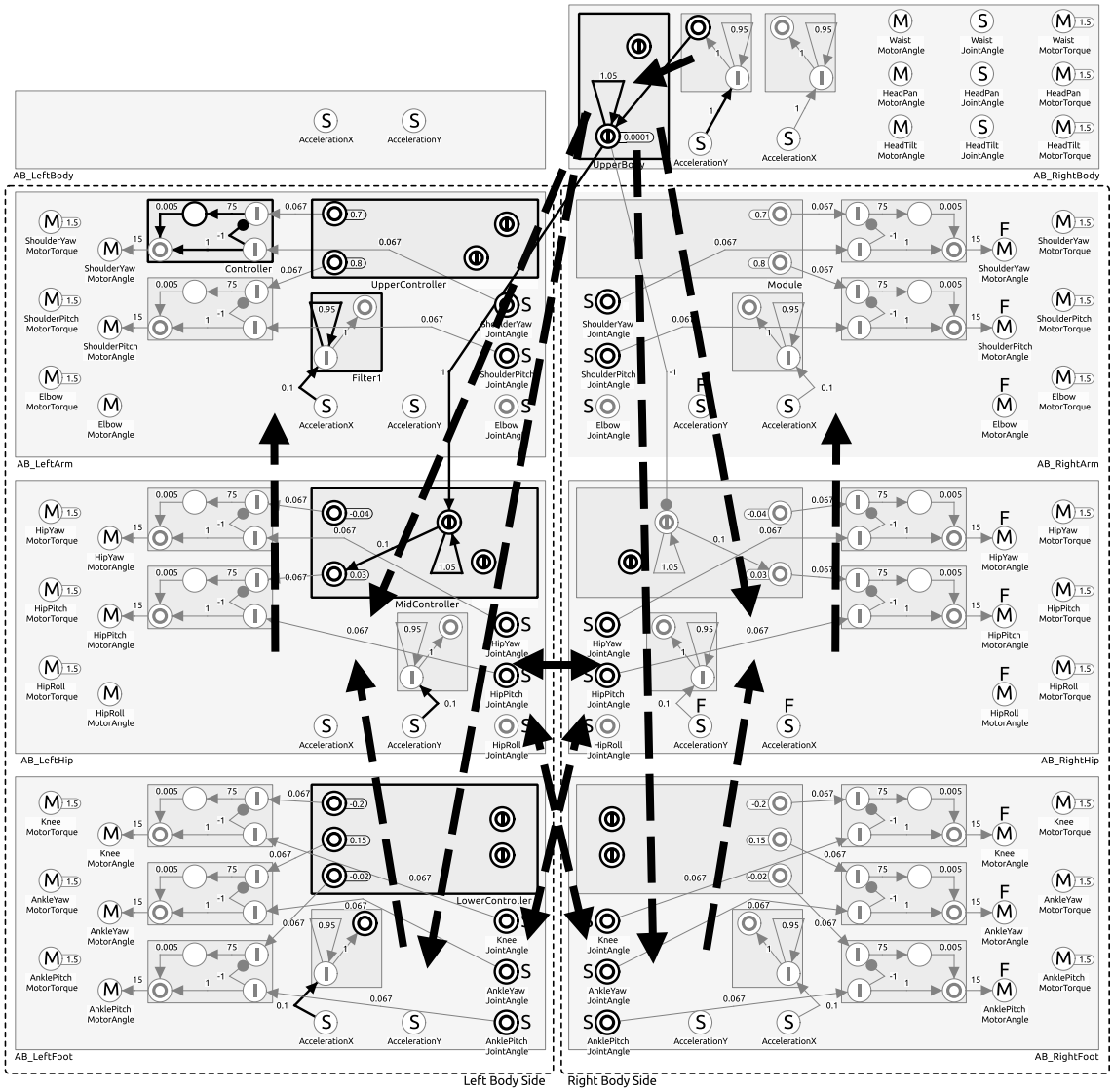


Figure 7.12: The initial neural network with its constraint mask for walking, based on a pruned, strongly simplified solution of the march-in-place task.

The major modification of the constraint mask is the enabling of additional motors and sensors by removing their *protection* tags. The networks now can also connect synapses to the sagittal motors of the hip, the feet, the knees and the arms, and to their corresponding angular sensors. This enables the robot to bend its knees to make higher steps, to move the legs forwards and backwards, to use the feet tilt to push the robot forwards, or to use the arms to keep balance during walking.

To prevent evolution from destroying the already achieved basic motion, the existence of all involved synapses and neurons have been protected. However the weights of these synapses and the bias values of the neurons have not been fixed and remain mutable.

In the upper modules additional module interface neurons have been introduced. This new interface facilitates new neural structures in the upper part of the body to make use of the arm motors and sensors. To include these arm control modules, the synaptic pathways have been adapted, so that a new pathway leads to these modules. It is assumed that the arm movement may depend on the state of the hip joints, so the synaptic pathway runs from the middle module to the arm module.

7.3.3 Parameter Settings

The evolution settings were similar to the previous experiments (see table 7.2 on page 86). Again, for the evolution no fixed parameter sets have been used, because the parameters are targets of an on-line observation and modification.

7.3.4 Results

Evolution was able to generate walking control networks in 26 of 100 performed evolution runs. The fitness progress of the best 10 evolution runs is shown in figure 7.13. The higher number of unsuccessful evolution runs may be partly a result of the interactive evolution. Undesired approaches, that do not seem to lead to a solution, can be stopped in early generations. Therefore, evolution runs may have been stopped prematurely to focus the search on more promising areas of the search space. In fact, all unsuccessful evolutions together had a average runtime of 44 generations, which is low compared to the successful evolution runs, that had an average runtime of 156 generations.

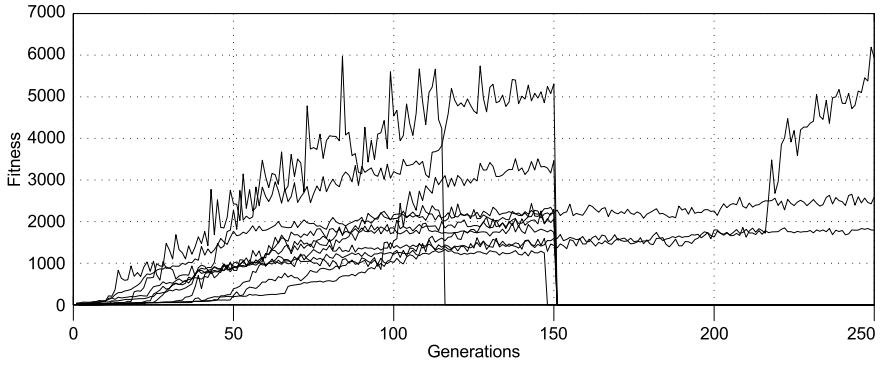


Figure 7.13: Maximal fitness of the 10 best evolution runs to solve the walking problem. Fitness curves dropping to zero indicate evolution runs that were interactively stopped due to stagnation or runtime limitations (at 150 generations).

As expected the sagittal hip or feet motors usually were used to realize the forward motion. Also, in some networks (like the one shown in figure 7.16 on page 99) the knees and arms were utilized to get a better walking performance (figure 7.17 shows the corresponding motion sequence). Surprisingly, in addition, some quite effective controllers solely used the arms to accelerate the body forwards using the inertia of the body (figure 7.15). In such networks, the arms are moved forwards and backwards to slightly rotate the robot on its

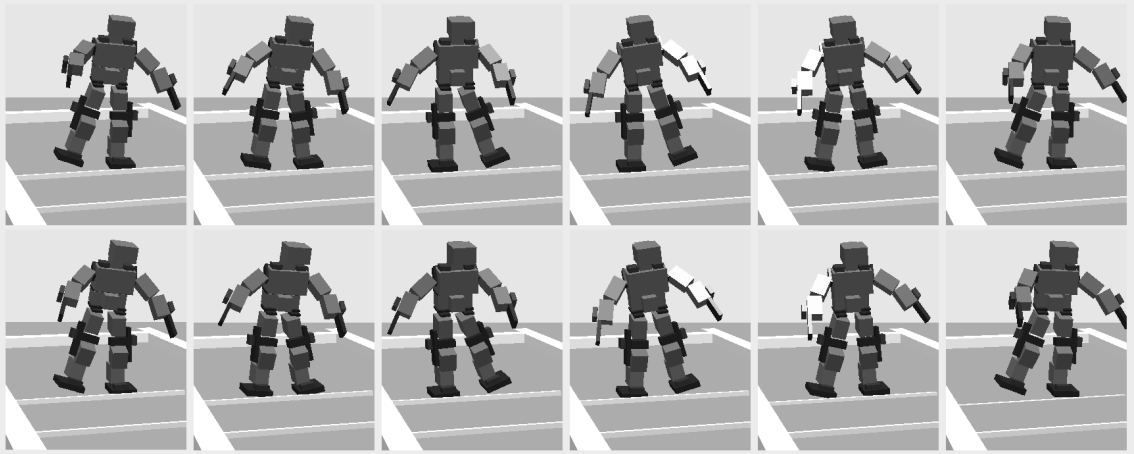


Figure 7.14: Time series of the robot controlled by a network using the arms for the forwards motion. The picture series shows every 150th simulation step.

supporting leg, so that the lifted leg touches the ground a bit further ahead than without the arm swinging. Because such controllers rely strongly on the friction between the feet and the ground, such solutions are not suitable for a final transfer to the robot. The picture series of such a behavior is shown in figure 7.14, the corresponding network in figure 7.15.

The evolved walking behaviors still are not very human-like or elegant. Also, depending on noise and the obstacles on the ground, the robot turns from time to time by a few degrees and continues its movement in the new direction. This behavior is difficult to overcome, because the robot does not have feedback sensors for the direction and therefore is not able to correct such deviations. On the other hand, taking the robot's limited motor and sensor equipment into account, these behaviors seem quite satisfactory.

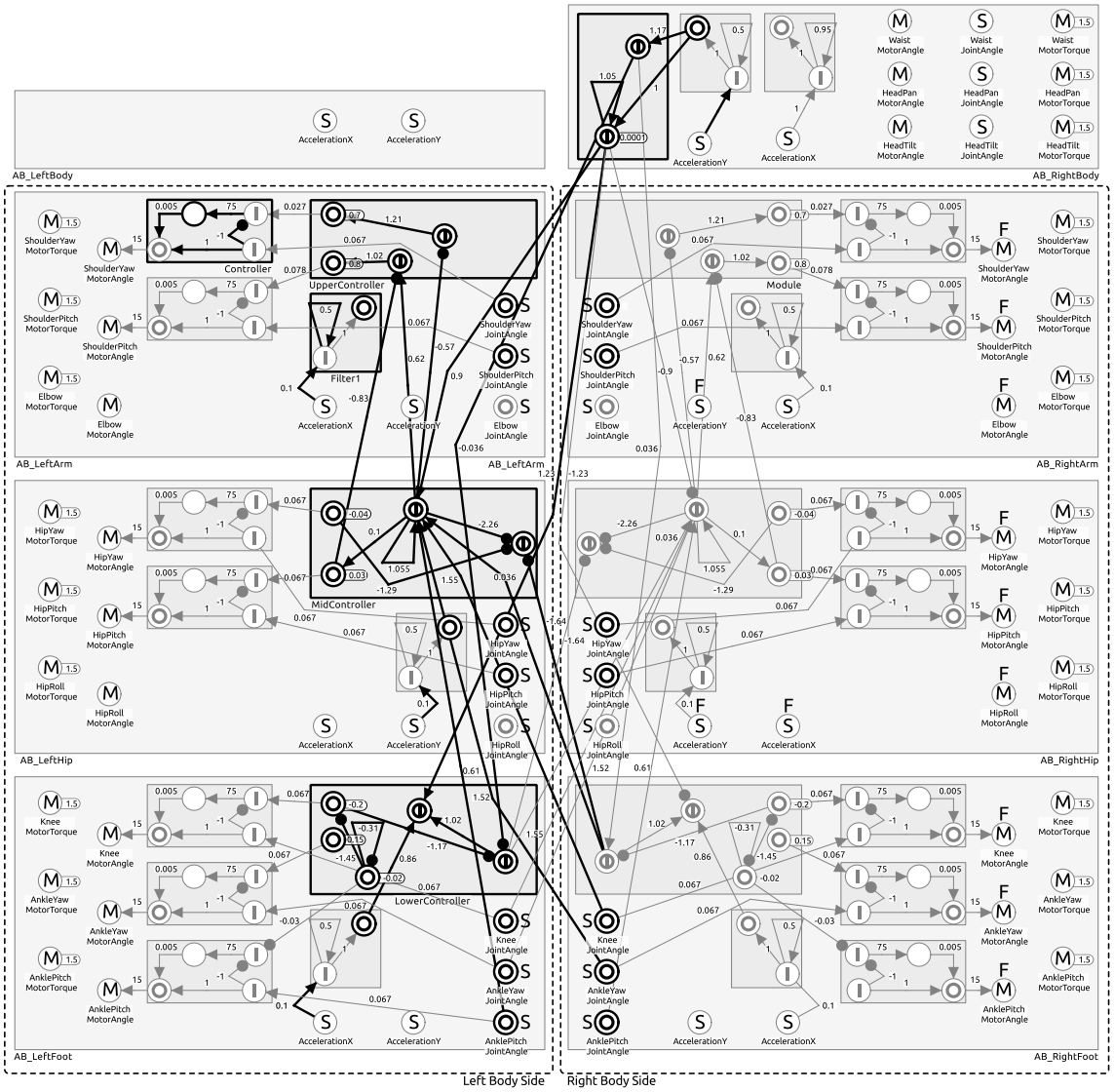


Figure 7.15: Network of a successful walker network, that primarily uses the forwards and backwards swinging of the arms to accelerate the robot forwards.
Network: N: 179 S: 169 Parameters: B: 8 W: 34

7.4 Transfer to Physical Robot

Controllers evolved in the simulator usually cannot be transferred to the physical robot without modifications, because the differences between the simulated robot model and the physical machine are significant. Although the main behavior in simulation and on the physical robot is very similar, even small differences between the two target platforms may disrupt a dynamic behavior like walking, where body and control are highly adapted to each other and may depend strongly on even fine particularities. Some differences may be

reduced by implementing more and more detailed models of the motors, the sensors and the body structure. In practice however this approach is limited by the performance of the utilized computers and the time spent for the implementation of such a model. Some aspects may not be taken into account at all, like small variations during the assembly of the robot, fabrication related differences in the motor or sensor behavior, or just behavioral differences caused by aging or heat. Other aspects are restricted by the utilized physics engine, which may not support elastic material, such as the plastic material of the robot's body parts.

Therefore adaptations of the behaviors to the target robot are usually unavoidable.

Evolving Adaptations to Physical Robot. Using simulated robots during evolution will provide neuro-controllers, that adapt to the simulated robot, but not necessarily to the physical one. Controller networks will optimize for all aspects of the simulated system, taking advantage of any implementation detail. This includes modeling errors and simplifications of the model. Because a convenient, error free robot model is not feasible, any model will have implementation details that can – and will – be exploited by evolution. This problem is called the *reality gap* (Jakobi et al. 1995).

To reduce this effect, a number of approaches have been proposed, such as adding sensor noise (Jakobi 1997; Jakobi et al. 1995; Lipson et al. 2006), (post-)evolving or adapting individuals directly on the hardware (Pollack et al. 2000) or co-evolving the simulation properties along with the behavior controllers (Bongard and Lipson 2004a, b; Koos et al. 2009). Here, our *Model Rotation* approach is used. The idea is not to evolve controllers for a single simulated robot, but for a variety of similar, but slightly differing robots. The fitness of a controller is then the minimum achieved on all of the target platforms. Consequently, the fitness corresponds to the robot model with the weakest performance. To get a high fitness, a neuro-controller has to perform well on all given robot models.

Because of this, the behaviors cannot exploit flaws of the models, as long as each model has different weaknesses and strengths. Resulting controllers are expected to work on a variety of similar robots, which means, that they are robust to small variations between the robots. Such robust networks have a higher chance to work also on other similar robots not used during evolution, including the physical robot.

The robot models used during model rotation should not be seen just as random variations (noise). Simple randomization works out only for some parameters, like the angles at which body parts are assembled to each other or the absolute size of the body parts. Randomly changing parameters of more complex parts of the robot, like the motor and sensor models, often does not lead to behaviors similar to the physical robot. This is due to the high dependencies between parameters. Therefore, entire sets of parameters, or even different model implementations, have to be found that produce a similar behavior, but have significant modeling differences. During evolution each controller then can be evaluated with each of the given model parameter sets.

Model rotation can be used during the entire evolution. This avoids partial solutions that are too dependent on a specific motor model. On the other hand, model rotation results in a much higher number of evaluations and slows down the evolution process.

Therefore it is often useful to start without or with a limited model rotation and do a separate final evolution with full model rotation to optimize the results for robustness.

Final Manual Adaption. The final step to transfer a controller to the physical robot is manual adaption. Even robust controllers often do not work out-of-the-box and require some fine-tuning. During this step, the hierarchical, modular structure of the modularized networks is advantageous. The structured networks are easier to understand, they assist in identifying the subnetworks responsible for specific functions, and help to isolate the synapses that have to be modified to optimize the performance of the controllers for the physical robot.

7.4.1 Modularization

For the transfer to the physical A-Series humanoid, the network in figure 7.16 was chosen because of its combined utilization of the legs, knees, feet, arms and hips. Figure 7.17 shows the motion sequence of the behavior. The network has been simplified by pruning all superfluous network structures, so that it is comparably easy to understand and manually adaptable if necessary. All constraints from the previous experiment have mostly been kept, so that modifications are still symmetric. The cloning constraints on the controller and filter modules have been relaxed so that only their structure is cloned. This allows a fine-optimization of the synaptic weighs of these modules depending on their actual location on the robot.

7.4.2 Parameter Settings

The evolution parameters have been chosen to allow only small changes of synaptic weights and bias terms. The structure modification operators have been disabled to prevent major changes of the evolved controllers. Table 7.3 lists the settings.

Operator	**Parameter**	**Setting**
General	Population Size	200
	Max Simulation Steps per Trial	3000
	Number of Trials per Evaluation	10
Tournament Selection	Tournament Size	5
	Keep Best Parents (Elitist)	1
Change Bias	Change Probability	0.01
	Deviation	0.005
Change Weight	Change Probability	0.01
	Deviation	0.005

Table 7.3: Settings of the main evolution operators. All operators not mentioned here have been disabled.

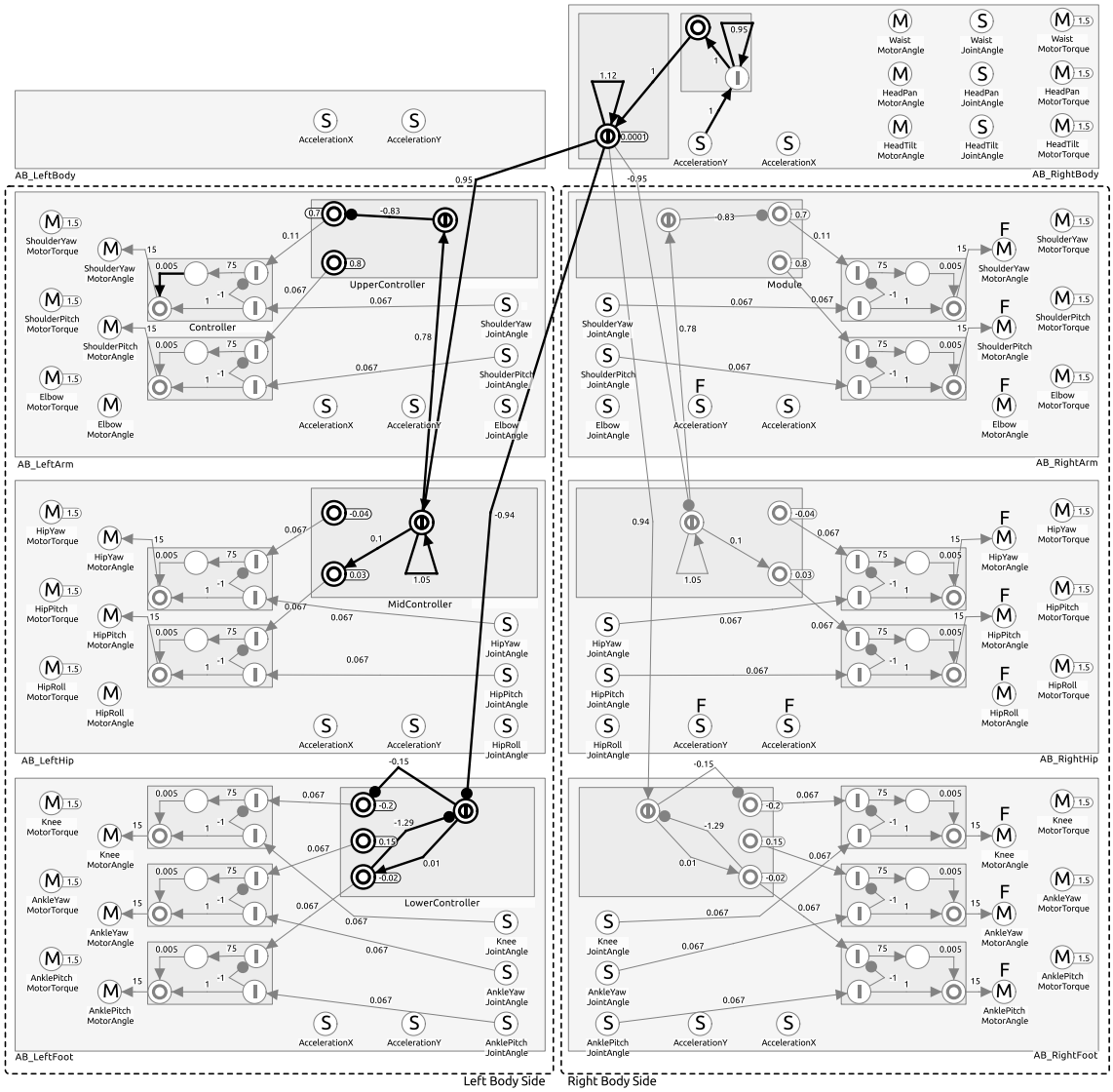


Figure 7.16: Network of a successful walker network using knees, arms and hip to accelerate the robot forwards. This simplified network has been used for the final robustness optimization. As can be seen, the remaining parameter space is very small.

Network: N: 158 S: 121 Parameters: B: 8 W: 15

7.4.3 Results

Figure 7.19 shows the time series of the walking controller running on the physical hardware. This network was optimized for robustness with model rotation. Five distinct parameter sets for the motors have been used during the optimization, each behaving closely to the physical machine for some test behaviors, but each differing in friction and control parameters. Figure 7.18 shows the fitness progression of the model rotation optimization.

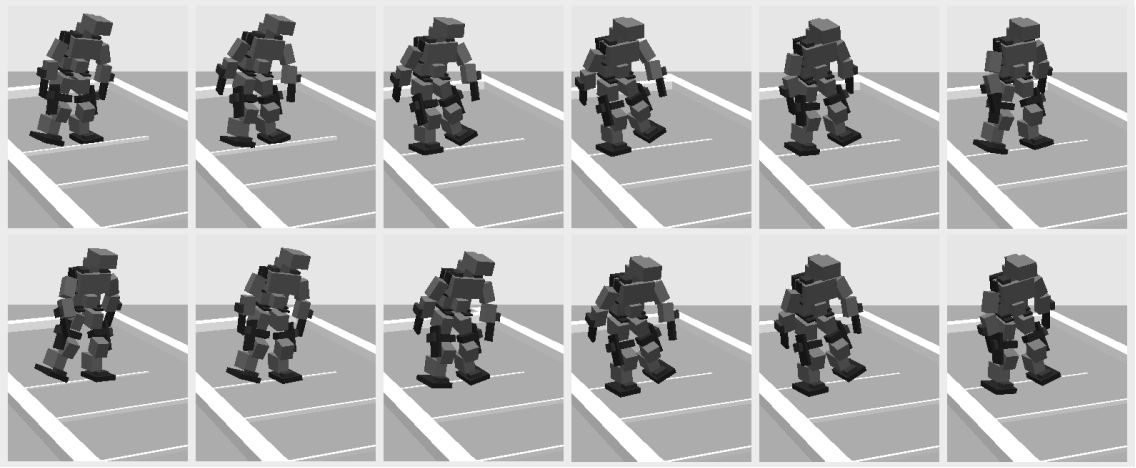


Figure 7.17: Time series of a robot controlled by the network in figure 7.16 that is used as initial network for the final optimization. The network uses the feet, knees, hips and arms to produce a quite dynamic looking behavior. The picture series shows every 150th simulation step.

As can be seen the fitness started low compared to the fitness achieved without model rotation. This is due to the fact that the minimal performance determines the fitness of a controller, therefore a single failing attempt reduces the fitness significantly. During evolution the fitness increased to a fitness level close to the one without model rotation. This indicates that the controllers became more robust with respect to small differences in the motor model.

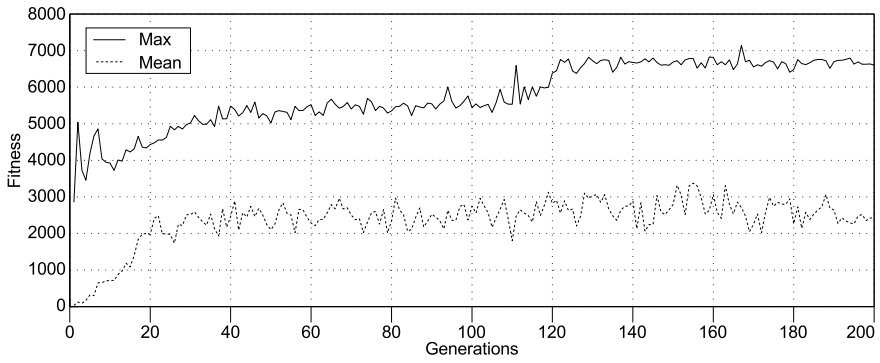


Figure 7.18: Fitness progress of a walking network during robustness evolution using model rotation.

For some solutions, adaptations were necessary due to the flexible body parts and the stronger friction on the ground. The legs had to be straddled wider and the upper part of the body had to be bent a bit forwards. Apart from these small changes of some bias terms, the behavior control performed well on the physical robot. Some solutions, like the one

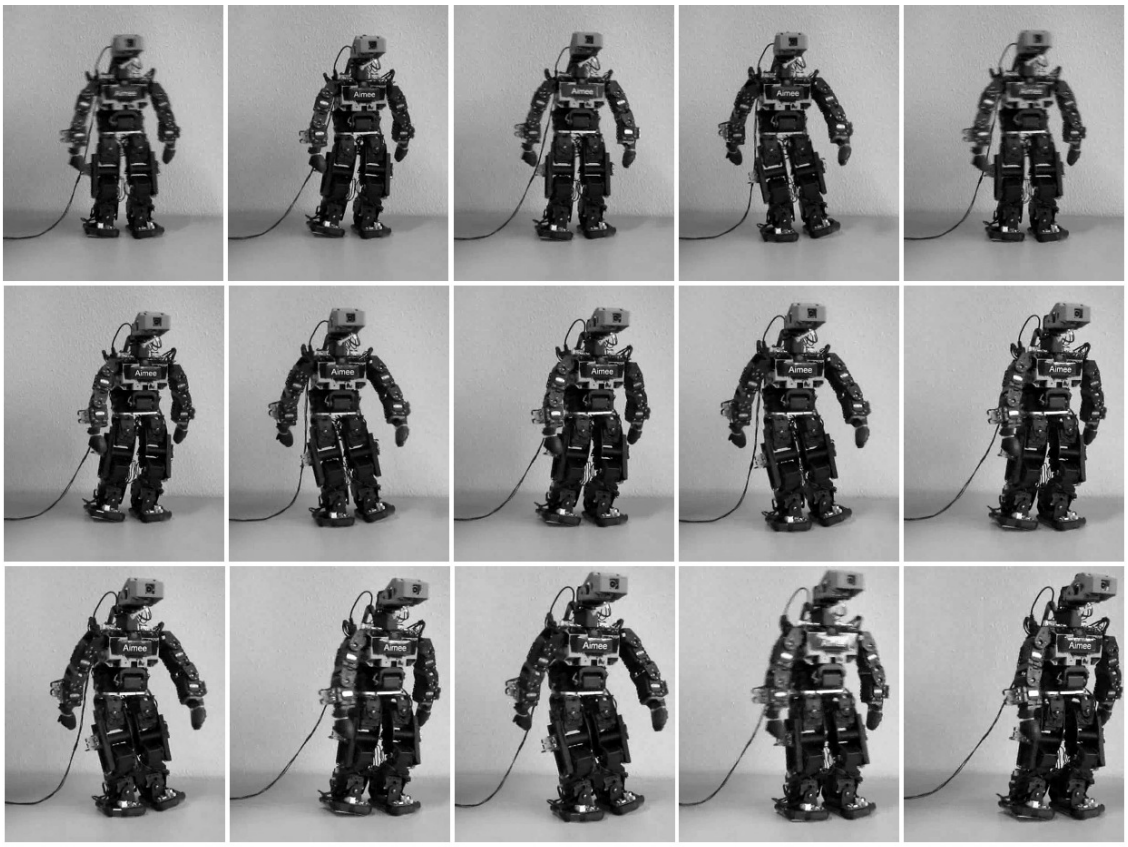


Figure 7.19: Time series of the physical robot controlled by the final walking network.

shown in figure 7.19, did not require any changes to the network at all and worked directly on the physical robot. Nevertheless, due to friction, elastic body parts and a missing vertical stabilization behavior, walking on the physical robot is by far not as stable as in simulation, where the simulated robot could walk hundreds of footsteps without falling over.

Chapter 8

Application: Evolving Controller Variants for a Closed-Chain Animat

8.1 Experiment Description

8.1.1 Approach

The experiments of this section focus on the application of *network shaping* (see section 2.4 on page 17), i.e. the restriction and guidance of the neuro-controller development on the network level. For this, a relatively simple animat with a nonetheless quite large number of sensors and motors was designed, that allows the demonstration of the ICONE shaping features.

The agent is composed of a variable number of plates that are connected with motor driven joints to a closed chain (see figure 8.1). Thus, each movement of a joint indirectly influences the other joints through the physical interaction of the connected plates.

Each motorized joint is controlled by two motor neurons: the first neuron controls the desired angular position of the motor and the second neuron controls the maximal torque applicable to reach the desired angular position. With this configuration, the torque neuron can be used to influence the passive flexibility of the chain, because the less torque on the joint, the easier it is to move the joint against its desired movement direction by its neighbor segments through the body.

In addition to the number of body segments, many other parameters of the animat can be adjusted to realize many different body variations. Among the parameters used for the described experiments are those that control the availability of certain sensors, the dimensions of the plates and the angular positions of the dead-stops of the joints. This allows the generation of animats with different sizes and different degrees of flexibility (see figure 8.1). The animat can be equipped with angular sensors in each joint, a force sensor on each plate, and – also per plate – acceleration sensors (3 axes) and gyroscope sensors (3 axes).

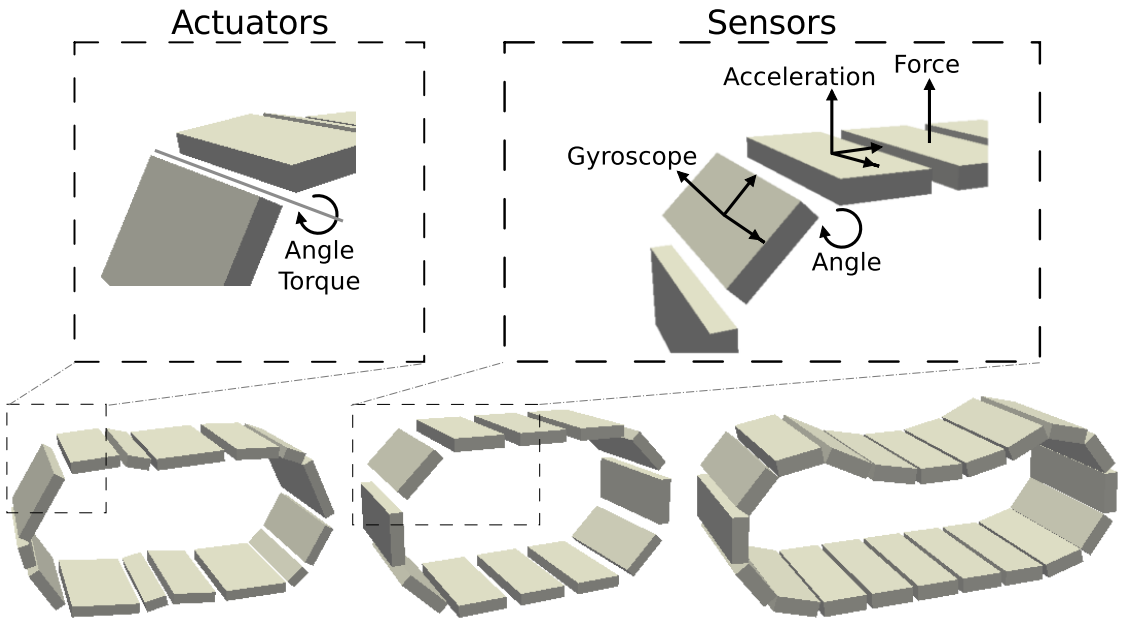
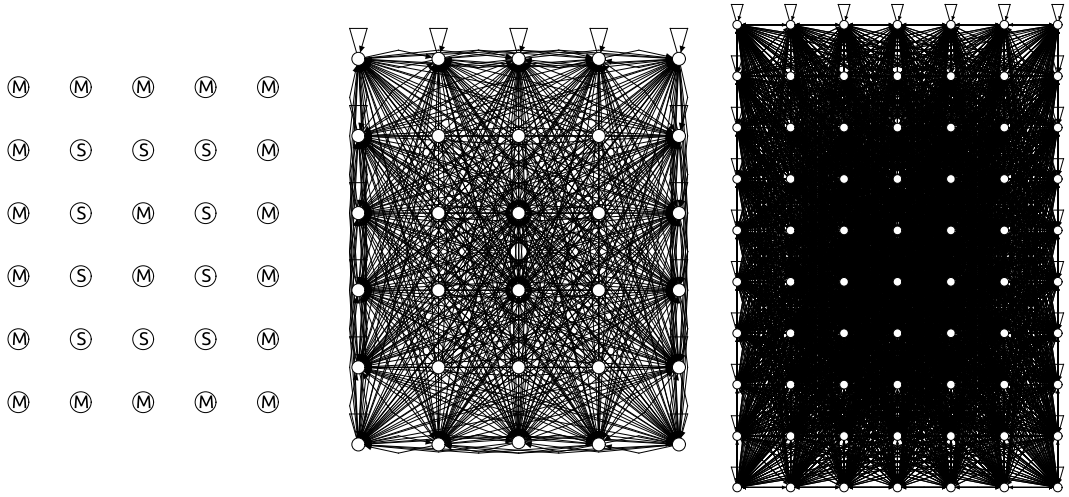


Figure 8.1: Different configurations of the smallest closed-chain animat (15, 12, 20 segments). The figure also shows the motors and sensors of the animat. Each shown motor and sensor type is available on each segment and can separately be enabled or disabled.

The networks of the animats belong to the mid-scale networks with – as used in these experiments – minimal network sizes (i.e. only counting input and output neurons) of approximately 30 (10 segments with angular sensors only) to over 150 (15 segments with full sensor equipment) neurons. The animat can be scaled to much larger configurations so that many more segments are possible, which further increases the number of neurons. For practical reasons, i.e. to reduce the computational effort for the physical simulation of such large animats, the animats have been limited to 15 segments for the experiments. However, for this regularly constructed animat, the ICONE method scales nicely with any number of segments.

The goal of the experiments of this section is the evolution of *different kinds* of control networks for forwards locomotion of the animat. In difference to a wheel-driven robot, where a single joint has to be actuated to result in a rolling behavior, the animat here has to coordinate the bending of all joints to result in a rotating forwards motion. A single disruptively operating joint can prevent this motion entirely, which makes the task very difficult for unconstrained, plain neuro-evolution.

Because the focus of this section is on the *network shaping*, rather than on the shaping through iterative experiments with varying objectives, a single fitness function is used throughout all experiments. Also, only a single experimental setting is used, with only a few configuration versions of the animat. Accordingly, the differences in the locomotion behaviors and the underlying neural control strategies for each configuration should primarily originate from the influences of the constraint masks used for the initial networks.



(a) Unconstrained Initial Network

(b) Search Space
Network: N: 30 S: 600
Parameters: B: 20 W: 600

(c) Search Space (+40 Neurons)
Network: N: 70 S: 4200
Parameters: B: 60 W: 4200

Figure 8.2: Search space of an unconstrained network for the closed-chain animat (10 segments, only using the angular sensor) . In (c) the search space of a network with 40 additional neurons is shown. That size is comparable to the networks evolved with the constrained networks in the forthcoming experiments.

As a rough comparison, evolution experiments with a minimal configuration using only 10 segments with only the angular sensors active, but with an unconstrained initial network (620 degrees of freedom, see figure 8.2) has been performed, in which not a single sufficiently working controller was found in over 100 evolution experiments. That experiment therefore is not shown here in detail, but should make clear that the domain of the experiment is already a challenging one.

The neuro-controllers evolved in the remainder of this chapter nicely demonstrate, how very different solutions can be evolved even with such a seemingly simple experiment, and how the ICONE method can be used to systematize this search for solution variants with network shaping. The experiments described in this thesis are only a subset of the successfully conducted and possible experiment variations. However, due to space reasons, only this subset can be shown here in detail.

8.1.2 Involved Techniques

The experiments in this section apply the ICONE techniques listed in table 8.1. Details on these techniques can be found in the corresponding sections given in the table.

Type	Technique	Section
Evolution	Interactive	5.2 p. 65
	Unsupervised for Variant Exploration	5.2 p. 65
Crossover	Active (if more than one module is evolved)	3.3.3 p. 39
Constraints	Cloning	4.1 p. 45
	Symmetry	4.2 p. 45
	Connection Symmetry	4.3 p. 46
	Network Equations	4.5 p. 47
	Prevent Connections	4.6 p. 48
	Enforce Directed Path	4.7 p. 48
	Enforce Connectivity Pattern	4.9 p. 49
	Restrict Weight and Bias Range	4.11 p. 50
	Synchronize Network Tags	4.12 p. 51
Extensions	Synaptic Pathways	6.1 p. 69
	Automatic Parameter Scheduling	6.5 p. 72
Property Tags	Protection	3.2.4 p. 31
	Mutation Control	3.2.4 p. 31
	Mutation Hints	3.2.4 p. 31
	Module Types	3.2.4 p. 32

Table 8.1: ICONE techniques involved in the closed-chain animat experiments.

8.1.3 Experiment

Simulation Environment. Most locomotion experiments with the closed-chain animat are evolved in a simulated hurdle track (figure 8.3). The hurdle track is framed by solid walls that keep the animat straight aligned. This is necessary to prevent the animat from tipping over and from leaving the hurdle track, when collisions with objects result in impacts and a slight change of the heading. Because of the missing joints and sensors needed for the agent to steer intentionally towards a specific direction, the agent has to be kept on track with such external measures. Also, once tipped over, the animat has no chance to get up again, so tipping over must be prevented. Collisions with the walls at the sides hereby do not terminate the evaluation, as was done in the humanoid walking experiment. Instead, they just passively push the animat back on the path. The reason for this strategy is, that such slight variations of the heading happen frequently as a result of the physics simulation and can affect all controllers. As a result, it is not possible to infer from wall collisions that a controller is improper. The wall at the back, however, can be used to terminate the evaluation at contact, because we are interested in a forwards movement and thus do not want to waste evaluation time on agents moving in the wrong direction.

The hurdle track is divided into four zones. In the first zone (1) a variable, not too short of a distance is kept free before the first hurdle is placed. This distance is required to support the development of a forwards locomotion as a first step in the evolution. Every controller with forwards locomotion, even those not able to overcome any obstacle, are still

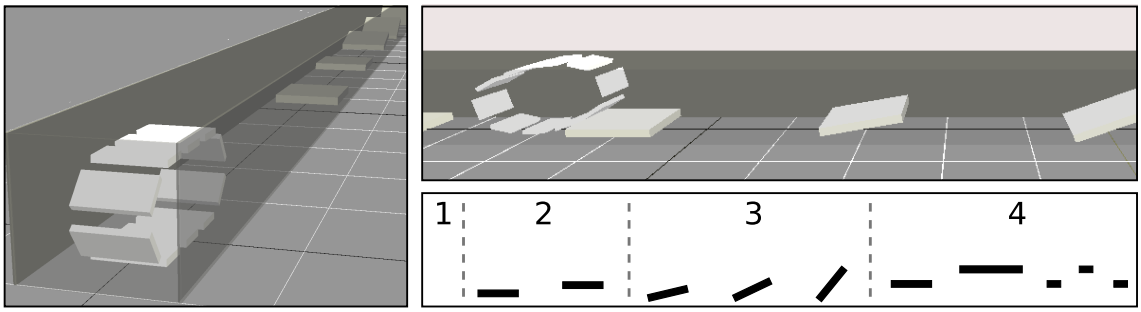


Figure 8.3: Simulation scenario for the locomotion experiments with the closed-chain animat. The placement of the hurdles and the four zones along the track are qualitatively shown in the lower right diagram.

considered interesting solutions. Consequently, this first part of the track must be long enough to allow at least one complete rotation of the animat. The hurdles themselves are arranged in an order with an (assumed) increasing difficulty to be overcome. Figure 8.3 shows such a hurdle track exemplarily in the lower right diagram. So, in the second zone (2) horizontal objects with increasing height have to be overcome, followed by ramps of different steepness (zone 3). The track is completed in the fourth zone (4) by objects of varied size, height and mutual distance (including tightly arranged obstacles). The exact positions and orientations can be varied during the evolution, so that each individual is evaluated in several, varied hurdle tracks to avoid an overfitting of the controllers with respect to a certain hurdle configuration.

The update rate of the physical simulation and of the neural network update have been set to 100 Hz to get a reasonably accurate behavior.

Fitness Function. For all experiments a single, parametrized fitness function is used. This experiment is primarily designed to get *rolling* behaviors, i.e. locomotion similar to caterpillar tracks. The fitness function therefore does not simply rate the distance traversed by the animat, because this includes also other forms of locomotion and known local, undesired optima, such as locomotion through very fast vibrations. The fitness function describes the desired plate-after-plate behavior more specifically: The fitness f at time step i is given by

$$f(i) = (1-\delta)g(i) + \delta \sum_{j=0}^{i-1} g(j) \quad , \delta \in [0,1] \tag{8.1}$$

$$g(i) = g(i-1) + \begin{cases} 0.1 & \text{if } max_i > max_{i-1} \\ -0.1 & \text{if } max_i < max_{i-1} \\ 0.0 & \text{else} \end{cases} \quad , g(0) = 0\,, \tag{8.2}$$

that is, $f(i)$ is the sum of all increments g up to the current step, where the ratio between current increment $g(i)$ and the increment history can be adjusted with parameter δ. This

allows the control of how important the speed of the agent is considered by the fitness function. If δ is high, then the fitness increases faster the earlier the agent increases the increment term. This increment $g(i)$ is based on the state of the positions of the animat's body parts. max_i denotes the index of the body part with the largest distance from the origin (in positive x axis) at time step i. If that index is higher than the previously furthest body part, then the increment is increased. If the index is lower, the increment is decreased. Otherwise no change takes place. The implementation of the fitness function considers that index 0 is the valid successor after the highest index and therefore is treated as being larger than the maximal index.

This fitness function may seem to have the disadvantage – compared to measuring the traversed distance – that also rolling on the spot is rewarded, e.g. when an obstacle cannot be overcome and the animat is still doing its rolling motion. But this is intended, because rolling on the spot is considered to be better than just stopping at an obstacle. An animat that is trying over and over to overcome an obstacle clearly has a better chance to finally overcome that obstacle eventually in comparison to an agent that is completely stopped by a hurdle. As the experiments demonstrate, this fitness function indeed helps to find very active animats that almost seem eager to overcome every obstacle by undertaking attempt after attempt, sometimes even with explicit run-ups in between.

8.1.4 Parameter Settings

For all experiments, similar evolution parameter settings have been chosen (see table 8.2). The parameters for interactive evolutions are listed as ranges, from which the settings have been taken as required. Most of the evolutions, however, have been done with unsupervised evolution runs to find variants of networks. In most of these evolution runs, the *parameter scheduling* extension (see section 6.5) has been used to change the parameter settings at fixed states of the evolution: the first parameter set (*init*) is used for the initial generation, the second set (*main*) starting with the second generation, and the third (*longrun*) starting with a high generation (depending on the experiment, e.g. the 150th generation). The significant increase of the fitness at that generation in many experiments is caused by an increase of the number of steps per evaluation try during this setting change. The settings of the different parameter sets are given in the last column of the table as slash-separated values. Evolutions have been stopped automatically when the fitness did not increase for more than 50 generations, therefore many evolutions did not reach the point where the *longrun* parameter set became active.

As the selection method, again an implementation of the standard *Tournament* selection (Miller and Goldberg 1995) was used.

A remark is necessary concerning the fitness development plots shown in the experiment descriptions of this chapter. Due to variations of the evolution parameter settings – such as the number of steps per try, the number of tries per individual and the weighting parameter δ of the fitness function – fitness values from different evolution experiments cannot be directly compared. A valid comparison of the gained fitness is only meaningful between several runs of the same evolution experiment. However, a qualitative comparison concerning the fitness development still is possible between different experiments.

Operator	Parameter	Interactive	Unsupervised
General	Population Size	[100, 200]	300/50/40
	Simulation Steps per Trial	[500, 3000]	500/1500/3000
	Trials per Evaluation	[2, 10]	1/3/6
Tournament Selection	Tournament Size	[3, 7]	8/4/4
	Keep Best Parents (Elitist)	1	10/1/1
Modular Crossover	Crossover Probability	0.5	0.5
	Probability per Module	0.5	0.5
Remove Neuron	Probability	[0, 0.005]	0.002
	Number of Removal Trials	[0, 2]	1
Remove Synapse	Probability	[0, 0.01]	0.005
	Number of Removal Trials	[0, 5]	5
Remove Bias	Probability	[0, 0.01]	0.004
	Number of Removal Trials	[0, 3]	2
Add Neuron	Probability	[0, 0.005]	0.01/0.002/0.002
	Number of Insertion Trials	[0, 2]	2/2/2
Add Synapse	Probability	[0, 0.02]	0.1/0.01/0.01
	Number of Insertion Trials	[0, 5]	10/2/1
	Init. Insertion Probability	[0.01, 0.1]	0.1/0.1/0.1
Add Bias	Probability	[0, 0.02]	0.02/0.005/0.005
	Number of Insertion Trials	[0, 3]	5/2/1
Initialize Synapses	Min	[-20, -1]	-5
	Max	[1 - 20]	5
Initialize Bias	Min	[-5, -1]	-2
	Max	[1 - 5]	2
Change Bias	Change Probability	[0.005, 0.025]	0.05/0.02/0.02
	Deviation	[0.005, 0.2]	0.5/0.1/0.1
	Reinitialization Probability	[0, 0.005]	0.001
Change Weight	Change Probability	[0.01, 0.2]	0.2/0.05/0.05
	Deviation	[0.005, 0.2]	0.2/0.1/0.05
	Reinitialization Probability	[0, 0.005]	0.005/0.002/0.002

Table 8.2: Settings of the main evolution operators. The settings are given as ranges in which the parameters have been varied during interactive evolution, and as fixed settings for the unsupervised evolution runs (*init*/*main*/*longrun*). The functions of the operators are listed in appendix B on page 177.

8.1.5 General Modularization

The initial networks of the agent are separated into modules, each encapsulating the neurons belonging to motors and sensors of one single body segment (figure 8.4). This reflects the heuristic, that local, meaningful network processing is more likely to emerge when interactions between locally related motors and sensors take place, in contrast to interactions between arbitrary, unrelated motors and sensors. In addition to this general modularization, each experiment has its own varied constrained mask, that is described along with the corresponding experiment. Usually, experiments use a clone constraint on all but the first module (used as a master module) to restrict the search space.

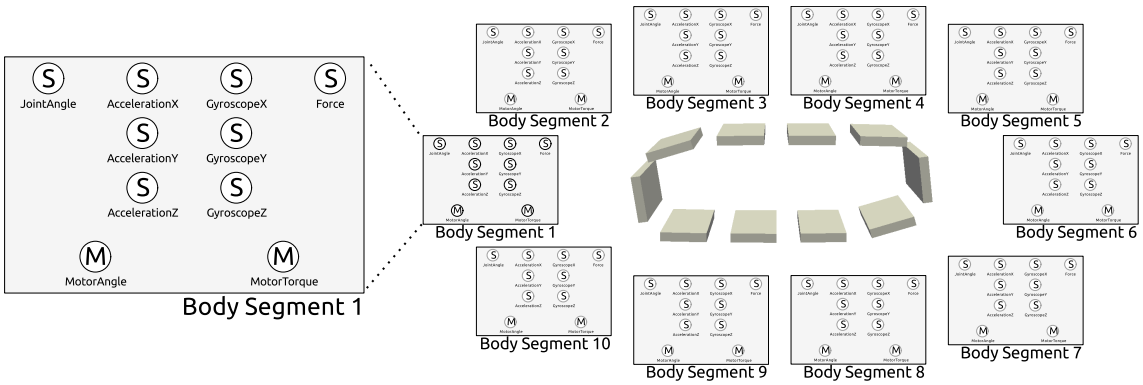


Figure 8.4: The general modularization for all experiments of this chapter. The number of modules depends on the number of segments used in the experiment (10, 12, 15 or 20). The large module at the left shows the first (master) module in detail.

Figure 8.4 shows the full sensor equipment of the animat. In the experiments, usually only a subset of these sensors is used. Hereby, the interface neurons can only be activated or deactivated in whole sets, i.e. the motor provides the `MotorJoint`, `MotorTorque` and `JointAngle` neurons, the force sensor a single `Force` neuron, and the acceleration and gyroscope sensors each three neurons for the measurements on all three axes (`AccelerationX-Z`, `GyroscopeX-Z`). So, when only one sensor of a set is needed, then the unwanted sensors of the corresponding set have to be disabled in the network using protection tags.

As a general practice, all neurons of the master module are additionally tagged to be treated like newly inserted neurons. This helps with the bootstrapping of the evolution, because newly inserted neurons have a different, usually much higher chance of being connected to the rest of the network (compare the *insert synapse* operator in appendix B, p. 181). This is done to initially connect a new neuron with the network to increase the probability, that the new neuron directly has an influence on the network. This is exactly what we also want for the neurons of the initial network, especially in the first generation, in which the creation of an initial pool of individuals rich in variants is desired.

8.1.6 Overview on Experiments

The rest of this chapter is structured as follows. The experiments are conducted in three series of experiment variants. For each series, only a subset of the performed experiments are described in detail to demonstrate the usage of the ICONE method. These shown neuro-controllers are the unpruned, unmodified networks as they have been evolved in the experiment. Hence, these controllers partially also contain superfluous network elements, that do not contribute to the behavior. These network elements have been kept to show the actually evolved network topologies and the hereby affected comprehensibility of the networks. The layouts of the controllers, however, have been adapted with the network editor to increase their readability.

Some of the more interesting resulting controllers are also briefly analyzed to show, that the evolved controllers significantly differ in their function and organization. However, it should be kept in mind, that the focus of this chapter is not the in-depth analysis of different neural control concepts, but merely the exemplification of the ICONE method, particularly with respect to network shaping. Therefore, the network analyses are kept superficial and do not cover all aspects of the neuro-dynamics.

The detail description of each controller also gives information about its complexity, in particular the number of neurons (N) and the number of synapses (S) of the *entire* network, and the number of actually mutable parameters of the network, i.e. bias terms (B) and weights (W). These mutable parameters do not include additional synapses and bias terms, that may be inserted during further evolution. That information is given for each experiment by a *search space plot*, showing the maximal network with all possible synapses and bias terms (not considering potential additional neurons). These plots provide information about the number of neurons and synapses of the entire network, and the maximal number of mutable network parameters. An example of such a plot is found in figure 8.6.

The first series of experiments (section 8.2) focuses on the effects of different interconnections between the segments of the animat. In the second series of experiments (section 8.3) the effects of different sensor sets and of their usage is examined exemplarily. The final series (section 8.4) then shows how different control paradigms can be induced to the networks using peripheral structures.

8.2 Interconnectivity Variations

The first series of experiments explore the influence of the inter-module connections between the main modules, that encapsulate the motors and sensors of each body segment. As a base for the experiments, the modularized network described in section 8.1.5 has been used. All main modules clone a single master module, so that only that single master module is part of the search space. Such a master module has to be evolved as part of the large network and cannot be evolved separately on its own, as it would be possible for a separate, local function (e.g. a neural PID controller or an oscillator (Pasemann et al. 2012)). This is, because the overall behavior is generated only as a result of the complete network and the body dynamics of the animat.

Over the course of the experiments, the connections between these modules have been varied in the initial networks. Starting with no connections between the main modules, a first experiment examines whether a forward-movement without any communication between the modules is – in principle – possible. The next experiment provides a simple interface between each module and its direct neighbors, specifically two synapses going to and coming from each of the two neighbors. This allows a module to use the signals of its neighbors in addition to its own sensors. A variant of this experiment is shown hereafter, where a communication between the adjacent modules is *enforced* by preventing the use of the own sensors. This demonstrates how signal pathways can be influenced with ICONE constraints. In the fourth experiment each module now can only communicate with its neighbors exactly two segments away. Because every second module is skipped now, two neurally independent circuits are formed, which requires a different kind of control. The last example demonstrates the use of the *connection symmetry* constraint by allowing the evolution of arbitrary, but symmetric interconnections between all modules.

8.2.1 Experiment 1: Prevented Segment Connections

The first experiment should show that a general locomotion behavior without any neural communication between the separate segments of the animat is possible. The necessary communication and coordination – if needed at all – has to come from the physical coupling of the body parts. This experiment uses an agent with 10 identical segments, each equipped only with the angular sensor of its joint. Additional experiments, allowing different sensors, have also been conducted, but are not shown here for space reasons. The experiment has been simplified because of the expected difficulty to generate a locomotion behavior without communication, therefore this experiment has been done without the hurdles.

Modularization. The network is composed of ten modules, each encapsulating the motor and sensor neurons of one segment. To prevent connections between these modules, no interface neurons have been defined. Synapses therefore can only be inserted between neurons of the same module. Recall, that in all experiments of this section, only a single master module is evolved, whereas all other modules are clones of that master module. The resulting search space is shown in figure 8.7(b) on page 114.

Results. In 67 evolution runs, 8 experiments resulted in sufficiently working neuro-controllers. Another 6 managed at least to move around, though they changed the moving direction from time to time. All others were either forming static shapes, or did useless motions on the spot.

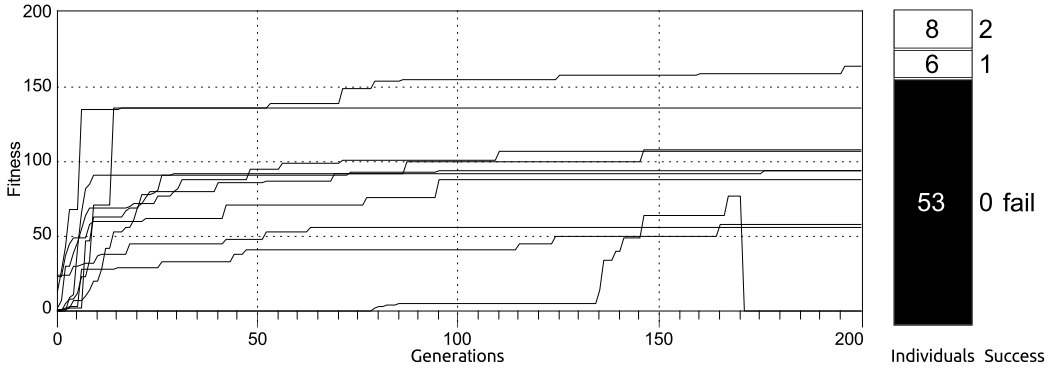


Figure 8.5: The fitness development of the 10 best evolution runs. The right diagram shows the distribution of controllers (0 failure, 1 locomotion with changing directions, 2 locomotion in one direction).

As a remark on the low success rate it should be mentioned, that in this experiment no walls have been used to prevent the animat from tipping over. So, many of the failed controllers tipped the animat over and then slipped over the ground doing fast movements, resulting in quite high fitness. In fact, the controller with the highest fitness is one of those. Therefore, only valid controllers have been considered in the fitness plot in figure 8.5.

All successfully evolved controllers produced a regular, pulsing behavior by alternating contraction and stretching of the body (figure 8.6). This is realized using torque modulation to allow a segment to either actively control a joint, or passively follow the forces applied by its neighboring segments. Stretching body parts actively in one place leads to a contraction of (passive) body parts in other places (or the other way round), which eventually leads to a synchronization of the overall motion through the body.

In the example shown in figure 8.7(a) each contraction is realized actively by setting the motor to a high torque and the desired angle to a pointed one. Once, a certainly pointed angle is reached, the torque is greatly reduced, so that the joint can passively be stretched by the pulling of the neighboring joints. The synchronization takes a few contractions, but once reached, the animat moves. The speed of the contractions is determined by a hysteresis effect, so that the contractions are not too fast or too slow, which both would end the movement.

Due to the missing neural feedback from the neighboring segments, the motion can be in both directions, backwards or forwards. The actual direction is determined by the starting conditions, but can – for some controllers – change spontaneously as a result of the interaction between the body and its environment.

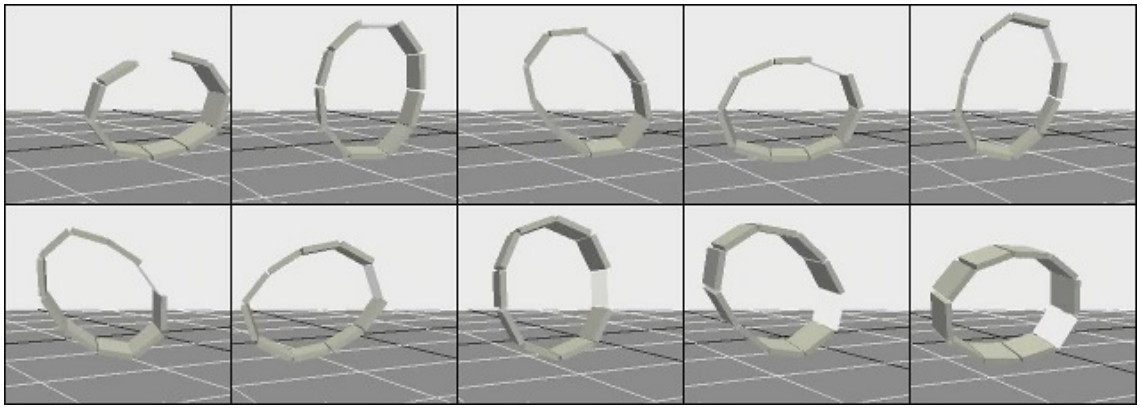
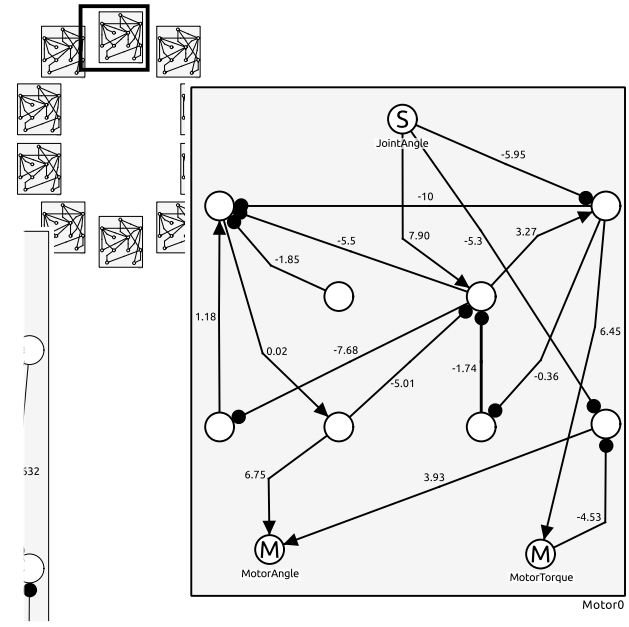
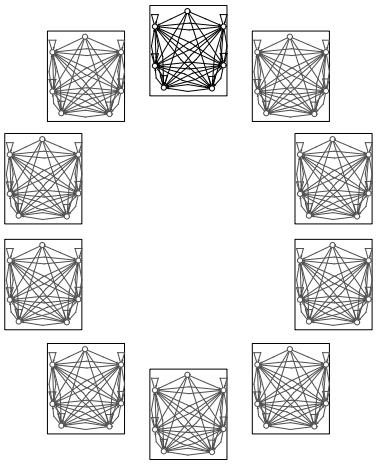


Figure 8.6: Motion sequence of the controller shown in figure 8.7. The pictures show every 20th simulation step.



(a) Network Details
Network: N: 110 S: 170
Parameters: B: 0 W: 17



(b) Search Space (+4 Neurons)
Network: N: 70 S: 420
Parameters: B: 6 W: 42

Figure 8.7: (a) Details of the locomotion controller without inter-module communication and (b) the maximal search space of the modularized network with 4 additional neurons per module.

8.2.2 Experiment 2: Direct Neighbor Communication

In this experiment the communication between direct neighbors is enabled. The communication is bidirectional, thus each module can influence its precursor and successor module.

As variants, this experiment can be further influenced with peripheral structures and constraints, for instance to allow only the communication in one direction or by specifying the number of connections between the modules. In this first experiment, the focus is only on the development of different kinds of forwards motions, thus no hurdles are involved. The agent is allowed to use its own sensors and additionally signals coming from the adjacent modules using 8 interface neurons per module (4 input neurons, 4 output neurons). A variation to this is shown in the next section (8.2.3), in which the hurdles then also had to be overcome. In that experiment, the agent is also forced to use the signals of adjacent modules and is prevented from using its own. This, in direct comparison, should illustrate how the outcome of the evolution can be directly biased with ICONE techniques.

Modularization. The initial network builds upon the previous experiment, so the motors and sensors of each segment are encapsulated together in one module per segment, using one module as master module and all other modules as clones of that master module. In addition, (4 - 8) interface neurons have been added to each module and tagged as input or output neurons to allow synaptic connections between the modules (see figure 8.8). In this first experiment, the direct connections to the neighbors have been added manually as peripheral structures. To reduce the search space, only one set of synapses between the modules are evolved, whereas all other inter-module synapses are directly derived from this set. For this, the *network equations* constraint has been used. The synapses of the evolvable synapses have been tagged with variable names and all dependent synapses have been tagged with the simple equation $w = var$ to calculate the synaptic weight w. var is the name used for the variable tag of the corresponding evolvable synapse.

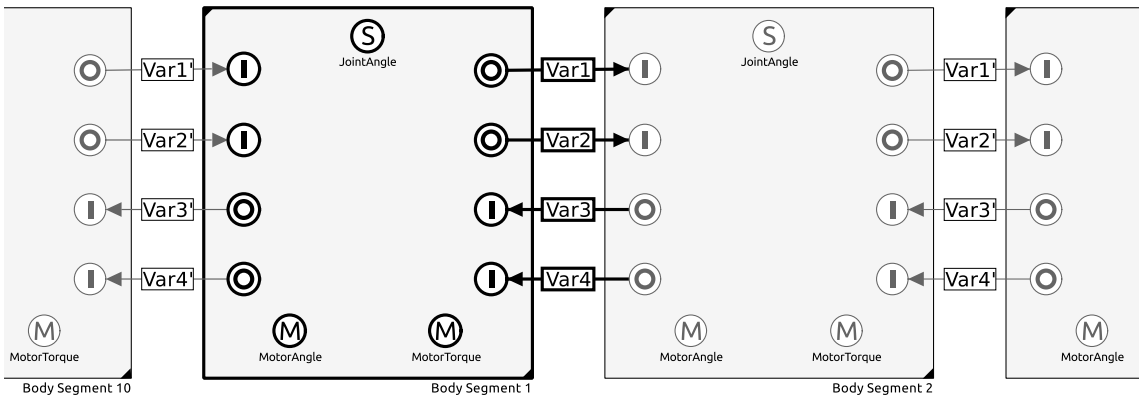


Figure 8.8: The modularization of the initial network for the direct neighbor communication experiment. The bold network elements are subject to mutations. All other network elements are dependent due to constraints and thus not part of the search space.

Results. In this experiment, 21 of 24 experiments resulted in successful forwards-rolling behaviors. The fitness development of the 10 best evolutions is shown in figure 8.9.

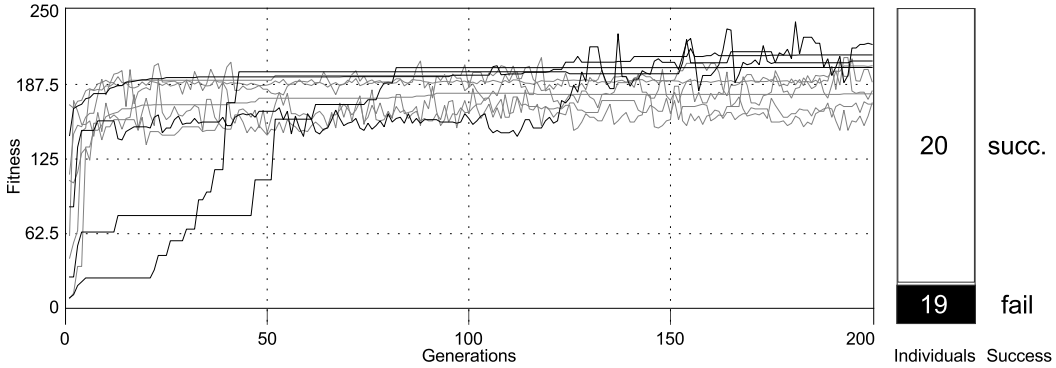


Figure 8.9: The fitness development of the 10 best evolution runs. For a better readability the best four evolution runs have been highlighted with a darker color. The right diagram shows the distribution of controllers with respect to their ability to move forwards.

Most evolved neuro-controllers produced an efficient locomotion, some of them astonishingly even able to pass the first two or three zones of the hurdle track when tested in such an environment. The most frequent behavior evolved in this experiment was a capsule-shaped, regular motion (see next section for such a behavior). However, some controllers also resulted in more organic-looking motions, such as the one shown in figure 8.10. Here, the agent moves indeed slower, with a less precisely formed shape, but still covers a large distance, even on the hurdle track. The underlying control approach hereby is interesting:

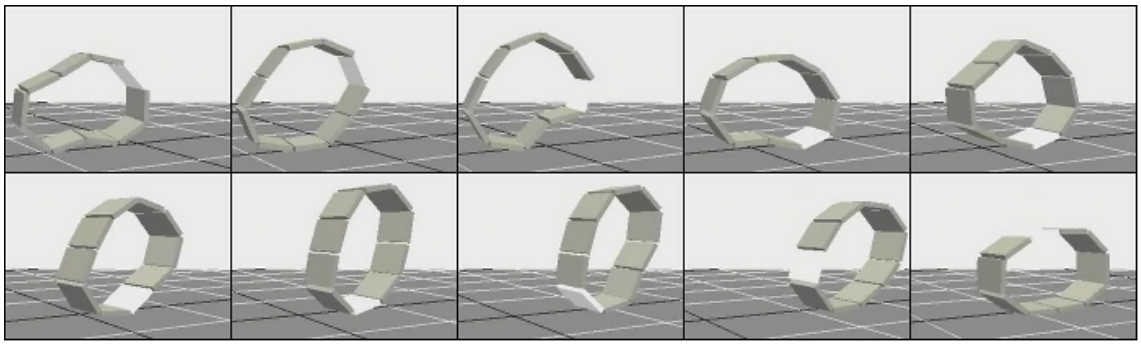
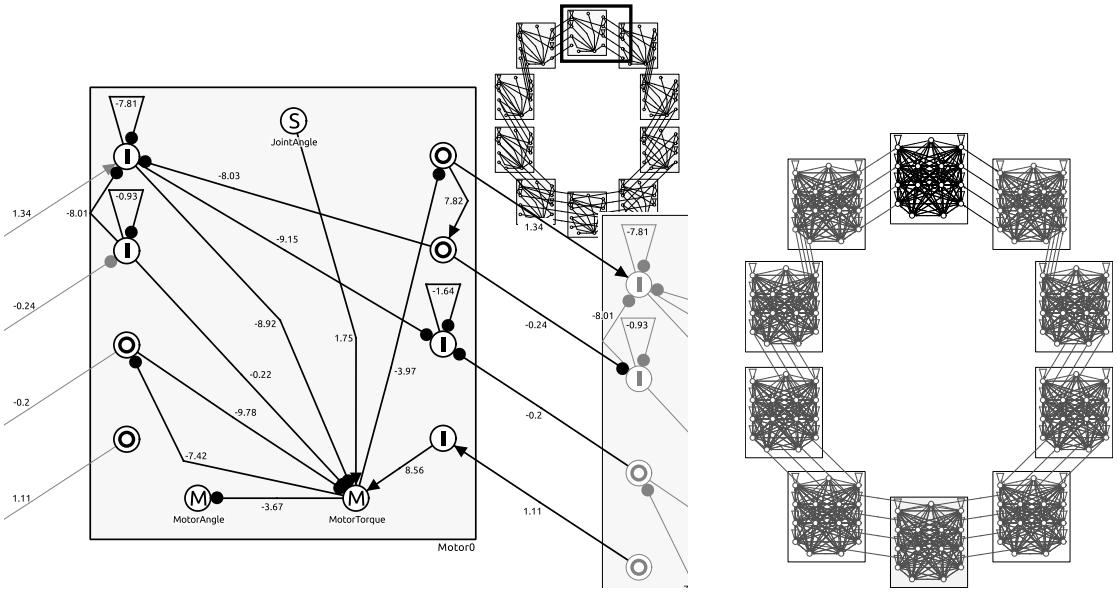


Figure 8.10: Motion sequence of the animat with the neuro-controller shown in figure 8.11(a). The picture sequence shows every 40th simulation step.

The dynamics of the network (figure 8.11(a)) is characterized by *modulated* period-two oscillators, i.e. period two oscillators, whose two states are modulated by the environment through the sensors.

Interestingly, the `MotorAngle` neuron is directly controlled by an inhibitory synapse coming from the `MotorTorque` neuron, which in turn oscillates. Most of the time that `MotorTorque` neuron oscillates between -1 and 1, which results – due to the integrative capabilities of the motors – in an effective medium torque applied to the motor. Because



(a) Network Details
Network: N: 110 S: 190
Parameters: B: 0 W: 19

(b) Search Space (+4 Neurons)
Network: N: 150 S: 2140
Parameters: B: 14 W: 214

Figure 8.11: (a) Details of the locomotion controller with a connection range of 1 and (b) the maximal search space of the modularized network with 4 additional neurons per module.

of the negative coupling, the `MotorAngle` neuron also oscillates between its two maximal settings -1 and 1. However, relevant for the motor control is only the desired angle that is active when torque is actually applied to the motor. This is always a positive activation. The motor therefore actively stretches. In a later phase, the behavior changes and the oscillation of the `MotorTorque` neuron shifts towards the positive activation domain (see figure 8.12, starting approx. at step 150 in the upper figure). At some point, the activation only oscillates in the positive domain, which leads to a constantly *negative* activation of the angular motor neuron. This results in a contraction of the joint. Accordingly, the oscillator realizes a *switch* between contraction and stretching. The contraction phase is entered, when the preceding module already is in that phase and its measured angle is low enough (is bent beyond a certain angle). Once entered, that phase is stable until the measured angle of the predecessor exceeds a certain angle (is stretched) again.

Interestingly, there is only one such contraction zone in the network, periodically traversing over the modules. So, if there is one end of the animat contracting and all other parts actively stretching, how can the second bending arise? This can be explained with the torque levels. As explained, the torque during the stretching phase is only approximately half of the strength during the contraction phase, because the torque neuron is positive for twice the time during that latter phase. The contraction is therefore much stronger than the stretching. The weakest point in the animat is in the end opposite to

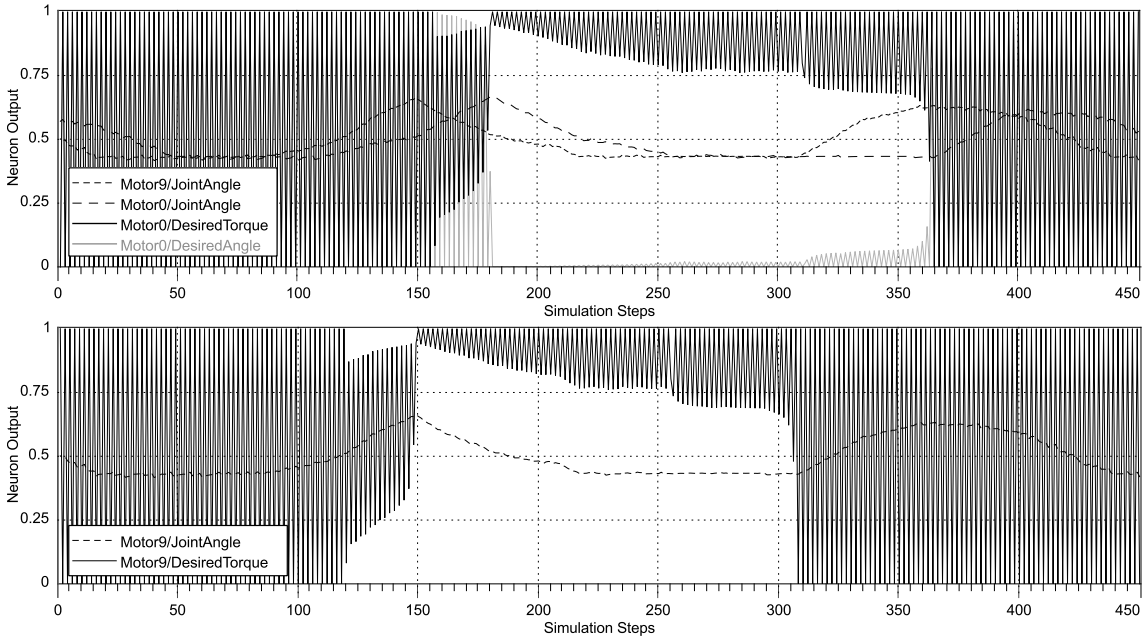


Figure 8.12: Activity plot of the sensors and motors of two successive segments. The upper figure shows the relation of torque and motor angle of a module and the angular sensor of its preceding segment. The lower figure shows the angular sensor and the torque neuron of the preceding segment synchronous to the upper figure to show the relation of the contraction phases.

the contraction area. Here, the segments are passively bent against their desire to stretch, because of the strong forces coming from the contraction area through the closed chain. Therefore, although each module contracts only once per animat rotation actively, there are two bending zones (see the angular sensors in figure 8.12). This also explains the more *organic*, *richer* motion, because the second bending area is not well defined and can vary depending on the forces coming from the environment.

8.2.3 Experiment 3: Enforced Neighbor Communication Variants

As a variation of the previous experiment, the impact of the direct communication between the neighboring modules has been further investigated. In this experiment, communication is enforced by *preventing* the direct use of the local sensors, so that local solutions are not possible any more.

Modularization. The initial network is derived from the previous experiment with all its constraints and peripheral structures. In addition an *Enforce-Directed-Path* constraint was added, configured to enforce all paths between sensors and motors to transit at least one module boundary, thus effectively preventing direct synaptic paths between the sensors and motors of a single module.

As further variations, four control variants have been examined: In the first experiment (1) the `MotorTorque` neuron was protected (so evolution could not add synapses to this neuron) and equipped with an initial, irremovable positive bias term. Evolution could still mutate that bias term, but the overall behavior had to be realized using the `MotorAngle` neuron only. In the other three experiments, the `MotorAngle` neuron was protected and also provided with a bias term. The behavior now had to be realized using only the `MotorTorque` neuron. The bias term of the angular motor neuron has been varied in the three experiments with fixed bias terms corresponding to (2) a *straight joint alignment*, (3) a *joint fully bent to the inner side* of the animat, and (4) *a joint fully bent to the outside* of the animat. This should show, how different (interesting) variants can systematically be explored with the constraint evolution approach.

All four experiments resulted in successfully working neuro-controller variants. However, for space reasons, only the results of one experiment is shown (variant 3), in which the `MotorAngle` neuron was protected and biased to bent the joint inwards.

Results. In 20 of 65 evolution runs, controllers were able to overcome all hurdles of the obstacle track. In 19 cases the behavior was not showing any forwards motion at all. In all other cases, the controllers could partially solve the task, e.g. by moving forwards (zone 1, 12 cases), passing the easy flat hurdles (zone 2, 7 cases) or by even overcoming the ramps (zone 3, 7 cases). The fitness of the 10 best networks is shown in figure 8.13.

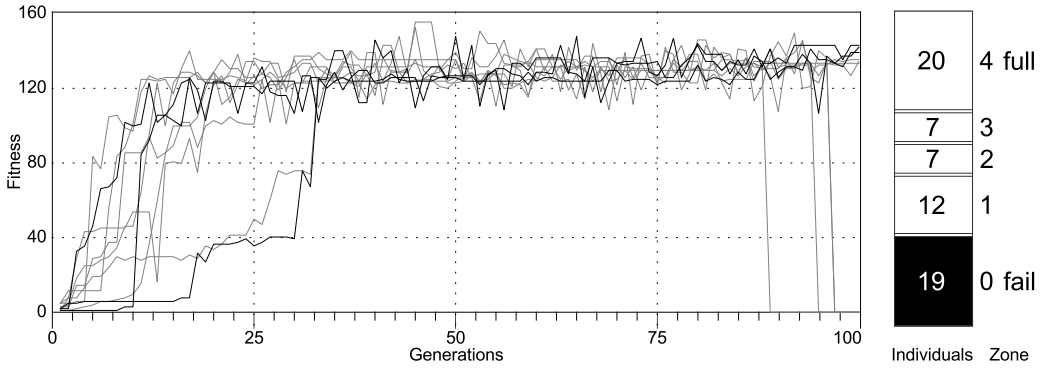


Figure 8.13: The fitness development of the 10 best evolution runs. For a better readability the best three evolution runs have been highlighted with a darker color. The right diagram shows the distribution of controllers with respect to their ability to overcome obstacles.

The evolved behaviors look often very similar: The animat is stretched to a capsule-like shape that does not change much during the movement (figure 8.14). A common strategy of the controller modules is to influence the relaxation of their subsequent joints. Such a network is shown in figure 8.15(a).

The basic desired angular position of all joints is – as it was fixed in the initial networks – the inwards bent state. Initially, the gravitation helps to flatten the body to the capsule-like shape, because with the gravitational support, the joints at the sides can bend easier

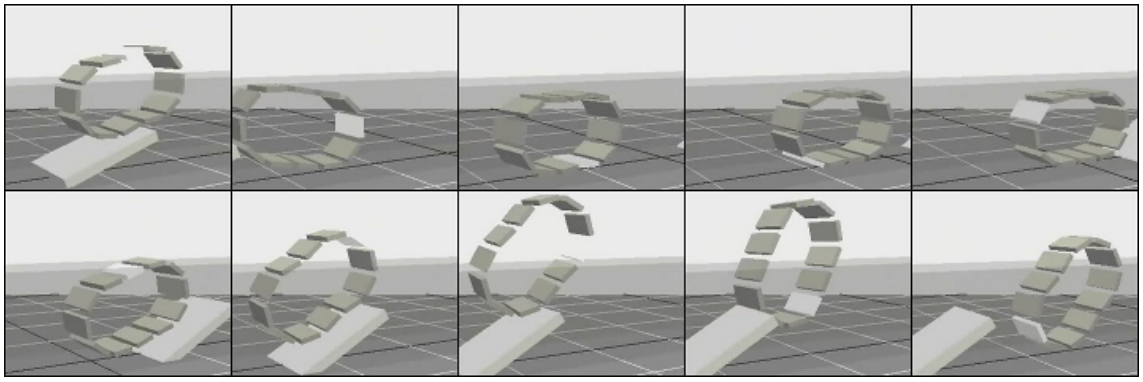
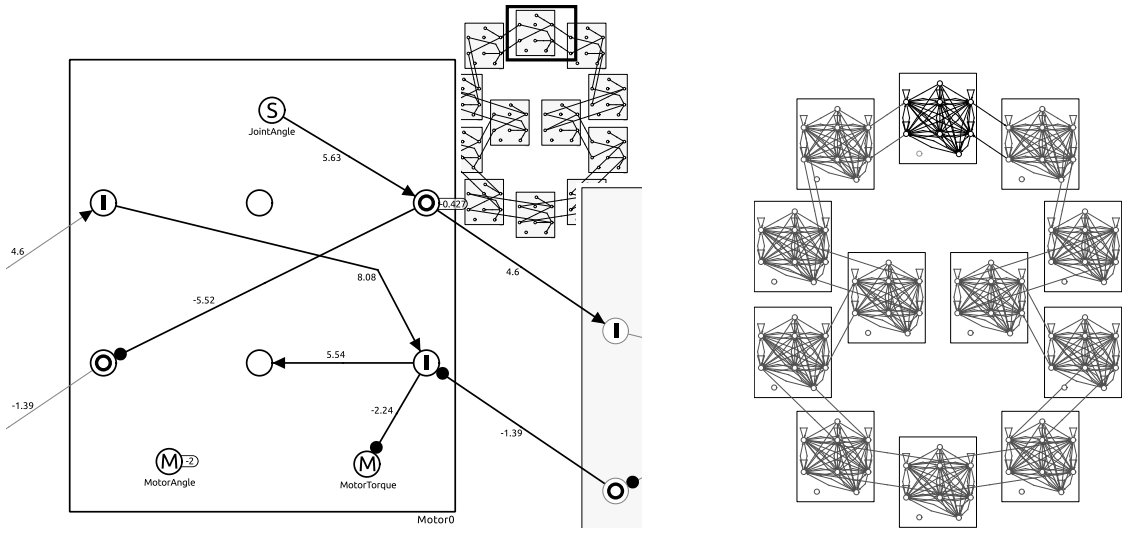


Figure 8.14: Motion sequence of the animat controlled by the network shown in figure 8.15(a). The picture sequence shows every 60th simulation step.



(a) Network Details
Network: N: 108 S: 84
Parameters: B: 2 W: 7

(b) Search Space (+2 Neurons)
Network: N: 108 S: 888
Parameters: B: 7 W: 58

Figure 8.15: (a) Details of the locomotion controller with an enforced connection range of 1 (variant 3) and (b) the maximal search space of the modularized network with 2 additional neurons per module.

than the ones at top and bottom. This starts the network dynamics that leads to the actual forwards motion:

When a joint gets stretched (initially via gravity) it strongly inhibits the `MotorTorque` neuron of its subsequent neighbor, which leads to a relaxation of that joint. A relaxed joint is passively stretched by the remaining actively bending segments through the force transmission of the chain and thus also starts to relax its successor. On the other side of

the animat, the still bent segments exhibit the `MotorTorque` neurons of their subsequent neighbors, that hereupon bend again.

So, in the lower front and the upper back of the animat, segments are relaxed, while at the upper front and the lower back the joints are contracted, resulting in a fast and stable forwards movement. The activations of the `MotorTorque` neurons of three subsequent modules in relation to the angular sensor of their middle module are shown in figure 8.16. The plot shows how the two relaxation phases (upper and lower) traverse over the modules.

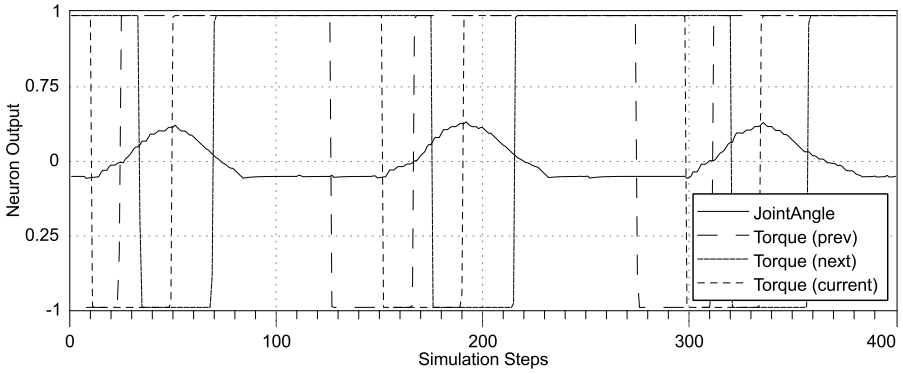


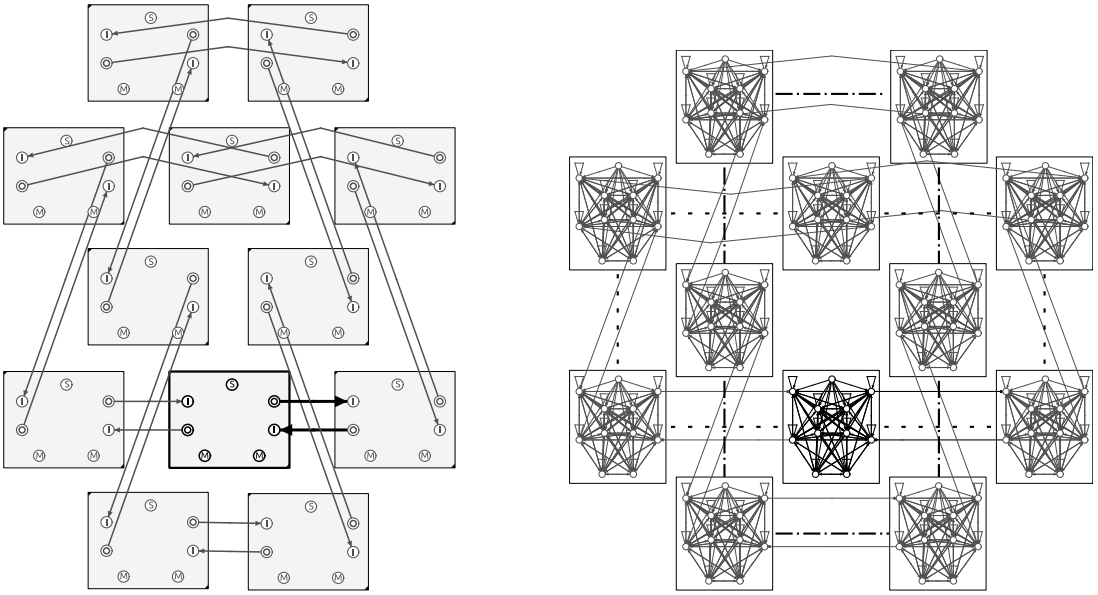
Figure 8.16: Plot of the angular sensor of a module and the torque neurons of that module and of its neighboring modules. The lower the angular sensor is, the more the joint is bent inwards. `MotorTorque` neurons with an activation of less than zero result in a relaxed joint that is not actively controlled.

8.2.4 Experiment 4: Connecting Every Second Segment

For an animat with 12 segments a variant with a communication range of two was realized. This configuration is interesting, because this creates two identical, but neurally independent circuits, each comprising six modules. The inter-module communication only takes place within these (neuro-dynamically independent) subnetworks and, of course, through the body.

Modularization. The initial network was almost identical to the previous one, but with predefined synapses not between directly neighboring modules, but instead connecting every second module (figure 8.17).

Results. Evolving controllers in this variation turned out to be more difficult for the evolution than the previous ones. Over one third of the experiments failed. And among the more successful ones, not a single controller could master the entire hurdle track. Figure 8.18 shows the fitness development of the 10 best experiments. The large fraction of controllers not being able to pass even the easy hurdles in the beginning seems to cohere with a fast hopping behavior that dominates the behavior approaches in this experiment.



(a) Network Modularization

(b) Search Space (+4 Neurons)
Network: N: 132 S: 1344
Parameters: B: 10 W: 112

Figure 8.17: (a) The modularization of the network, showing the interconnection pattern between the modules and (b) the maximal search space of the modularized network with 4 additional neurons per module. The figure also highlights the two independent circuits with dotted lines.

This behavior has problems to overcome obstacles, but seems to be a local optimum for this constraint mask.

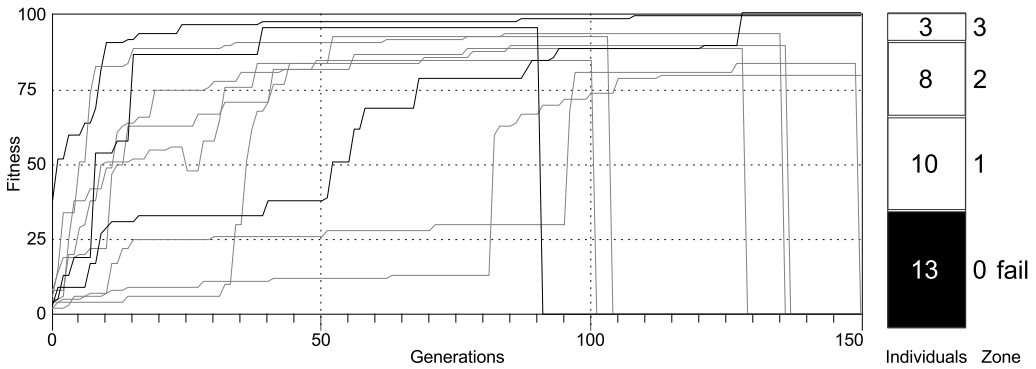
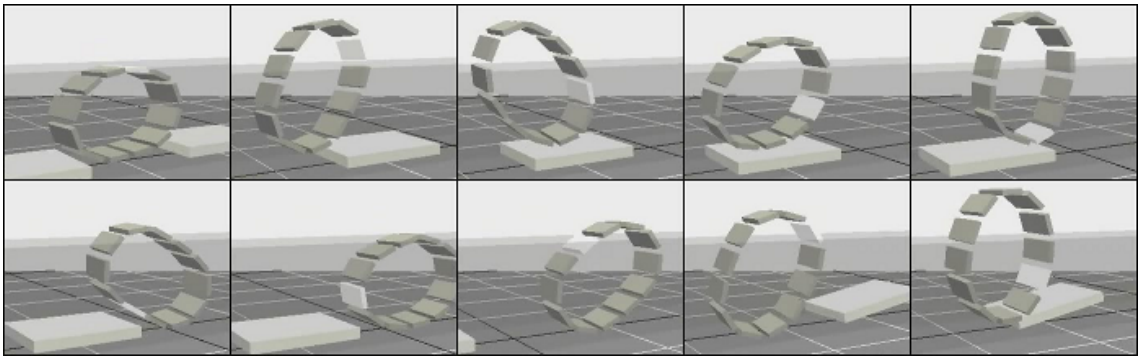
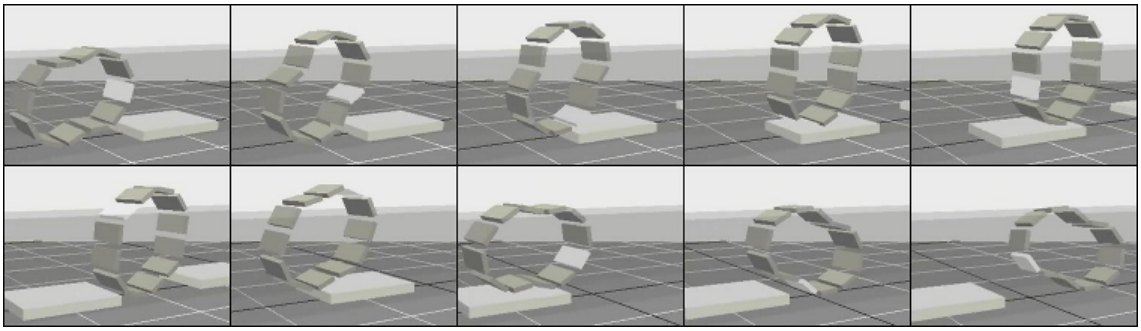


Figure 8.18: The fitness development of the 10 best evolution runs. For a better readability the best three evolution runs have been highlighted with a darker color. The right diagram shows the distribution of controllers with respect to their ability to overcome obstacles.

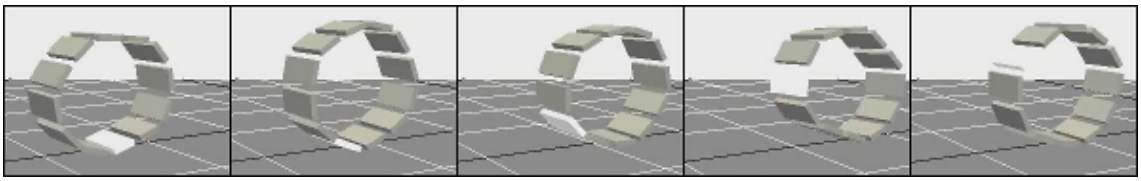
However, as hoped, also such controllers have been found, in which the separated circuits contribute to the overall behavior in an interesting way. The network shown in figure 8.20 is an example of such a network, that does not only exploit the independent circuits, but also implements three coexisting, differently looking behaviors (see figure 8.19). Depending on the starting conditions, the motion can either be an *eight-shaped rolling* motion, a *pulsing forwards motion* similar to the behaviors in the first experiment or a *wheel-shaped*, inefficient rolling.



(a) Bouncing Locomotion



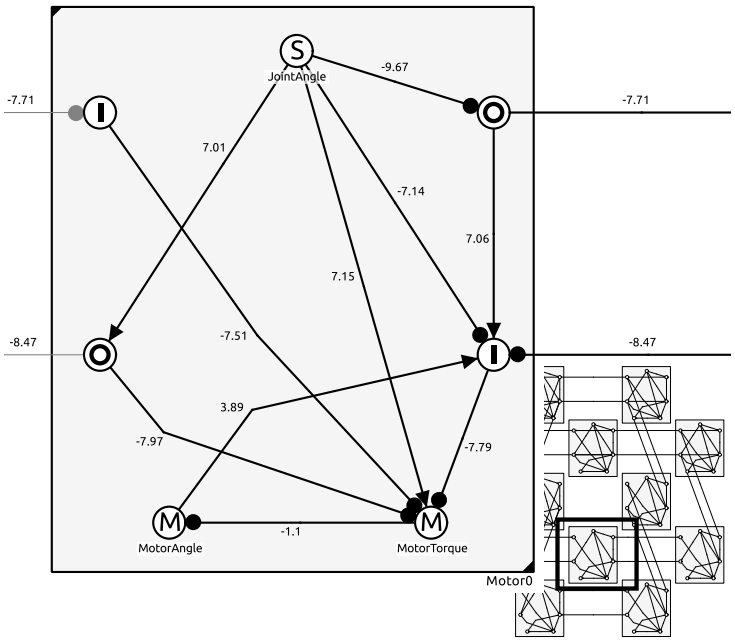
(b) Eight-shaped Rolling Locomotion



(c) Circle-shaped Rolling Locomotion

Figure 8.19: Motion sequences of the three coexisting locomotion behaviors of the neurocontroller shown in figure 8.20. The picture sequence shows every 40th simulation step.

The main behavior is the pulsing behavior that enables the animat to traverse all obstacles up to the steepest ramp. The behavior is generated by regular, very fast contractions of the segments, alternating with relatively short relaxation phases. Each separate circuit with its 6 segments provides 3 contraction phases. During each phase, both opposite



(a) Network Details
Network: N: 192 S: 336
Parameters: B: 0 W: 28

Figure 8.20: Details of the locomotion controller with a connection range of 2.

segments contract almost simultaneously, which makes sense because in the capsule-like shape the opposed segments should always do the same. The three contraction phases synchronize each other by forwards and backwards communication: The relaxation phase is entered when the predecessor joint is maximally stretched (angle sensor high) and the successor module is almost fully contracted. The contraction phase is entered when the predecessor module enters the relaxation phase. Here, successor and predecessor refer to the neighbors in the circuit, not in the body, i.e. every second module. As long as a module is in relax mode, it blocks its predecessor from entering that phase as well. Together, this leads to an exclusive phase locking, where each of the three relaxation phases cannot overlap (figure 8.21(a)). Because of the missing neural connections between the two circuits, the phase shift between the circuits can vary (upper graph in figure 8.21(a)) and is only synchronized through the body of the <animat.

With the proper starting conditions, the network shows a stable, eight-shaped behavior. The eight-shaped motion is created because the dynamics of one of the two independent circuits gets stuck in a fixed point, leading to a continuously active contraction of every second joint. The forwards motion now has to be realized by the six remaining modules, which astonishingly still works. Figure 8.21(b) shows the angular sensor of the active circuit. Interestingly, the phases are now much more precise compared to the quite fuzzy angular sensor output of the previous behavior, which may be caused by the missing

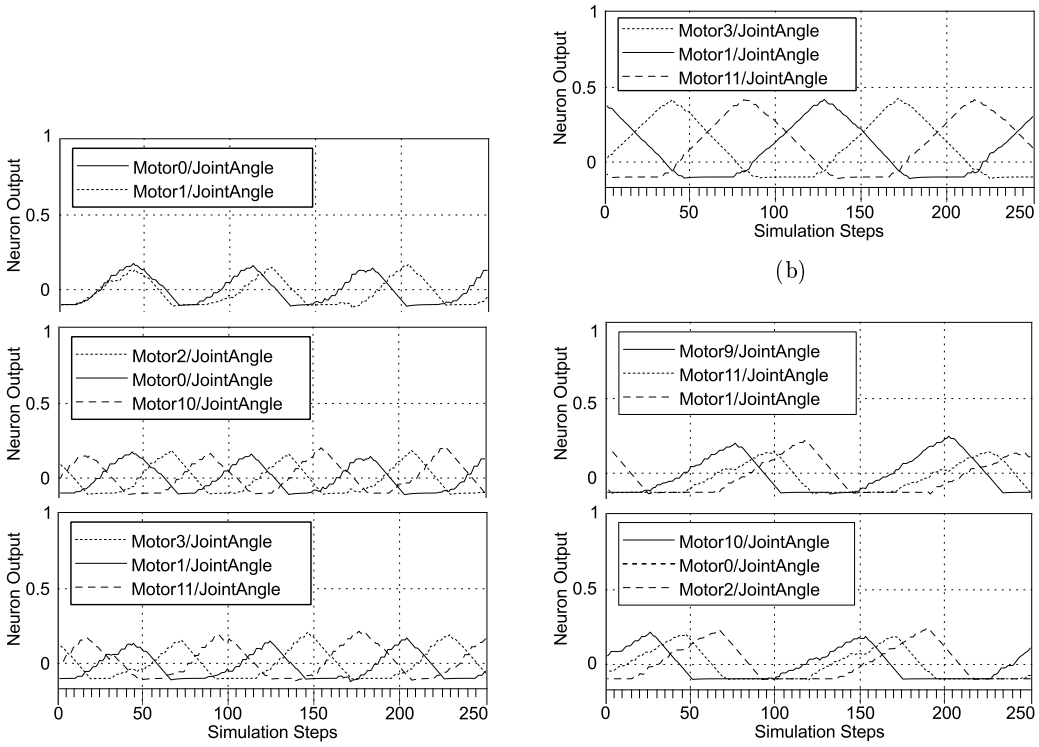


Figure 8.21: Comparisons of the contraction phases for each of the three coexisting behaviors. The activation of the angular sensor increases during the relaxation phase and decreases when the joint actively contracts. (a) Angular sensors of 3 successive modules of each of the two circuits (mid and lower plot). The plot at the top shows the variability of the phase shift between the two circuits. (b) The angular sensors of the eight-shaped behavior (only one active circuit) showing the more accurate, stronger stretching of the joints. (c) The bursts of the relaxation phases of the circle-shaped motion.

disruptions from the second, unsynchronized circuit. Also, the behavior is slower and the active segments have to be stretched much further to compensate for the continuously contracted segments of the fixed circuit. The behavior benefits from this and is even more efficient on the hurdle track than the main behavior.

The third behavior is a slightly pulsing circle with the capability to move forwards. However, this behavior only works on plain ground. Even the easiest obstacles are impassable. In this behavioral state, the relaxation phases are not interlocked as before, where each relaxation phase only started then the relaxation phase of the preceding segment was completed. Instead, the three relaxation phases follow shortly after each other as a burst, followed by a phase where all joints are contracting (figure 8.21(c)). Sometimes one of the circuits may enter a stable fixed point due to environment interactions, which leads to a

transition to the eight-shaped behavior. Thus, the third behavior has to be considered being unstable.

8.2.5 Experiment 5: Arbitrary Symmetric Connections

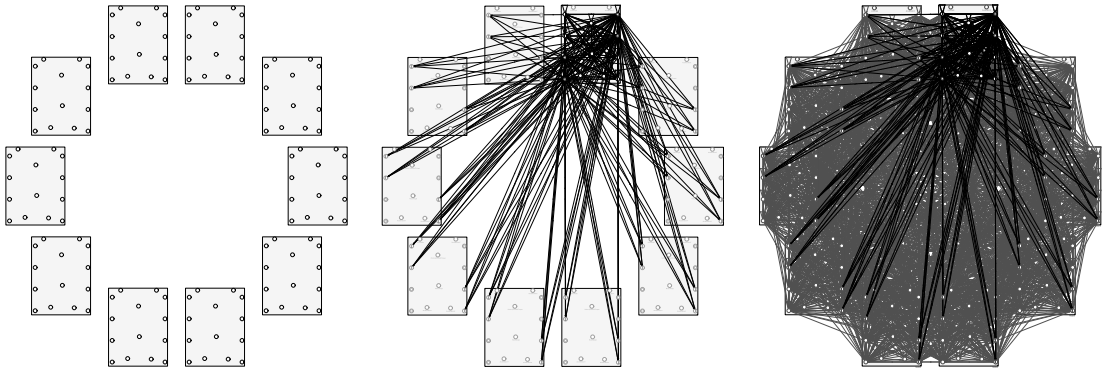
After exploring other predefined interconnection patterns between the modules (e.g. communication of range 3, of 4, of 1 and 2 combined, of 2 and 3 combined) it seems that the probability to successfully solve the locomotion problem decreases with the range of communication. Detailed results on these experiments are not shown here for space reasons.

In a next step, the inter-connections between the modules have been opened up for free evolution. However, to keep the search space still manageable, the connections still have to be symmetric, i.e. all external connections added to the master module also have to be added to any other module with the partner modules chosen relative to each module's position. So, adding a synapse from module 0 to 5 also adds synapses from module 1 to 6, 2 to 7, 3 to 8 and so forth. Hence, any inter-module connection added to the master module extends the network by 12 new, similar synapses. For this experiment, an additional set of sensors has been enabled for an animat configuration with 10 segments. The body segments of the animat should now use their acceleration sensors instead of their angular sensors to realize the locomotion behavior (compare also to the sensor variation experiments in section 8.3).

Modularization. The basic configuration of the initial network again provides one evolvable master module and 9 cloned modules. The acceleration sensors of the animat now have been enabled to allow new control approaches. The animat provides three acceleration sensors for all three dimensions. For this experiment, the z-axis has been protected (and herewith has been excluded from evolution), because accelerations lateral to the agent should not be considered. In that direction the agent has no degrees of freedom to influence the lateral motions, so a sensor in this direction is considered harmful or at least superfluous. To force the animat to use the acceleration sensors instead of its angular sensors, the `JointAngle` neurons have been protected as well. Each main module was configured to have four input and four output neurons, through which connections to other modules can be established. To enforce a symmetric interconnection pattern, the *connection symmetry* constrained was used. Properly configured, this constraint ensures that the same relative connection pattern present at the master module is also present for any other module, including the weights of the synapses. Figure 8.22 shows the initial network and the search space with all independently evolvable synapses 8.22(b). The maximally connected network is shown in figure 8.22(c).

Results. Despite the larger involved search space, evolutions with this constraint mask are quite successfully solving the problem. 7 of 24 controllers *fully* master the hurdle track, some of them with astonishing speed. Only 3 evolutions could not develop a forwards movement. Figure 8.23 shows the fitness progress for the 10 best evolution runs.

All successful controllers, including the ones that only partially overcome the obstacles, all show a behavior rich in variations. One reason is surely the use of the acceleration



(a) Initial Network (b) Evolvable Synapses (c) Search Space
Network: N: 140 S: 2640
Parameters: B: 10 W: 264

Figure 8.22: (a) The initial network with the *connection symmetry* constraint and a module interface of 4 input and 4 output neurons per module. (b) The maximal number of independent synapses. (c) The fully connected network showing the maximal search space.

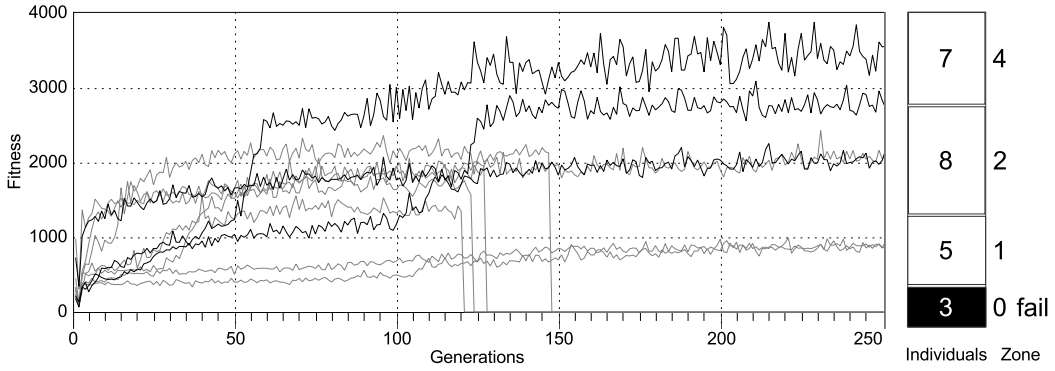
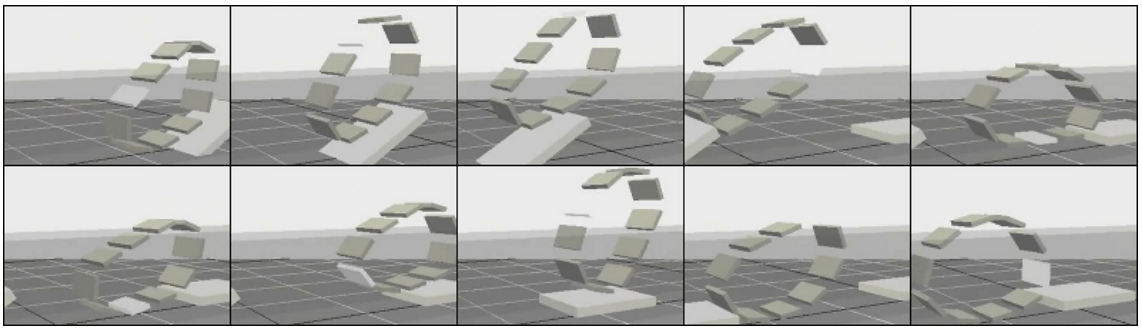


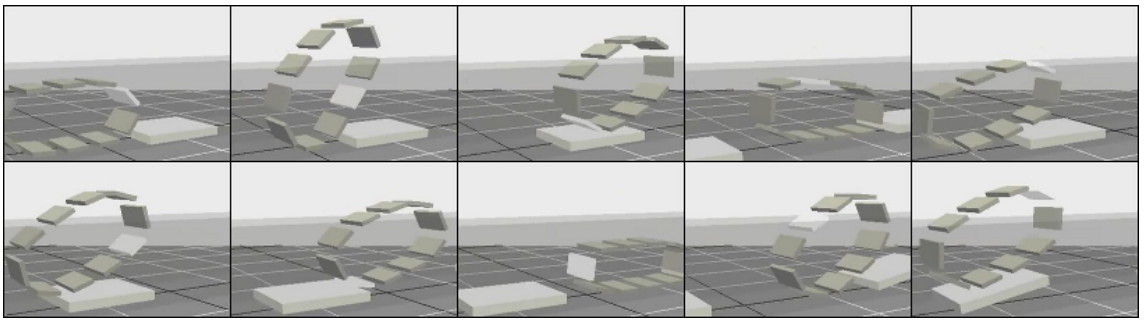
Figure 8.23: The fitness development of the 10 best evolution runs. For a better readability the best three evolution runs have been highlighted with a darker color. The right diagram shows the distribution of controllers with respect to their ability to overcome obstacles.

sensors, because these allow the animat to react much stronger on interactions between agent and environment, especially when approaching obstacles. But also the various ways of interaction between the modules plays a role.

Here, two evolved networks are shown, both very efficiently able to pass the entire hurdle track. The controller in figure 8.25(a) shows an interesting behavior, having the tendency to raise the front a bit to form a ramp, so that obstacles can be approached more easily (see figure 8.24(a)). The second controller, that is shown in figure 8.25(b), uses a different strategy. Instead of a continuous forwards movement, this controller moves the animat forwards in intermittent attempts, hereby lifting its front, which helps to approach the obstacles (figure 8.24(b)). During the irregular slowdowns, the body of the animat



(a) The *ramp-shaped* motion generated by the controller shown in figure 8.25(a). The picture series shows every 40th simulation step.



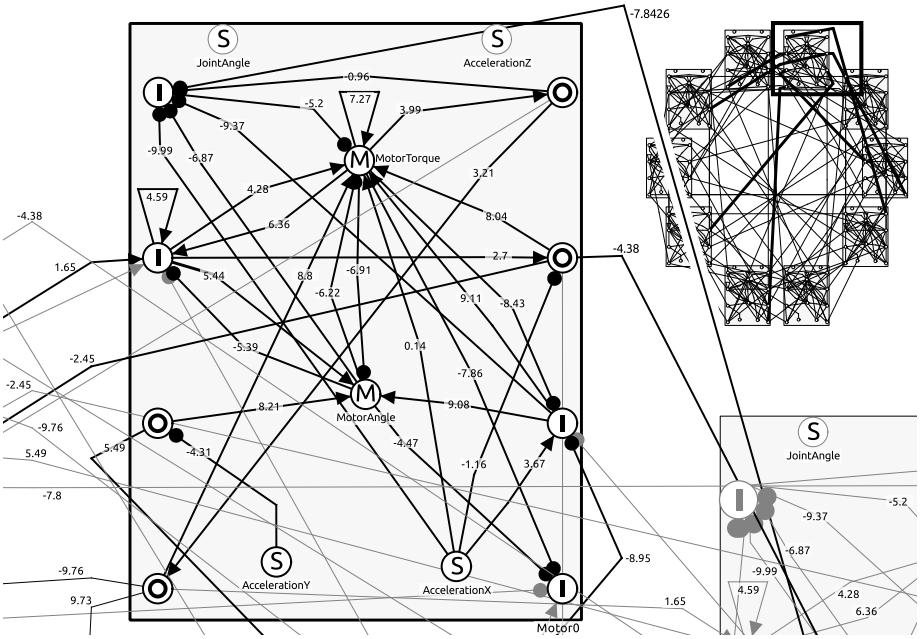
(b) The *intermittent moving* motion generated by the controller shown in figure 8.25(b). The picture series shows every 60th simulation step.

Figure 8.24: Motion sequences of two evolved neuro-controllers using arbitrary, symmetric module interconnections.

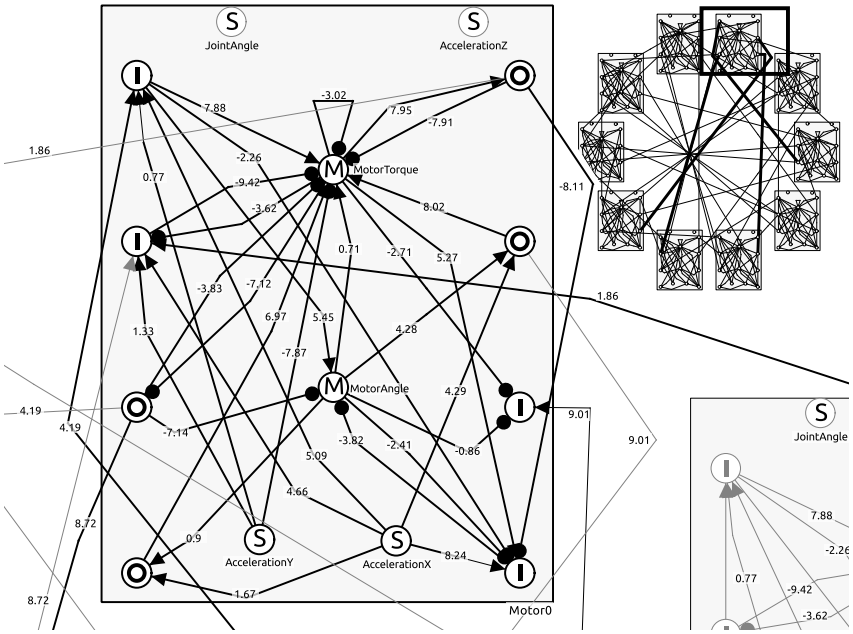
can settle back on the ground, so that inefficient loops are reduced. In fact, despite of the intermediate slowdowns, this controller is one of the fastest solutions to cross the entire hurdle track.

Both controllers have – like most controllers of this experiment – a quite complex topology, not to talk about the resulting neural dynamics. Such controllers cannot be analyzed roughly like the previous controllers and require an in-depth analysis of their neuro-dynamics, that is out of scope in the context of this thesis. However, even if these networks are not easy to analyze, this still nicely demonstrates that such neuro-dynamically non-trivial controllers can be found even in such a simple evolution experiment using constrained neuro-evolution.

Further Variants. Each of the experiments of this first series has only been shown with a single set of sensors. However, each experiment can also be repeated with a different sensor equipment instead of the angular sensors. This leads to other interesting solutions. The next section shows some of the sensor variations for networks allowing only a communication between directly neighboring segments. Surely, any of these experiments can be combined with the variations of this section to search for additional variants.



(a) Detail of the *Ramp-Shaped Motion* Controller.



(b) Detail of the *Intermittent Movement Motion* Controller.

Figure 8.25: Two evolved motion controllers using acceleration sensors and an unrestricted, symmetric inter-module communication. The thick lines in the miniature plots of the whole networks in the upper right corner of the plots denote the evolvable inter-module synapses belonging to the master module.

This illustrates, how many different combinations of sensors, motors, peripheral structures, connectivity paradigms, control theorems and the like are possible even in such a simple animat, and how these can be systematically explored with the ICONE method.

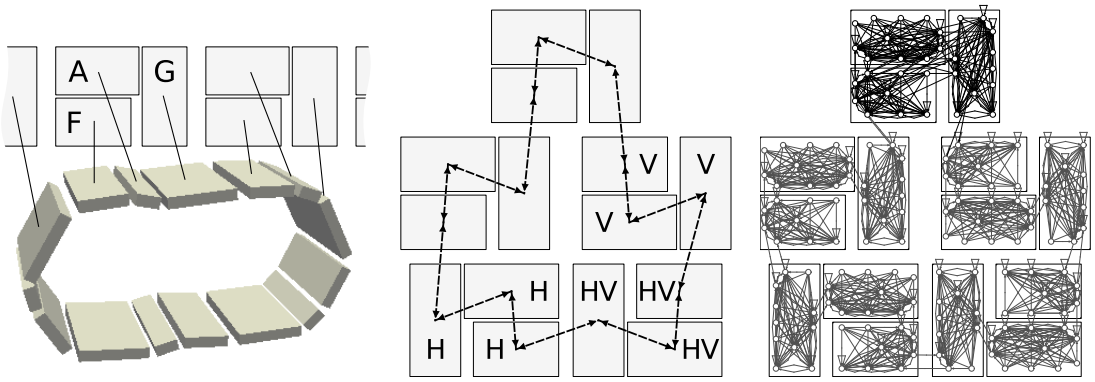
8.3 Sensor Variations

The second series of variation experiments explores the role of different sensors in a less uniform variant of the animat. A simple variation of the sensors has already been anticipated by the previous experiment. But now, the animat provides a richer, heterogeneous combination of different sensors that can be explored by evolution in different ways. The animat provides three different body segment types, that are arranged alternately (figure 8.26(a)). Each of the now 15 body segment (5 x 3) has, like in the first series of experiments, one motor-driven joint connecting the segment to its successor. Each motor also provides its corresponding angular sensor for its controlled joint. The actual difference between the segment types is their size and their additional sensor equipment. The first segment has dimensions comparable to the body parts of the first experiments and provides a force sensor, that measures the force applied perpendicularly to that segment. The second segment is smaller and equipped with an acceleration sensor measuring accelerations in three axes. The third segment is larger and has a gyroscope sensor, that provides information about the orientation of that segment in three axes (see figure 8.26(a)).

The focus of this series of experiments is to investigate, how the behaviors differ due to a specialization of the modules with respect to size and sensor equipment. Multiple successful experiments have been conducted and again, only a subset of these experiments can be shown here in detail due to space reasons. In this section, two of these experiments are shown to illustrate the general practice of how in such a configuration ICONE techniques may be used to shape the networks, to reduce the search space and to guide the search.

The first experiment starts with a network configuration, in which all sensors can be used arbitrarily in a symmetrized network. A variant of this experiment, that is not shown here, prevented the use of the angular sensors to force the animat to use the new sensors. A further class of variants was the restriction of the sensors to a single type (force, acceleration or gyroscope), that had to be used to control all segments, including those that now do not have any sensors. One of these experiments, in which only the five gyroscope sensors of the animat have been used for the locomotion, is described in detail as second experiment of this section. All variation experiments so far evolved successful behavior controllers that, due to the additional sensors, significantly differ from the controllers evolved in the previous experiments.

General Modularization. In difference to the first experiments, it is now not possible to evolve a single master module and clone all other modules. Because of the differences between the body segments, three master modules are needed now. Figure 8.26(c) shows the arrangement of the master modules (the top) and their cloned counterparts. To enhance the comprehensibility of the network and to point out its ring-shaped arrangement, some



(a) Arrangement of the modules with respect to the animat

(b) Order, interconnectivity and symmetries of the controller modules in the networks

(c) Search Space (+1 Neuron per Module)
Network: N: 155 S: 1135
Parameters: B: 21 W: 243

Figure 8.26: (a) Relation between the modules, the animat and the enabled sensor equipment (force sensors (F), acceleration sensors (A) and gyroscope (G)). (b) The module organization in the network, showing the symmetries with vertical (V), horizontal (H) or both (HV) axes for a better readability. (c) The maximal search space of the modularized network with one additional neuron per module.

modules use horizontal or vertical symmetry instead of cloning. This symmetry here only affects the layout, so it can also be omitted.

Each module was extended by four interface neurons through which connections to its neighbors can be created. For the connections between the modules, the *connection symmetry* constraint may be used, but now there are three different symmetries to consider, one between each neighboring segment type. In the two experiments shown in this thesis, the *network equation* constraint has been used instead, because here only six fixed connections between the master modules are used. In this configuration the *networks equations* constraint was favored because it does not affect the positions of the affected synapses (which increases the readability), but both constraints are valid here.

In addition to this general modularization, each experiment defines its own specific constraint mask, which is described separately for each experiment.

8.3.1 Experiment 6: Sensor-Rich Heterogeneous Controller

This first experiment of the sensor variation series does not restrict the new animat configuration further than described in the previous section. With this configuration the experiment examines, what kind of behaviors and motions develop without further guidance. Starting with a modularized initial network like the one described in the previous section, a search space is explored that – in principle – contains all outcomes of the later variation experiments. Thus, with this experiment, it should be tested, in how far the results of later experiments really are a matter of the constraint masks, or whether they would also have been found using the general modularization.

Since only one additional experiment is shown here for space reasons, it should already be noted at this point, that the variation experiments all came up with motions and network structures, that have not been found with this experiment. On the other hand, it should be kept in mind, that this experiment was only conducted 33 times and therefore cannot claim to have covered all likely outcomes. But a tendency towards dominant solutions could be identified.

Modularization. The modularization used for the initial network of this experiment is essentially the same as described in the previous section. In this first experiment, even potentially harmful sensors, like the lateral acceleration sensor (`AccelerationZ`) and the lateral gyroscope sensor (`GyroscopeZ`), have been kept unprotected to allow all kinds of results. As before, all neurons in the network have been marked to be treated like new neurons to foster a tighter initial interconnection of the neurons.

Results. Despite of the now larger search space (compare figure 8.26(c)) the experiment was quite successful. Only 7 experiments did entirely fail to generate a forwards locomotion. Five evolved controllers could even pass the entire hurdle track. The Figure 8.27 shows the fitness development of the 10 best evolution runs.

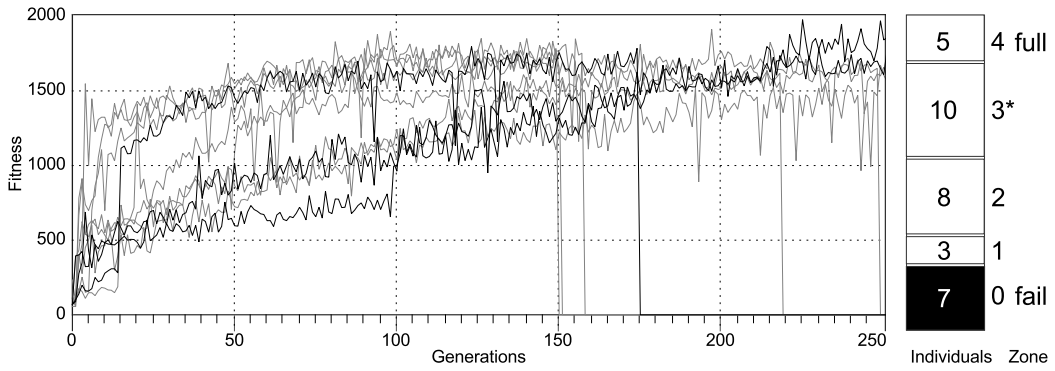


Figure 8.27: The fitness development of the 10 best evolution runs. For a better readability the best three evolution runs have been highlighted with a darker color. The right diagram shows the distribution of controllers with respect to their ability to overcome obstacles (3* stands for the ramp zone *excluding* the steepest ramp).

A noteworthy observation is that most controllers could not pass the steepest ramp of the third zone. Ten controllers could pass all hurdles except for the most difficult one. All of these controllers and many of those that got stuck after the second zone, belong to the same class of controllers, that seems to be the dominant local optima of this starting configuration: a star-shaped (or gearwheel-shaped) rolling motion (figure 8.28). This shape might be beneficial for the early hurdles, because the notches can gear into the edges of the hurdles to prevent to slip off the hurdle. However, this shape results in a quite compact circle, that can only be bent to a small extent when an obstacle has to be overcome that requires a shifting of the weight (see middle image of figure 8.28). The steeper the ramps

become, the more the animat has to be elongated to lean forwards in order to overcome the hurdle, which seems to be the reason why those controllers failed here. Only one controller of this kind could pass the entire track: a flattened version of the star-shaped motion.

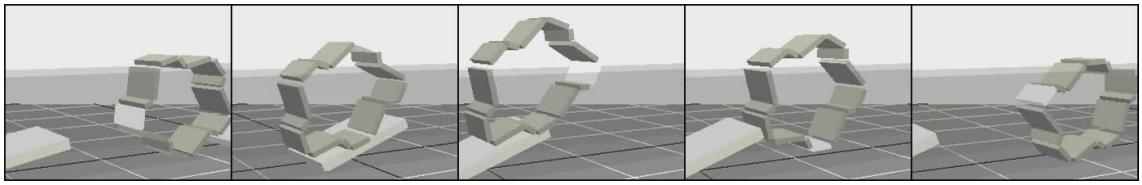


Figure 8.28: The most common behavior class of this experiment: the star-shaped rolling. The picture series shows every 60th simulation step.

A minority of evolved controllers produced different behaviors. While some were using almost 'classical' approaches (i.e. approaches already encountered qualitatively in previous experiments), there were also some novel approaches. The behavior shown in figure 8.29 is able to elongate and stand on its 'tiptoe' (one of the small segments), leading to an overturning of the animat, that can be used to propel it over an obstacle.

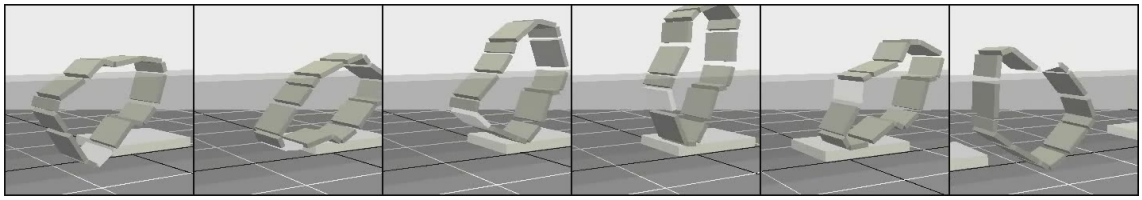


Figure 8.29: A controller that frequently raises on a small body part to overcome obstacles. The picture series shows every 60th simulation step.

A third interesting controller realizes a slackly, limp-looking motion, where the plates are bent in unspecific angles during the motions, forming a predominantly slender silhouette (figure 8.30). Nonetheless, the applied forces are quite high, so that the animat is almost catapulted forwards. The neuro-controller of this behavior is shown in figure 8.31. A rough analysis could not identify the underlying principles yet, but pruning experiments show, that all types of sensors are mandatory to preserve the behavior. Though, some of the sensor neurons (1 joint angle, 2 acceleration sensors and 2 gyroscope sensors) can be removed from the master modules, but the performance of the remaining behavior – although the basic motion is still observable – greatly suffers.

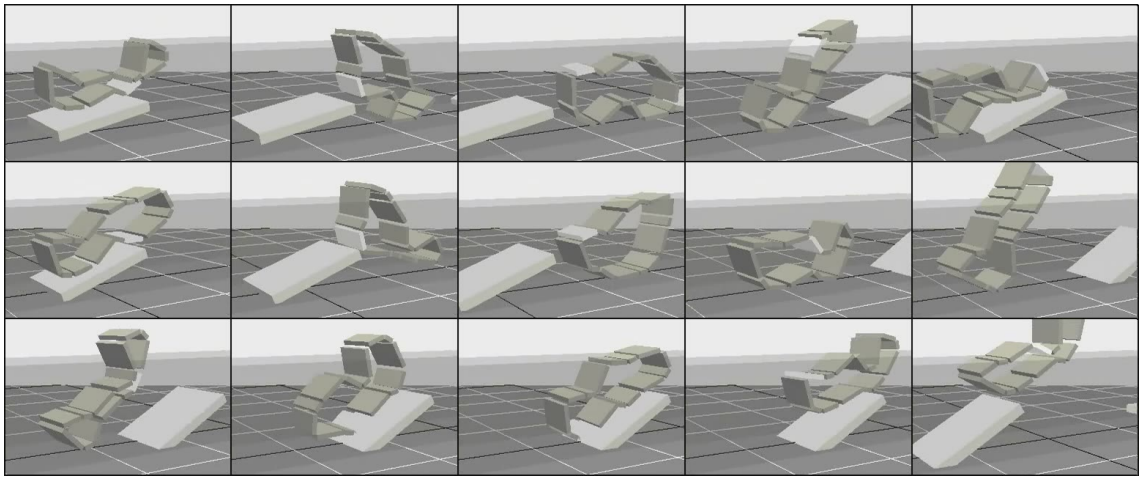
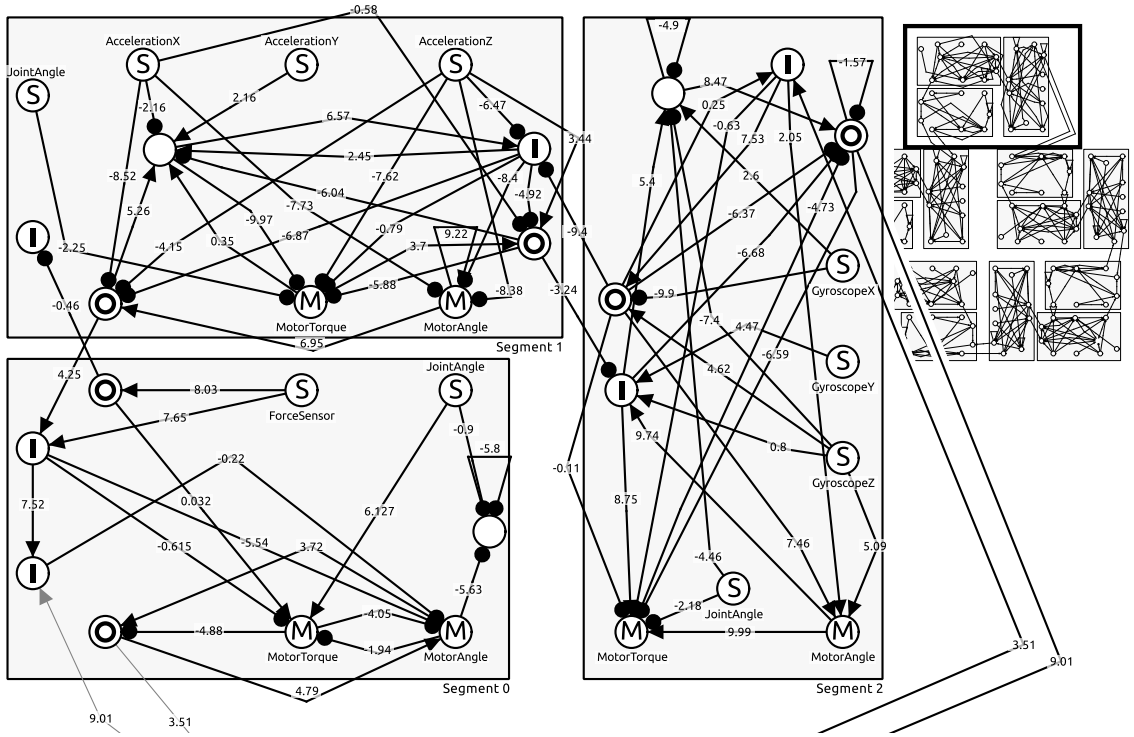


Figure 8.30: The slender motion controller, whose network details are shown in figure 8.31. The picture series shows every 60th simulation step.



(a) Network Details Network: N: 150 S: 360 Parameters: B: 0 W: 72

Figure 8.31: Details of the slender motion controller, that effectively uses all types of sensors available on the animat.

8.3.2 Experiment 7: A Gyroscope per Segment-Triplet

A mini-series of experiments was conducted to find controllers that use only one type of the available sensors. The animat configuration hereby still provides each sensor type only once per segment-triplet (except of the angle sensors), so that 15 joints of the animat have to be controlled based on the signals of only 5 sensors of the chosen type, located on 5 equally distributed segments of the animat.

One way to approach this is to create 5 large modules containing all sensors and all motors of each segment-triplet. Then, only one of these large master modules has to be evolved and the other modules could just be cloned. In this experiment, however, the distributed architecture of the segments with their limited communication pathways should be preserved. Because of their encapsulation and the required longer synaptic pathways, the motor neurons of the sensor-less modules are expected to be affected by sensor signals that are more strongly modified and have a greater delay. This may promote novel control concepts in which we are interested in here.

The experiment described in this section explores the usage of the gyroscope sensor in that manner, because that sensor was not used in one of the other experiments before. However, the experiment was conducted similarly for each of the sensor sets, each with successful results.

Modularization. In addition to the general modularization, that was described above, all sensors except of the `GyroscopeX` neuron have been protected to prevent evolution from using their signals. To ensure that all motor neurons of all evaluated controllers are at least slightly affected by that single sensor neuron, an *enforce directed path* constraint has been added. This constraint was configured to enforce a directed, arbitrary long synaptic path from the sensor neuron to each of the motor neurons of the module-triplet. That way, all evaluated controllers are guaranteed to have at least one directed chain of synapses going from the sensor neuron to each of the motor neurons. Controllers that do not fulfill that condition are not considered and removed, because such controllers are unlikely to control all joints in the desired way. However, as variants, this constraint may be relieved, e.g. to enforce a path only to the `MotorTorque` neurons or only to the `MotorAngle` neurons, and to permanently activate the other motor neuron type with an initially given bias term (like in the experiment of section 8.2.3). This would again bias the search towards a different direction with potentially novel solutions.

Results. Finding suitable controllers with this configuration was a difficult challenge for the evolution. Not a single neuro-controller has been found that could pass the entire hurdle track. Only two controllers were able to pass some (but not all) ramps in the third zone, all other controllers already failed before the first ramp or earlier. Six of the 19 evolved controllers did not even learn to persistently move forwards. The fitness progress of the 10 best evolution runs is shown in figure 8.32.

With these numbers at hand, it is all the more astonishing, that among the – with respect to the hurdle track – poor performing controllers not fewer than three interesting, novel control approaches have been found. Without a protecting constraint mask, such

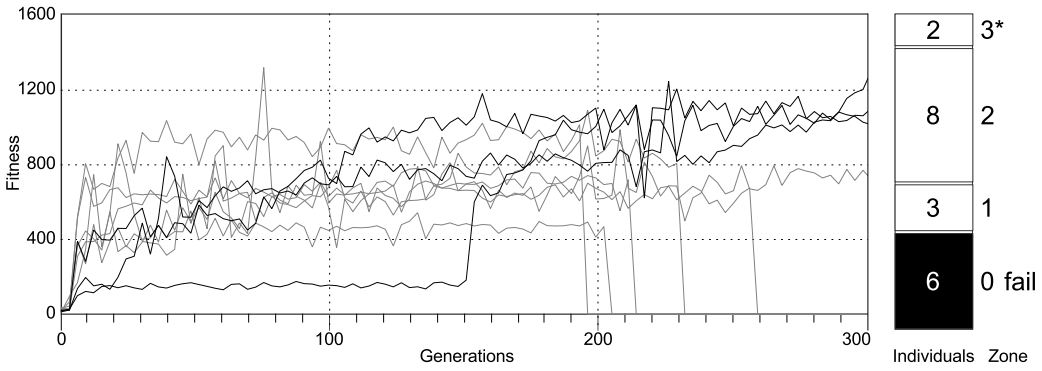


Figure 8.32: The fitness development of the 10 best evolution runs. For a better readability the best three evolution runs have been highlighted with a darker color. The right diagram shows the distribution of controllers with respect to their ability to overcome obstacles (3* stands for the ramp zone *excluding* the steepest ramp).

controllers would have been dominated by easier to evolve, better performing solutions, like the star-shaped motion of the previous experiment, and thus would very likely not have evolved.

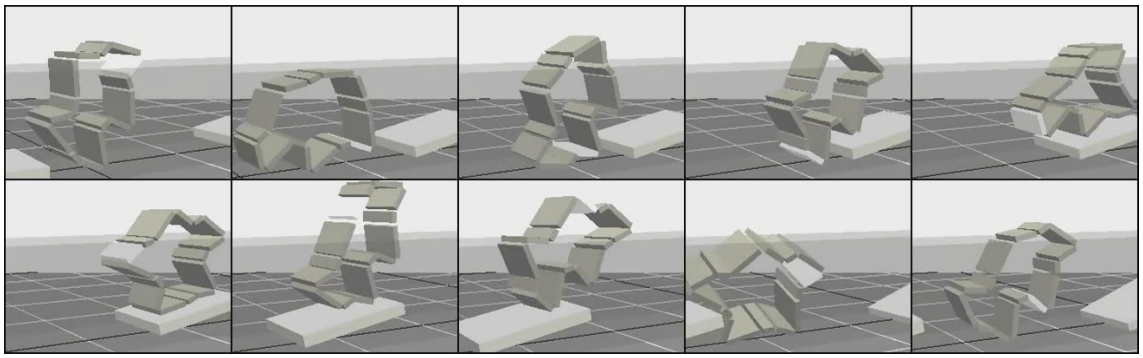


Figure 8.33: The 'horse-shaped' locomotion controller. The picture series shows every 60th simulation step.

These approaches could be called humorously the "animal parade", because their shapes and motion patterns remind of that of animals, like a horse (figure 8.33), a pinniped (figure 8.34) or a tadpole (figure 8.35).

The controller of the tadpole-shaped motion is shown in figure 8.36. This controller is interesting, because it produces a regular, repeating body motion, that reminds of an animal using its 'tail' to push itself forwards. In reality, the controller still performs a rolling motion, but the pushing of the 'tail' helps the animat to overcome obstacles. Hereby, the 'virtual' body parts, e.g. the 'head' and the 'tail', remain well defined throughout the entire motion.

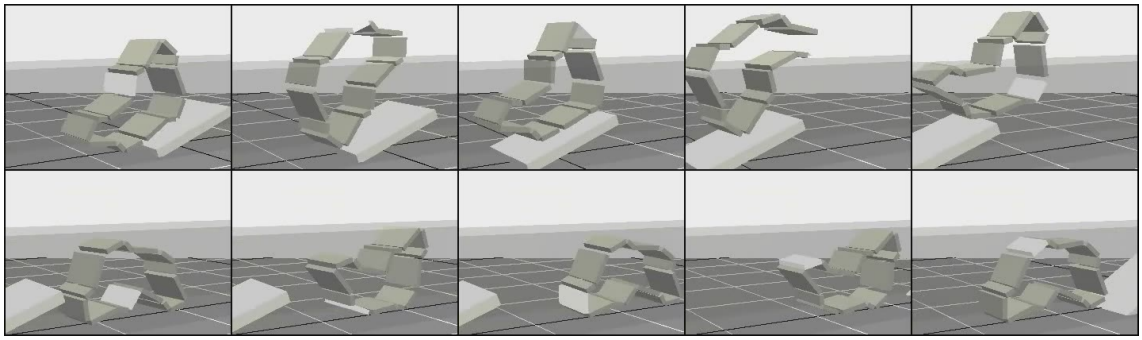


Figure 8.34: The 'pinniped-shaped' locomotion controller. The picture series shows every 50th simulation step.

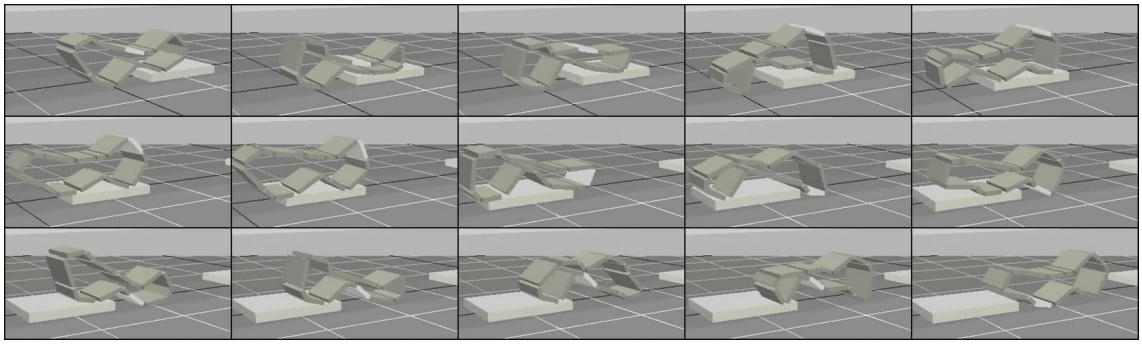
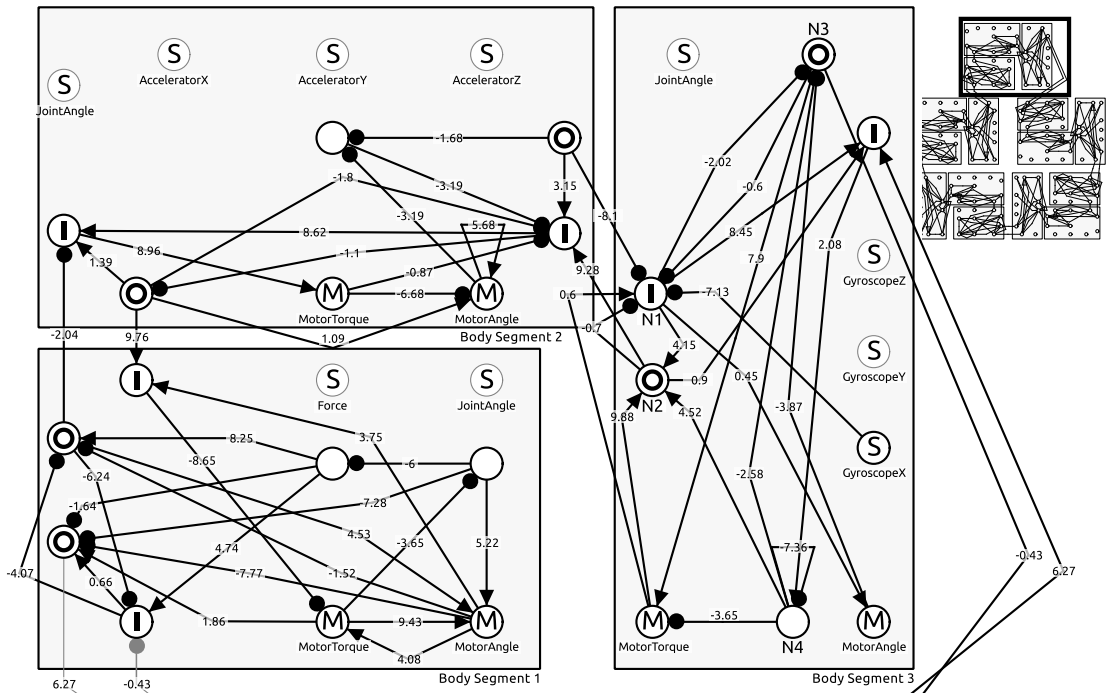


Figure 8.35: The 'tadpole-shaped' locomotion controller. The picture series shows every 30th simulation step.

Looking at the neural network (figure 8.36) it becomes clear, that – as expected – the pathways between the sensor and the motor neurons are quite long. The motor neurons of both, the preceding and the successive sensor-less modules have a minimal synaptic path length of between 5 and 7 synapses. But really astonishing is, that the gyroscope sensor only provides a single synapse, that feeds with a very high weight into an oscillatory structure (formed by the neurons `N1`, `N2`, `N3` and `N4`), from where the signals are distributed to all other parts of the network. The oscillatory behavior of this structure is shown in figure 8.37 in relation to its corresponding gyroscope sensor.

It can be seen, that the behavior is composed of six strictly consecutive phases (step 0 - 780). The transitions of these phases are controlled by the gyroscope sensor and therefore ultimately by the body. In the first phase (steps 0 - 290) the structure (neurons `N1` and `N2`) oscillates in-phase, which results in an activation pattern for the four involved body segments (the light-gray *lead* segment corresponds to the module with the gyroscope (segment 3) and is at the second position of that quadruple), such that the four controlled segments try to form an s-shape. The first joint hereby is bent inwards, the other two joints outwards. Because of the involved oscillations, the `MotorTorque` neurons are only activated every second update step, which results in an applied medium torque (see also



(a) Network Details Network: N: 150 S: 360 Parameters: B: 0 W: 72

Figure 8.36: Details of the gyroscope-based neuro-controller producing a tadpole-shaped motion.

section 8.2.2) that can easily be overruled by stronger forces coming through the segment chain.

When the lead segment reaches the front of the animat and points almost perpendicularly towards the ground, the inhibitory effect of the gyroscope sensor becomes too strong, so that the oscillation is stopped. In that phase (steps 290 - 370), all three joints are relaxed by setting their `MotorTorque` neurons to negative activations. Due to the forces in the body chain the three joints are passively bent inwards, so that the tight 'head' is formed. The oscillation remains inactive until the lead segment is oriented almost horizontally. This is due to a hysteresis effect inherent to the structure formed by the four processing neurons (`N1-N4`) of that module.

In the third phase (steps 370 - 560), the oscillation is restarted and the quadruple again approaches the s-shape. In the 'tail' of the animat, the lead segment is again tilted towards the ground as a result of to the successful realization of that s-shape. This again stops the oscillation because of a too strong inhibitory influence of the gyroscope sensor.

The fourth phase (steps 560 - 680) is similar to the second, so the segments are all relaxed and are passively bent inwards, forming the 'lifted tail' as preparation for the pushing motion. During this phase, the lead segment is getting horizontal again. This leads to a transition to the fifth phase, in which the segments again try to form the s-shape.

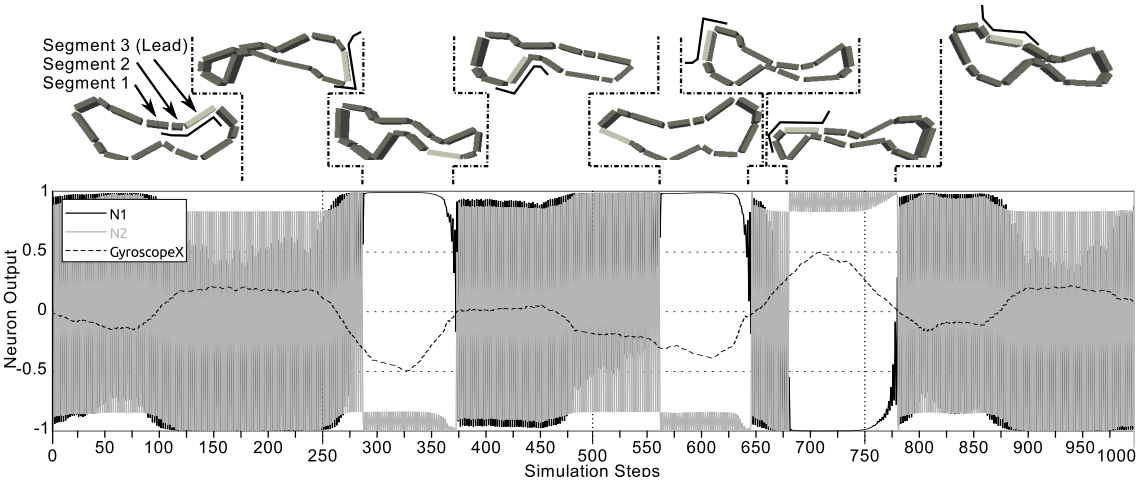


Figure 8.37: The phases of the motion controller of figure 8.36(a). The gray areas represent period-2 oscillations of neuron `N2`, which largely hides a second period-2 oscillation of the `N1` neuron (black areas). The small images of the animat on top of the figure show the motion states reached at the *end* of the corresponding phase. The actively approached s-shapes are indicated in the small animat images with black lines. Starting with step 780 the first phase is entered again. Due to interactions with the body, every animat rotation is a bit different, but the six main phases are always observable.

Due to the forces of the body, this phase of oscillation now is much shorter, because the lead segment is quickly pushed steeply upwards. The resulting high output of the gyroscope again stops the oscillation, but now with an excitatory influence. In the hereby triggered sixth phase (steps 680 - 780), the quadruple now approaches the opposite s-shape, bending the first joint outwards and the other two joints inwards. This causes the typical pushing motion of the 'tail'. During this motion, the lead segment reaches a horizontal orientation again. This triggers the first phase, in which the quadruple approaches the s-shape again.

This brief look at the network dynamics also explains, why the animat moves so compactly flattened. In the 'head' and the 'tail', the plates are almost fully bent passively, forming a narrow ring. And in the middle part, two s-shapes are pushed towards each other, i.e. towards the inner side of the body against each other. Hereby, the segments of the opposite sides often collide and shortly get stuck because of friction, which makes this animat look very organic and life-like.

Other Variants of the Experiment. As stated above, the experiment has been performed with different configurations and with different enabled sensors. To show, that the local optima of the different constraint masks significantly differ and because that behavior nicely fits into the "animal parade" analogy, figure 8.38 shows the dominant behavior of a constraint mask, where all sensors except the angular sensors have been enabled. This

motion looks like a camel or dragon, persistently keeping two humps (or spikes) on its back. Hereby, the controller effectively crosses the entire hurdle track without any backtracking.

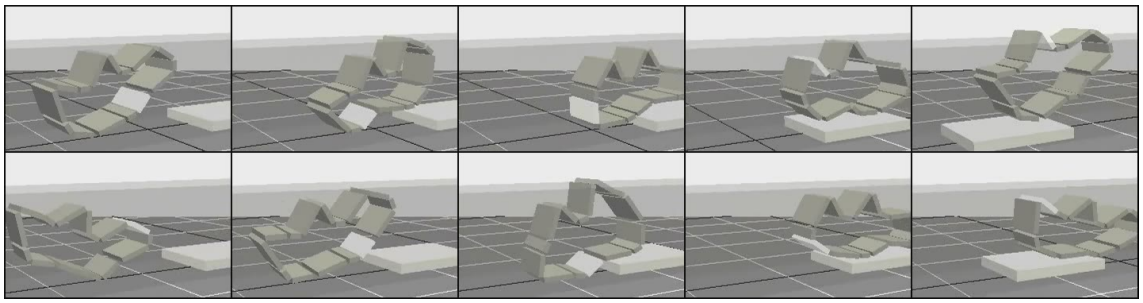


Figure 8.38: The 'camel-shaped' locomotion controller evolved with all but the angular sensors. The picture series shows every 60th simulation step.

8.4 Peripheral Structure Variations

The third series of experiments demonstrates the guidance of the evolution through the definition of peripheral structures and the use of very specific constraints. This strongly biases the search space far more than in the previous sections, because these constrained initial networks focus the search on very specific structures and network organizations. Such initial networks are useful in many contexts, for instance when a control approach should be replicated, verified, varied or simply given a try. The search with highly constrained networks usually focuses on quite complex network configurations, that are unlikely to evolve on their own and that often are so specific, that the corresponding network configurations are scattered incoherently across the unconstrained search space. An evolutionary search on such a class of networks can only be performed successfully, if the search space is reshaped in such a way, that all the previously scattered configurations are now dominating or better exclusively defining the search space. Then, evolution cannot drift away from those normally unlikely configurations, so that – if the evolution succeeds – all results are guaranteed to solve the problem in the desired way.

This section shows three examples for such highly constrained networks. The first experiment forces the evolution to integrate a previously developed oscillator from the module library as a base for the locomotion behavior. The second experiment predefines a structure with rotating activity centers, that should be used by the controller to synchronize the body segments in a suitable way. In the third experiment a very specific, quite artificial structure is examined: This structure only allows a vertical, excitatory processing of the module-internal sensors, which in turn can be modulated by inhibitory synapses horizontally coming from the neighbor modules. Independently of whether such a structure makes sense at first glance, it nicely demonstrates how specific the constraint masks with ICONE can be, and that once again, novel successful neuro-controllers could be found, even with such a 'strange' initial network.

8.4.1 Experiment 8: An Oscillator as Pattern Generator

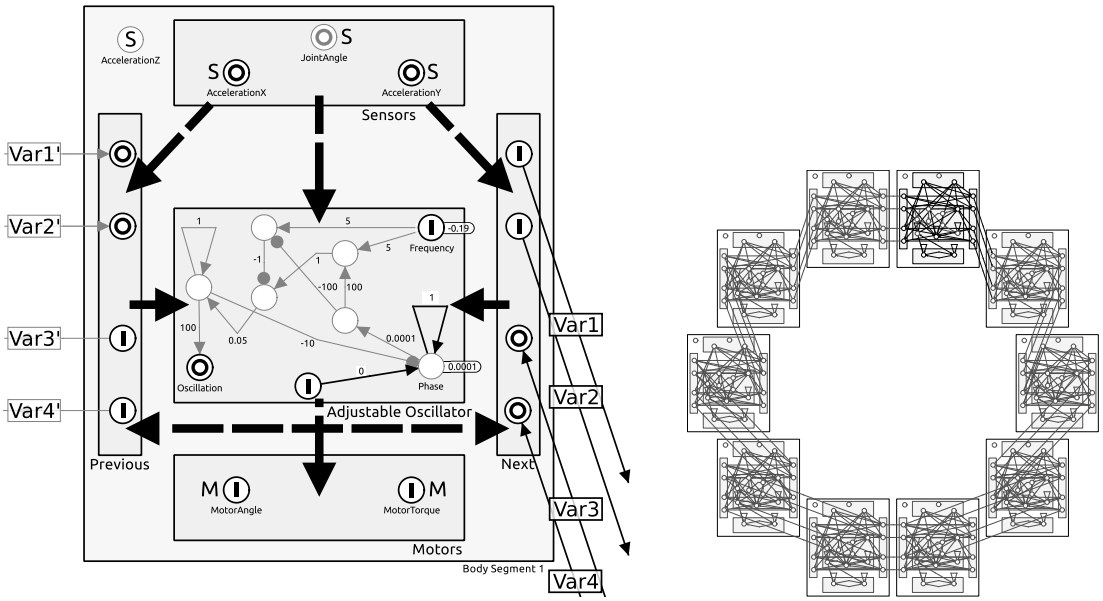
The regular, repetitive motion of some evolved neuro-controllers inspires to examine, whether a rolling locomotion behavior may also be realized as combination of modulated pattern generators, such as oscillators. For this, an oscillator structure, that has been developed in a previous experiment, has been taken from the neuro-module library (Pasemann et al. 2012). This module is shown in figure 8.39(a) in the middle (`Adjustable Oscillator`). The oscillator allows the control of its frequency during the execution of the network. The more activated the `Frequency` neuron is, the lower the frequency of the oscillation becomes. The frequency can be determined either statically by specifying a bias term on the `Frequency` neuron, or dynamically by adding synapses to that neuron. A second way to influence the oscillation is given by the second input neuron. Through this neuron, the current state of the oscillation can directly be influenced, which allows for instance to reset the oscillation.

This oscillator module is fully configured with constraints and network tags (see section 5.1.2 for more information about *module refinement*), so that only a few parameters of that subnetwork are mutable. In the figure, these mutable neurons and synapses are drawn with solid black lines. Only one bias term and two synapses can be affected by mutations. These parameters are additionally restricted in their range to reduce the probability of destructive settings. This largely protects the function of the oscillator during evolution.

In this experiment, such an oscillator module has been added to the master module of the initial network to force its usage for the behavior generation. The primary challenge of this experiment is, that due to the *clone* constraints, all oscillators are identical and will also produce an identical output, if the individual oscillators are not somehow influenced by sensor signals. For a locomotion, a phase shift between the oscillations is expected to be required, so the oscillation needs to be modulated by the sensors.

Modularization. As shown in figure 8.39(a) the oscillator module from the module library has been added to the master module. To force evolution to use the oscillator for the locomotion and hence to prevent that the motors are controlled without using the oscillator, synaptic pathways have been added. The sensor signals (`Sensors` module) may be connected to the `Previous` or the `Next` module through a fixed interface, so that these signals may influence their direct neighbors. Both, the sensor signals and the signals from the neighbor modules are allowed to connect to the `Adjustable Oscillator` module or to each other. The motors can only get input from the oscillator module. This ensures, that all signals from the sensors have to pass through the oscillator.

In all submodules except of the oscillator module, new neurons are allowed for a processing of the signals. The connections between the main modules have been realized using fixed connections and the *network equations* constraint, like in the experiments before. The self-coupling of the `Phase` neuron has been limited to the range of [0.8, 1.2] to preserve the required integrative properties of the neuron. The mutation probability of the bias term of the `Frequency` neuron, that defines the base frequency of the oscillator, has been increased to 200% and its change variance has been reduced to 25%.



(a) Modularization details of the master module of the oscillator-based controller.

(b) Search Space
Network: N: 230 S: 560
Parameters: B: 12 W: 45

Figure 8.39: The initial network for the oscillator-based controller experiment. (a) The modularization of the master module and (b) the maximal search space of the modularized network.

Results. Only 6 out of 26 experiments did not manage to find controllers with a suitable forwards motion. Two controllers have even been able to pass the entire hurdle track. The fitness development of the 10 best experiments is shown in figure 8.40.

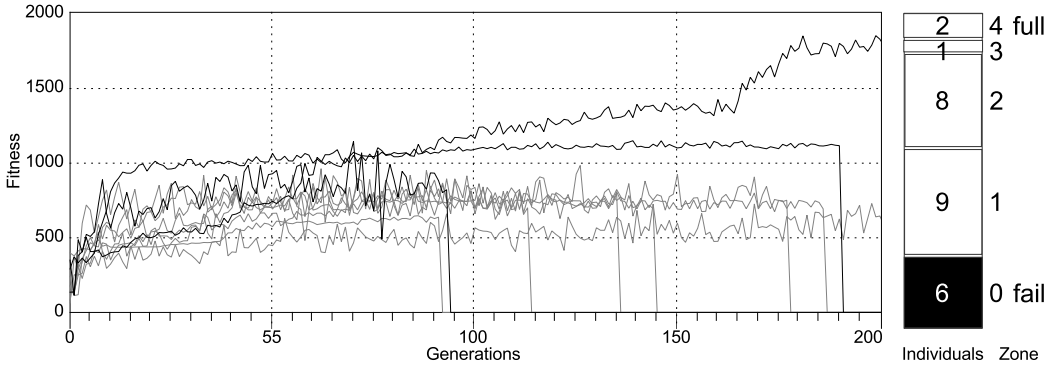


Figure 8.40: The fitness development of the 10 best evolution runs. For a better readability the best three evolution runs have been highlighted with a darker color. The right diagram shows the distribution of controllers with respect to their ability to overcome obstacles.

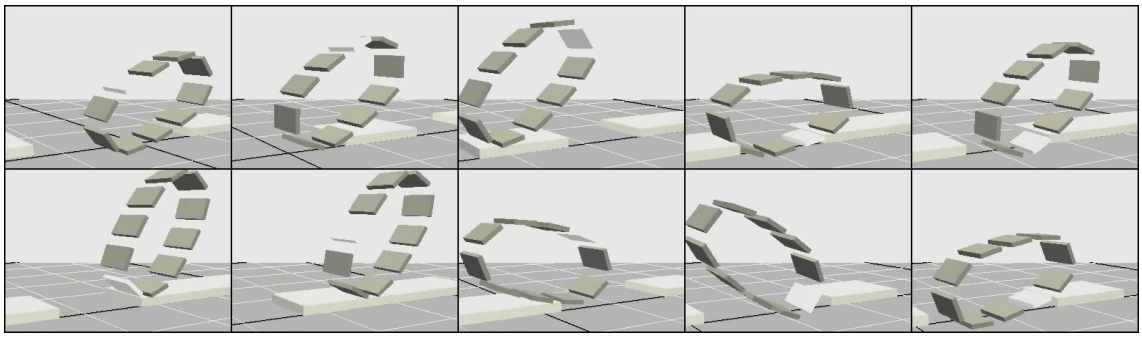


Figure 8.41: The motion sequence of the oscillator-driven locomotion controller shown in figure 8.42. The picture series shows every 40th simulation step.

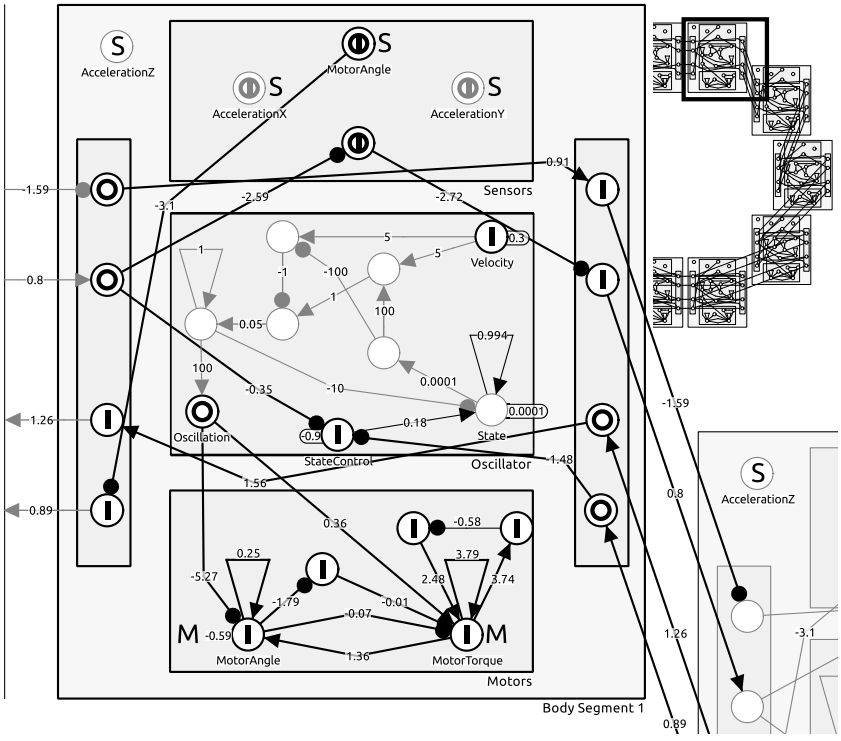
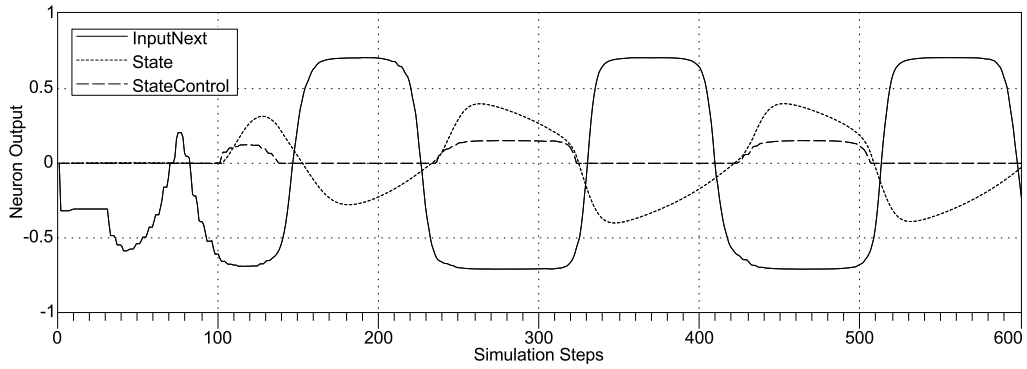


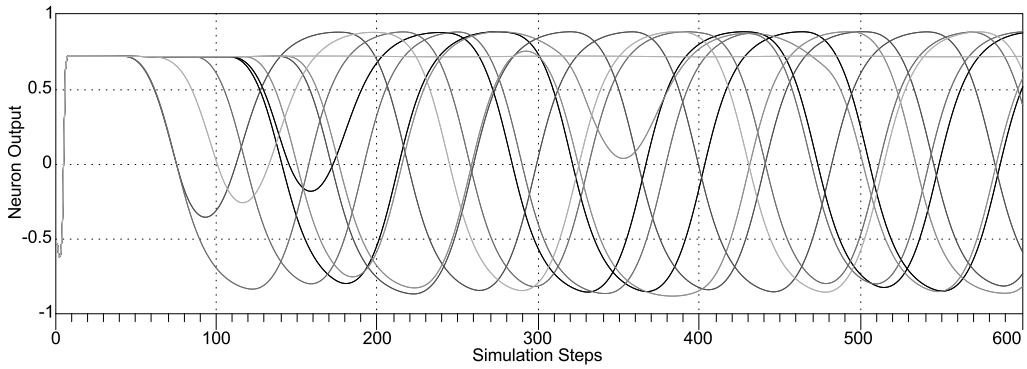
Figure 8.42: Network details of the neuro-controller using a damped, driven oscillation.
Network: N: 270 S: 350
Parameters: B: 2 W: 24

right from the beginning, using the angular sensors, the body dynamics and a surprising reconfiguration of the oscillator into a damped oscillator.

The base of this reconfiguration is the reduction of the self-coupling of the `State` neuron. With this weaker self-coupling, the oscillator is not able to oscillate with its full amplitude anymore. Instead, the initial oscillation is barely measurable and thus not enough to have an effect on the motor neurons. However, if the `StateControl` neuron is activated, then the oscillation is initiated, but decays after a short while. The same effect can be observed, when the previously activated `StateControl` neuron becomes inactive again. Then the oscillation is again initiated and quickly decays. As a result, the oscillator is a damped oscillator now, that can be innervated by alternating, strong activation changes of the `StateControl` neuron.



(a) Modulation of the oscillation output of segment 1 through its `StateControl` neuron.



(b) The activation of the `MotorAngle` neurons of all 10 segments during a rolling motion. The segments are colored with increasing lightness the higher their indices are, starting with segment 1 having a black color. After a short while, there are five pairs of almost synchronous oscillations, each belonging to two opposite joints.

Figure 8.43: Activity plots of the neuro-controller in figure 8.42.

The trick for the synchronization now is to activate and deactivate the `StateControl` neuron based on the angle sensor signal of the successive segment, and thus to keep the oscillation running by frequent, angle dependent impulses to the `State` neuron (figure 8.43(b)). So, starting again with the gravity deforming the animat in the beginning and the

hereby emerging different joint angles in the animat, the oscillators of the single segments are initiated one after the other and soon (re-)synchronize with each other, so that segments at opposite positions oscillate almost in synchrony (figure 8.43(a)).

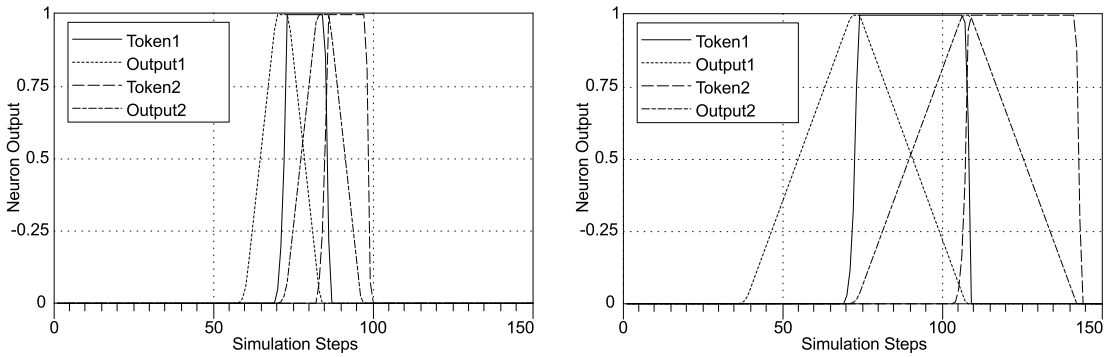
8.4.2 Experiment 9: Coordination via Token Passing

Another experiment focusing on a specific control approach examines the performance of a sensor-less joint coordination based on a token passing strategy. This should confirm, that a successful, robust locomotion behavior can also be generated entirely without any sensors. Instead, an inner rhythm is used to drive the behavior. For this, a peripheral structure was added, that spans over all main modules and is able to generate this inner rhythm (see figure 8.45(a)). The structure uses neurons with linear transfer functions that are limited to an output range of [0,1]. With their strong self-connections, the `Token` neurons of that structure remain persistently active once they get activated beyond a certain threshold and can then only be suppressed again by an active `Token` neuron of the successive body segment. Accordingly, there may be multiple `Token` neurons active in the entire network, but never simultaneously in directly neighboring segments. While active, a `Token` neuron slowly increases the activity of the `Output` neuron of its successive module and decreases the activation of its own `Output` neuron. The rate of activation change hereby is determined by the weight of the excitatory/inhibitory synapses between a `Token` neuron and the affected `Output` neurons. That weight is assumed to be similar (for inhibitory synapses with reversed sign) for all synapses of that type in the entire network, so that the generated inner rhythm is constant throughout the entire body. Once the activity of an `Output` neuron exceeds a certain threshold, it activates its corresponding `Token` neuron strong enough and that `Token` neuron gets fully active because of its self-coupling. This inhibits the preceding `Token` neuron, slowly deactivates the now active `Output` neuron and slowly activates the successive `Output` neuron.

Accordingly, the `Token` neurons are activated and deactivated in a cyclic way, that may be interpreted as passing one or more tokens (or activation centers) from one segment to the next with a steady, fixed speed. Figure 8.44 shows the activation of two successive `Token` neurons and their corresponding `Output` neurons, while an activation center passes through those two body segments.

Modularization. As in the experiments before, only one master module is evolved and the modules of the other body segments are clones of that master module. The neurons and synapses of the token passing structure have been protected to prevent its destruction during the course of the evolution. Only the weight of one synapse (marked as `Var1`) can be mutated, limited to a range of [0.005, 0.2]. This range corresponds to plausible rhythm frequencies. That weight is used as a variable with the *network equations* constraint to calculate the weights of all other synapses that control the speed of the activity forwarding. These other synapses are tagged each with a network equation, either $w = Var1$ or $w = -1 * Var1$ to calculate their weights (see figure 8.45(a)).

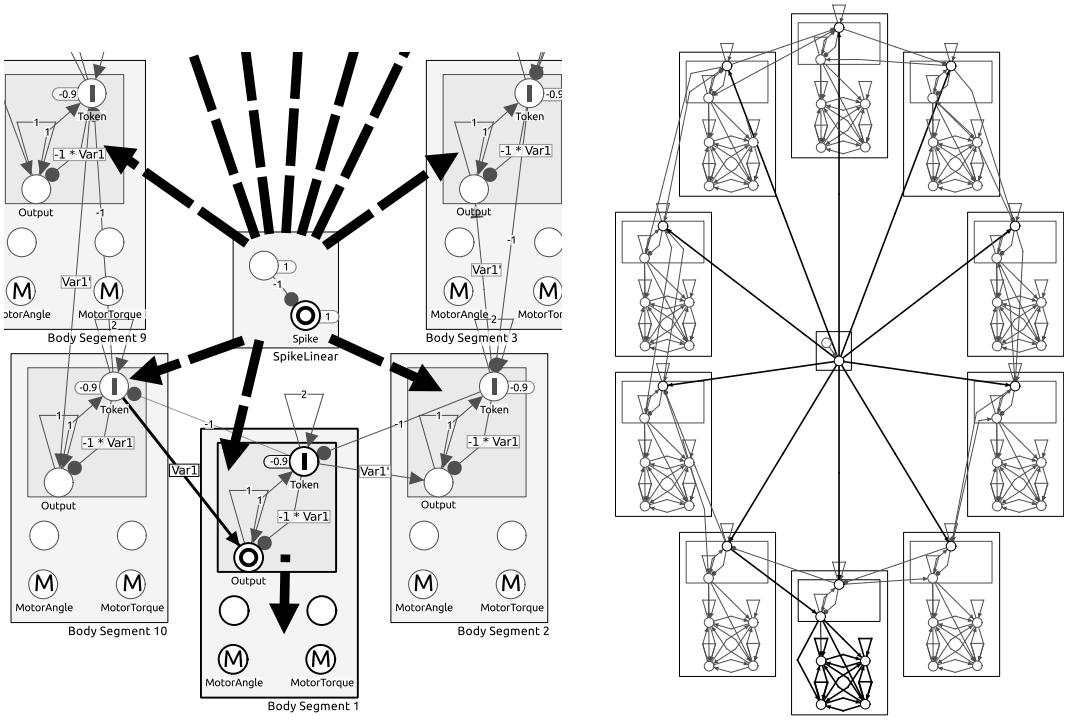
The valid synaptic connections are restricted with *synaptic pathways*. Only synapses going from the output of the token forwarding structure of the master module towards its



(a) Weight $Var1 = 0.1$.

(b) Weight $Var1 = 0.03$.

Figure 8.44: Activation of the `Output` and `Token` neurons of two successive modules for two different choices of `Var1`.



(a) Modularization details of the master module of the token passing network.

(b) Search Space (+2 Neurons)
Network: N: 62 S: 271
Parameters: B: 7 W: 31

Figure 8.45: Details of the modularization for the token-passing experiment. (a) The modularization of the master module and (b) the maximal search space of the modularized network.

inner neurons are allowed. In addition, connections from the middle module (`Spike`) to each of the input neurons of the token forwarding structures (`Token` neurons) in all main modules can be added during evolution. These connections are used to activate one or more `Token` neurons when the network starts. The middle module creates a single activation peak during the first simulation steps, which is sufficient to activate a connected `Token` neuron. This initial activation is necessary. Otherwise the entire network remains inactive. Instead of giving a single activated `Token` neuron in advance, the number and distribution of such activations are left open for evolution to get more varieties of the locomotion coordination. The synapses towards these `Token` neurons are limited to a positive range of [0, 10]. The visibility depth of the `Token` neurons as module input neurons have been increased to 2, so that the neurons are visible for the output neuron of the `Spike` module.

The maximal search space with two additional processing neurons in the main module is shown in figure 8.45(b).

Results. Starting with this constraint mask, only a single experiment failed to develop a forwards locomotion. Most evolved controllers (25 of 34 experiments) could fully master the hurdle track. This indicates, that with the given peripheral structures, successful locomotion behaviors are quite easy to find. In fact, all that is necessary is the selection of a suitable number of active tokens, a suitable rotation speed for the tokens and an appropriate motor response to an active token in each segment. Actually, among the evolved neuro-controllers of this simple, sensor-less behavior was the fastest solution of all experiments. The fitness progress of the 10 best evolution experiments is shown in figure 8.46.

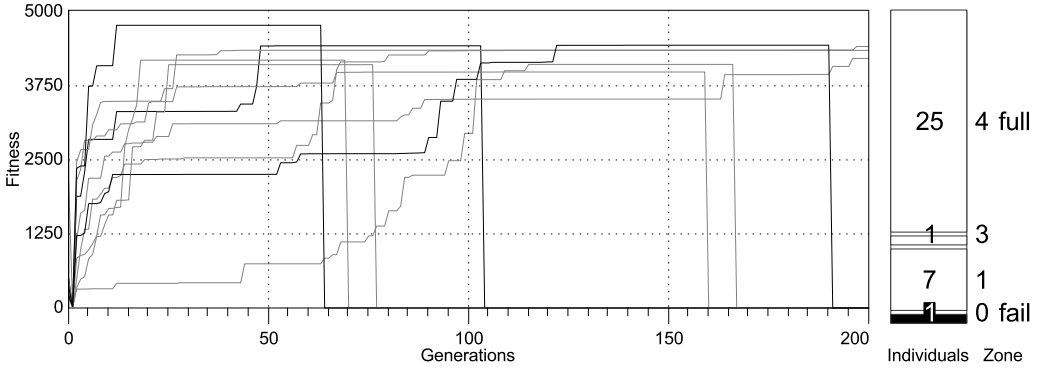


Figure 8.46: The fitness development of the 10 best evolution runs. For a better readability the best three evolution runs have been highlighted with a darker color. The right diagram shows the distribution of controllers with respect to their ability to overcome obstacles.

Interestingly, although any combination of tokens has been possible, almost all controllers used either a single token or two tokens in opposite segments. This makes sense, because as a reaction to an active token, a segment can only either stretch, contract or relax. And the chosen strategies cannot be mixed due to the clone constraints, therefore the animat has either only contraction zones, stretching zone or relaxation zones. A single

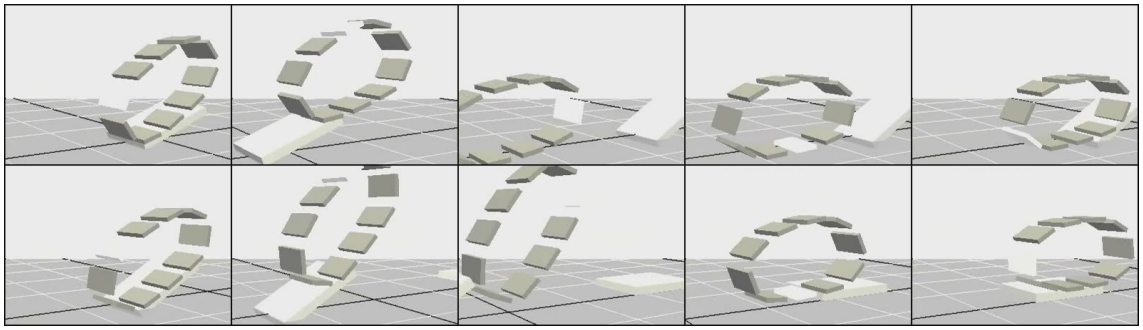


Figure 8.47: The motion sequence of the token-passing neuro-controller shown in figure 8.48. The picture series shows every 60th simulation step.

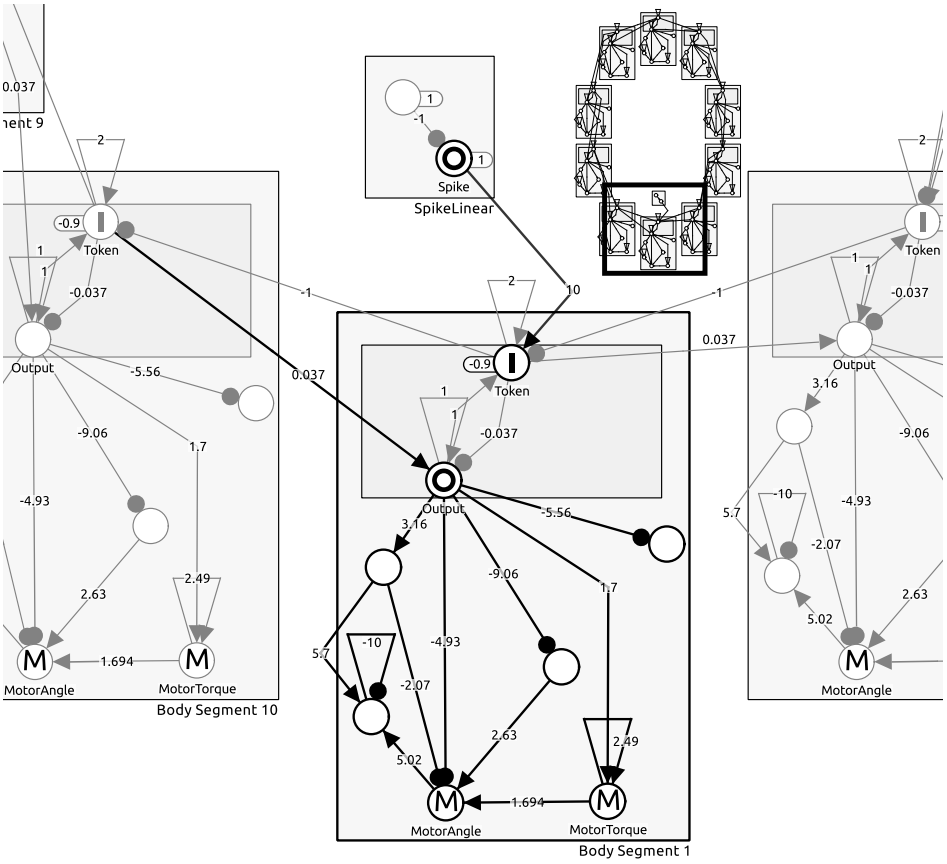


Figure 8.48: Details of an evolved token-passing controller. The `Spike` neuron is here connected to the master module for a better comprehension, although the evolved controller had a connection to one of the other main modules. Functionally, there is no difference.

Network: N: 82 S: 182

Parameters: B: 0 W: 13

zone of any kind is already sufficient to generate a fast forwards motion (figure 8.47 shows the behavior of such a controller). Two zones are also suitable, as long as they are approximately opposite of each other, but any other combination disturbs the motion pattern or deforms the animat so that the hurdles become difficult to pass. The only combination of more zones would make sense for relaxation zones, in particular, if all segments except for one or two opposite segments are in a relaxed state. But because tokens cannot exist in directly neighboring segments, this scenario cannot evolve.

Figure 8.48 shows one example of the evolved controllers. This controller uses a single rotating activation center and contracts the body segment when the `Output` neuron becomes active. When the `Output` neuron is inactive, then the segment is stretched. This already is sufficient – in combination with the closed chain of the body – to move the animat forwards (see figure 8.47).

8.4.3 Experiment 10: A Feed-forward Processing Column

The next experiment shows another example of how ICONE constraints can be used to restrict the organization of a network to fit very specific requirements. Signals are here assumed to be forwarded excitatory from the local sensors to the corresponding local motors. The signals are hereby modulated by inhibitory signals that come from the further away sensors of the neighbor modules. This model is arbitrarily chosen and does not try to prove a specific hypothesis. However, such a model nicely demonstrates the use of the ICONE constraints and shows that even in such an artificial structure, interesting neuro-controllers can be found.

Modularization. For the modularization of this network, a number of additional submodules have been added to allow a fine control of the synaptic pathways. Figure 8.49 shows the modularized organization of the network.

The desired network structure should consist of a pure feed-forward structure with only excitatory synapses between the sensors and the motors of each main module. For this, three modules have been added. The one named `Input` (I) represents the beginning, the `Output` (O) module the end of the processing column. The `Controller` (C) module between the two is the subnetwork meant for the actual processing. The other modules at the right and the left (P_I, P_O, N_O, N_I) are the interface modules for the communication with the neighboring modules (previous and next).

The transfer function of all neurons are changed to prevent their activity from becoming negative. This would undo all efforts of forcing excitatory and inhibitory synapses, because depending on the neuron activity, the influence of a synapse can be of both, inhibitory or excitatory, when using the hyperbolic tangent. The transfer function here is defined by

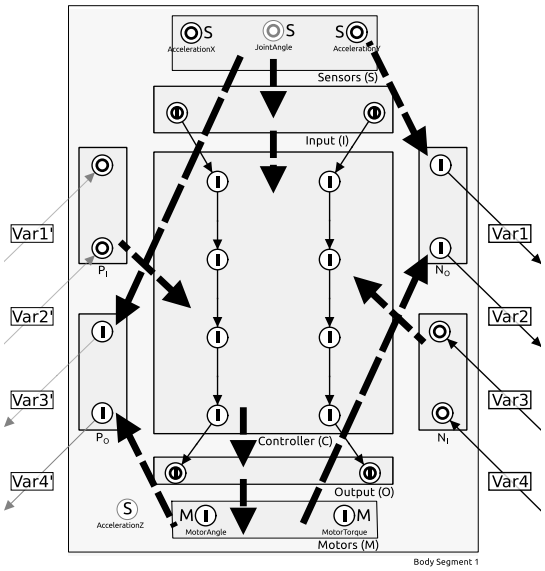
$$o_i(t) = \frac{1}{1 - e^{-10a_j+5}}; \tag{8.3}$$

with a_j being the activation of the neuron (compare section 2.1). This transfer function (*shifted sigmoid*) is shown in figure 2.1(b) on page 11. The function is nonlinear and limited to a range of [0, 1]. Thus, the output of all neurons is always positive.

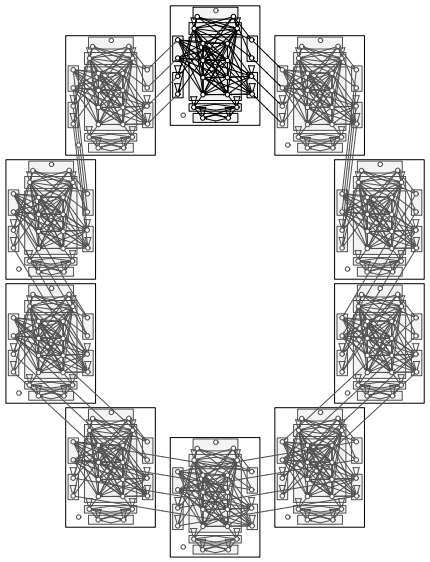
The signal flow of the network is defined using the *synaptic pathways* extension. The valid synaptic connections for this network are depicted in figure 8.49 by the thick black arrows. The sensor signals have to flow vertically from the `Sensors` module (S) over I, C and O to the motor neurons in `Motors` module (M). Horizontally, the sensor signals can be connected through the interface modules to the column modules C of the previous and the next neighbor.

The synapses between the main modules are, as before, predefined and their weights are controlled with a *network equations* constraint, so that only one set of weights has to be evolved to define the connective synapses between all main modules.

The submodules of the master module are strongly constrained. In all modules except of the C module, new neurons are prevented. Using a *restrict-weight-and-bias-range* constraint, all synapses between I, C and O are limited to a range of [0, 10] (excitatory) and all synapses between P_I, C and N_I to a range of [-10, 0] (inhibitory). All synapses within modules C, P_I, P_O, N_I and N_O also are restricted to be excitatory in a range of [0, 10]. Furthermore, module C is constrained by an *enforce connectivity pattern* constraint to guarantee a feed-forward structure. So, all new synapses added to that module will together form a valid feed-forward network without recurrences. Because new neurons are



(a) Modularization details of the master module.



(b) Search Space
Network: N: 260 S: 1140
Parameters: B: 22 W: 114

Figure 8.49: Details of the initial network for the feed-forward processing column experiment. (a) The modularization of the master module and (b) the maximal search space of the modularized network.

allowed to be added to the middle module, a *synchronize tag* constraint was added to automatically tag any new neuron as module input, so that it instantly becomes part of the module interface and thus can be a target of synapses coming from the interface modules at the sides. The resulting search space is shown in figure 8.49(b).

Results. With this constraint mask, the development of controllers was quite successful. 8 of 65 controllers could master the entire hurdle track, only 9 experiments failed completely. The majority of controllers (35) could at least pass the easy obstacles in zone 2 and partially some of the ramps. The fitness progress of the 10 best evolutions is shown in figure 8.50.

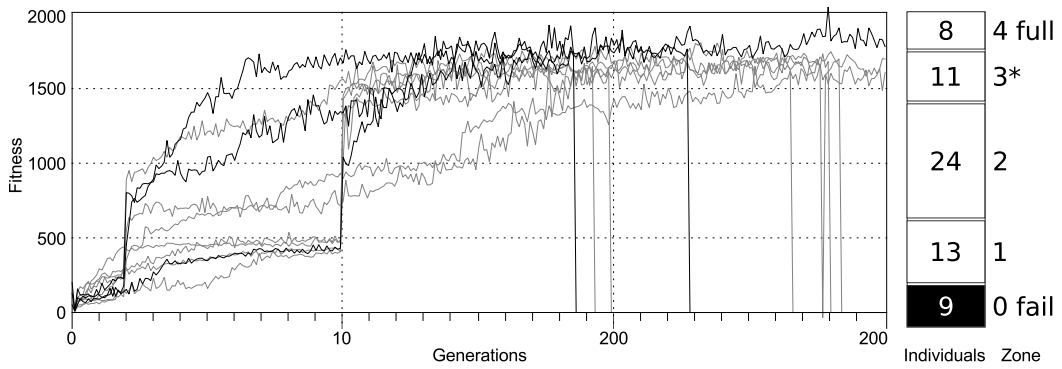


Figure 8.50: The fitness development of the 10 best evolution runs. For a better readability the best three evolution runs have been highlighted with a darker color. The right diagram shows the distribution of controllers with respect to their ability to overcome obstacles (3* stands for the ramp zone *excluding* the steepest ramp).

Most of the evolved controllers use the inhibitory influence between the neuro-modules to coordinate the locomotion behavior. Surprisingly, some controllers, like the one shown in figure 8.52, only use the inhibitory synapses for a fine-tuning, whereas the main behavior is generated by a solely local processing. This novel solution, however, has some properties not observed at other evolved controllers so far. First of all, the main 'shape' of the locomotion behavior is dominated by a circle, that from time to time elongates a bit. In previous experiments, such circle shaped behaviors have shown a limited performance on the ramps, because the round shape does not allow a sufficient forwards leaning to overcome the obstacles. This particular controller, in difference, masters the entire hurdle track with an above-average speed. One reason is, that the animat rolls quite fast on plain ground by keeping a circular shape and by using only brief elongation impulses to accelerate the rolling. With this approach, the animat can exploit the passive rolling capabilities of a wheel. Hindered by an obstacle, however, the animat can elongate explicitly to shift its center of gravity forwards.

At obstacles the behavior is quite rich in variation: The animat sometimes tries to reach out more and more forwards to overcome an obstacle, but it may also track back a bit and then lean forwards again with momentum. Sometimes, the animat also just moves

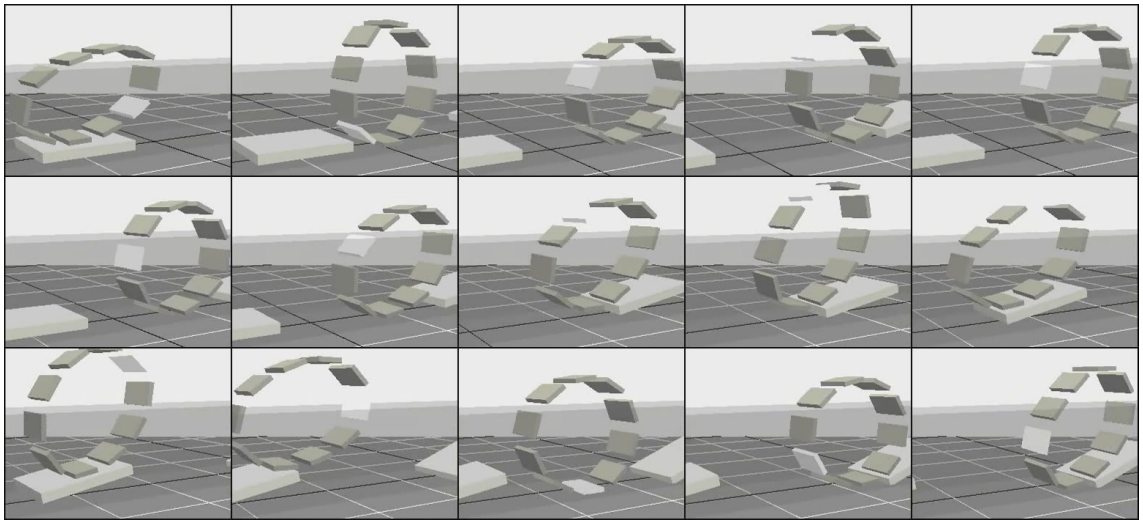


Figure 8.51: Wheel-shaped locomotion with elongation at hurdles and backtracking capabilities (pictures 4-7). The picture series shows every 60th simulation step.

backwards a short distance and then approaches the obstacle again with a higher velocity and thus with an additional run-up, that often is enough to propel the animat over the obstacle.

The controller (figure 8.52) and the output of the activation of the motors and sensors (figure 8.53) hereby are interesting, although at a rough analysis, the neuro-dynamical cause of the rich backtracking behavior could not be absolutely identified yet. As indicated before, the inhibitory synapses from the neighbor modules are optional and do not contribute essentially to the overall behavior. Furthermore, most of the feed-forward synapses in the middle module can also be removed, which only slightly reduces the performance of the behavior. So, the main behavior is not generated – as was expected – in the middle module, but instead in the `Input` and the `Output` modules of that processing chain (modules I and O in figure 8.49(a)) and in the motor layer (module M). The input and output modules preprocess the noisy acceleration sensors and integrate their noisy sensor peaks to more coherent, better defined patterns. In the motor layer, the integrative self-coupling of the `MotorTorque` neuron seems to be especially important, because it causes the characteristically slowly decaying torque patterns shown in the activation plot of figure 8.53(c). This slowed decay of the torque activation sets the joints of the animat permanently (with varying intensities) under a constant tension, which results in the approximately round shape. However, the acceleration driven peaks shift the tension always a bit between the segments, so that the short elongation impulses form. At obstacles, the rolling of the animat is stopped and thus, due to the now less varying acceleration signals, some segments remain in the state with the torque and angle peak, while the `MotorTorque` of the other segments without new torque impulses decays to zero. During a normal circle-shaped motion, the torque outputs of the segments never become fully inactive and most often, the majority of the joints are active at a medium strength, which leads to the rounder

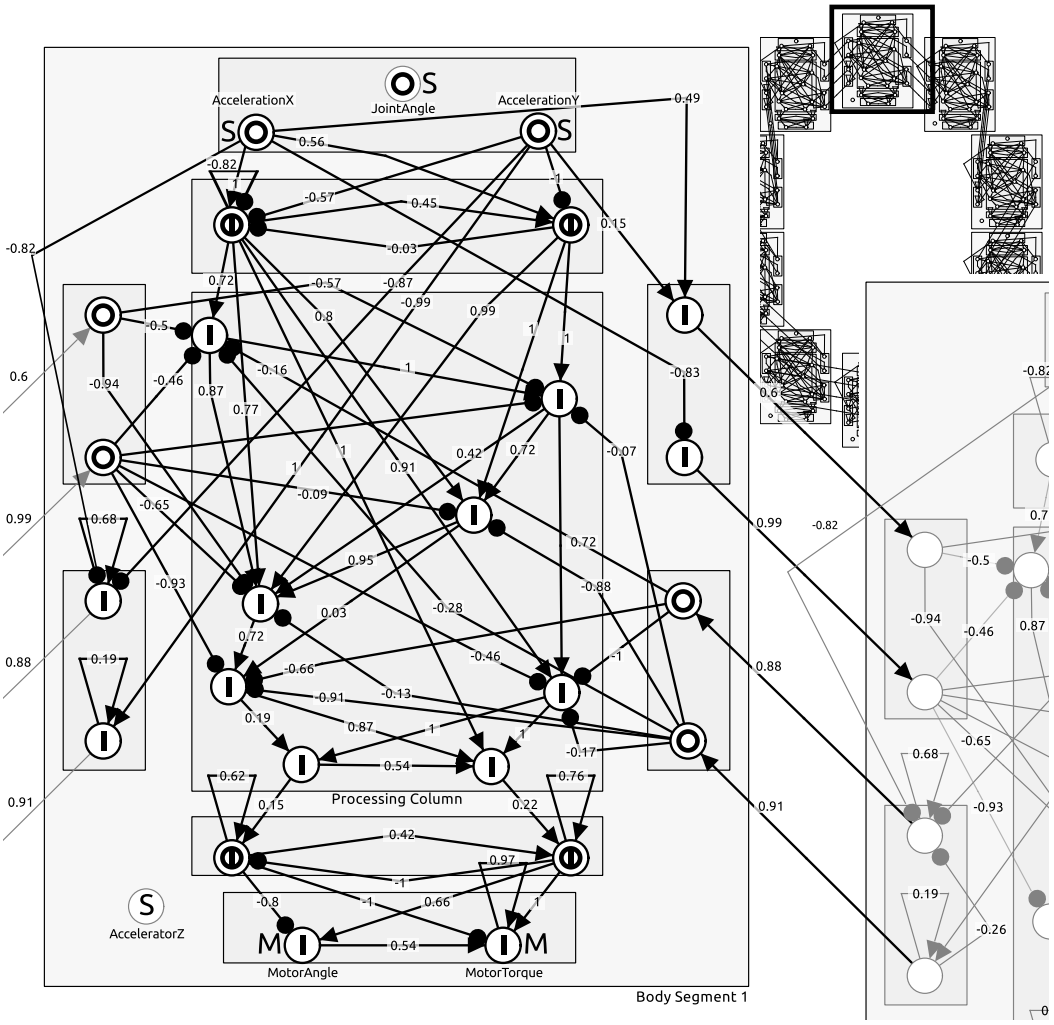
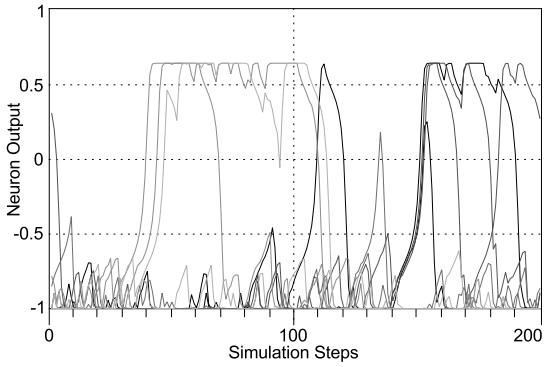
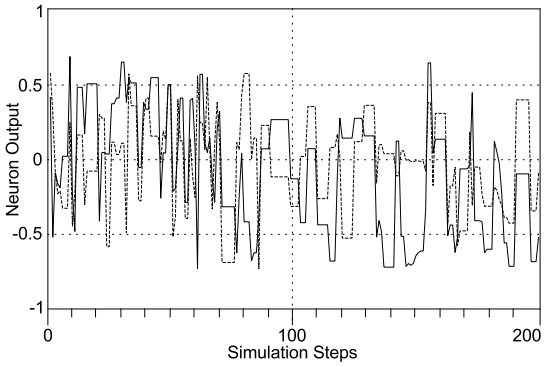


Figure 8.52: Details of the wheel-shaped feed-forward controller with backtracking capabilities.
Network: N: 192 S: 336
Parameters: B: 0 W: 28

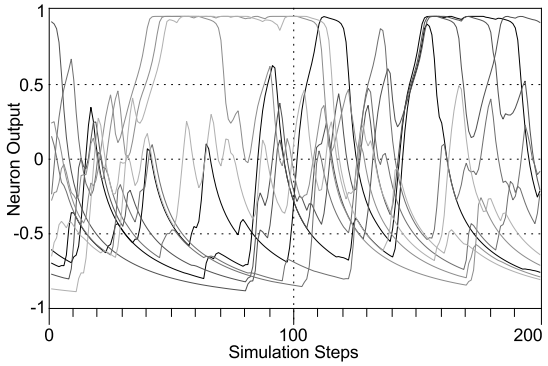
shape (compare figures 8.53(c) and 8.53(d)). Now, when some joints become fully passive while others are strongly activated, a pronounced elongation appears, that often helps to overcome the obstacle. During this elongation, the acceleration sensors with their quite noisy output (figure 8.53(b)) disturb the elongation in unpredictable ways, which leads to the rich behavioral repertory at the obstacles. In this context, the backtracking and the preparation of a run-up might also be just effects of a disturbed elongation process, at which the then wheel-shaped animat simply gets a backwards impulse, whereupon it rolls backwards for a short distance.



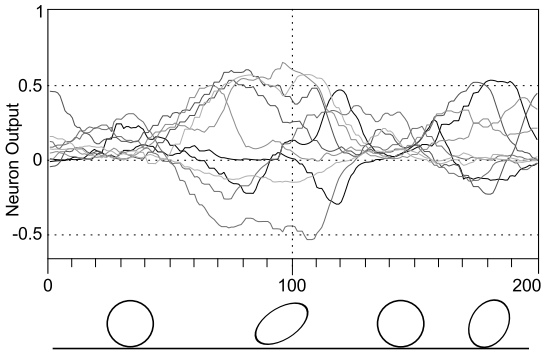
(a) The activation of the `MotorAngle` neurons of all 10 segments



(b) The `AccelerationX` and `AccelerationY` sensors of segment 1



(c) The activation of the `MotorTorque` neurons of all 10 segments



(d) The activation of the `JointAngle` sensor neurons of all 10 segments

Figure 8.53: Activity plots of the neuro-controller shown in figure 8.52 while the animat is moving on plain ground. The segments are colored with increasing lightness the higher their index are, starting with segment 1 having a black color.

Conclusions. This experiment illustrates again, that even with very specific constraint masks, the outcome can be surprising and may exploit a different approach than the expected one. However, the probability of finding such a particular controller differs between different constraint masks, and therefore, without that specific constraint mask, its discovery would have been unlikely. So it seems, that the definition of constraint masks does not necessarily have to be guided by an elaborated plan, strategy or heuristics, but that even arbitrary, sufficiently restricting constraint masks may do the trick to find novel controllers. This suggests, that having a constraint mask at all is more important than providing a particularly 'clever' one.

Chapter 9

Discussion

This chapter discusses the ICONE method as a search tool for the domain of mid-scale neuro-controllers with non-trivial animats. The discussion covers issues with respect to the method's features, its applicability, the evolution performance and the quality of the evolved neuro-controllers. The chapter is then concluded with a brief outline of the ICONE related future work.

9.1 ICONE Features

This section discusses the effects of selected features of the ICONE method. This addresses especially the mechanisms for search space restriction, the guidance of evolution experiments and the influencing of the evolution process.

9.1.1 Search Space Restriction

The high dimensionality of the search space has been identified as the major reason for the scaling problem in neuro-evolution. However, referring to the sheer number of parameters only reflects the problem at the surface. In detail, successful variants of larger, non-trivial neuro-controllers are seldom spaciously distributed over the search space, but instead tend to be largely clustered in a comparably small set of subspaces of the global search space. Thus, with increasing size of the neural networks, it becomes also increasingly more difficult to find such promising clusters at all, especially when these clusters only account for a very small percentage of the search space. This clustering is often due to inherent organizational features of mid-scale networks, for instance regularities, symmetries, a separation of function into specialized network areas, a hierarchical organization (of behaviors or signal processing), structure reuse, the tendency to prefer local over global processing (small world properties, compare Bassett and Bullmore 2006; Newman 2000; Sporns and Honey 2006), just to name a few of the features. In biological nervous systems, the existence of such organizational features is common knowledge (Amaral 2000). I argue, that

these features become more and more important and severe the larger the networks become and the more (distinct) functional and behavioral capabilities are combined in the same network.

This certainly means that the assumed existence of such organizational features in mid- and large-scale networks can be used as heuristics to focus the search on promising subspaces of the parameter space instead of performing a global search. This does, in fact, exclude some valid network configurations from being found, but the overall search space often can be significantly reduced, so that it becomes more likely to actually find a successful controller. As a reminder, the focus here is primarily on finding successful neuro-controllers at all, which already is a difficult task in that domain. About this fact and about the necessity of a search space reduction, there is not much of a debate. The debate starts when the *'correct'* heuristic, measure or technique is discussed. I argue, that up to now, no general heuristic method has been proposed, that fits all experimental scenarios and that can be used solely to reduce the search space. Instead, all proposed measures show strengths in some experiments and severe weaknesses in others. This is the reason why ICONE was designed to be as open as possible for different heuristics regarding the search space restriction, so that the best heuristics for each experiment can be *combined* to benefit from their strengths and to avoid their weaknesses.

With its many search space reduction features – such as modules, functional constraints, synaptic pathways, network tags, interactive and iterative evolution – ICONE does not provide a single search space reduction heuristics, but instead a very flexible, fine-adjustable and controllable way to realize various restriction heuristics specifically defined for each single experiment. These heuristics are also not necessarily static over the course of the experiments, but can be adapted flexibly, leading to a context sensitive search space reduction. Because of its specificity with respect to the experiments, heuristics in ICONE can combine very complex and experiment specific features that would not be possible with any general search space reduction heuristic.

9.1.2 The Power of Peripheral Structures

A common and very efficient search space reduction measure in ICONE is the distinction between *peripheral* and *focus* structures in the evolving neuro-controllers. The predefinition of peripheral structures condenses the evolutionary search problem to those *unknown* network parameters that are the focus of the experiment. This avoids the costly search for network structures that are considered necessary, but that are – with respect to the experiment – trivial accessory structures. Starting evolution with an initial network already providing such known structures as (optionally mutable) peripheral structures results in a much smaller search space, a simpler fitness landscape and therefore a higher chance of finding a solution.

Also obviously, larger networks evolved using peripheral structures are often neurodynamically richer than smaller networks without peripheral structures evolved in search spaces with comparable dimensionality. This is because peripheral structures contribute own, often well developed and elaborated dynamical properties to the networks, resulting in more complex global dynamics of the evolved networks. So, although both types of net-

works are evolved within similarly large search spaces, networks with peripheral structures have a greater potential to show interesting neuro-dynamical properties.

As a side effect of using peripheral structures, the search is biased towards specific solution approaches. At first glance, this may seem as a drawback, especially from the perspective of scientists that think of neuro-evolution as a universal, 'miraculous' problem solver that produces surprising and ingenious solutions without any guidance. In practice, however, without some guidance many valid questions cannot be approached with neuro-evolution in an acceptable way. In that context, peripheral structures can be used explicitly and on purpose to induce domain knowledge and to influence the direction in which controllers are to be searched. Many systematic explorations of specific control approaches are in the first place only possible using peripheral structures. Therefore, the power of guidance inherent to peripheral structures should not be underestimated.

9.1.3 Structure Evolution vs. Fixed Topology Evolution

NE algorithms with the capability of evolving both the topology and synaptic weights of the network are considered much more powerful, especially in finding interesting controllers, than those only evolving the synaptic weights on a fixed-topology network. This is especially common in the context of ER and neuro-cybernetics, because many interesting features of the network dynamics and network organization are considered results of a suitable network topology. This view, however, is only valid when the involved fixed-topology networks are classical standard networks, such as fully connected or layered networks. If applied to fixed, but complex networks with already interesting topologies, a sole weight optimization without structural modifications can still lead to interesting scientific results. This is especially true for experiments, in which a certain control approach is merely to be confirmed, than to be developed. For such experiments, the entire structure of the network can be given in advance (e.g. based on reasoning), whereas the definition of proper weights for the structure often is too difficult to be done analytically. The evolutionary algorithm here can search the typically comparably small search space to find a solution that in turn proves the validity of the approach. ICONE supports this practice natively with the definition of constrained initial networks.

9.1.4 Local Coexistence and Mixing of NE Approaches

With functional constraints, many valuable features of published NE approaches, related extensions and network models can be used with the ICONE method. Examples are learning rules (Urzelai and Floreano 2001; Zahedi and Pasemann 2007), the use of delta encoding (Whitley et al. 1991), topology regularities (D'Ambrosio and Stanley 2007), specialized neuron models, fixed connectivity patterns (Wilimzig and Schöner 2004) and topology construction heuristics.

A major difference compared to using the corresponding NE approaches or NE extensions directly is, that with ICONE all the approaches can be combined and mixed within the same evolving network using constraints. Moreover, the effects of each feature can be limited to arbitrary subsets of network elements via neuron-groups and neuro-modules.

Therefore, each feature is only applied to those network areas where its effects are expected to be beneficial, avoiding potentially harmful effects on other network areas.

A neat feature when using NE extensions via constraints is the ability to attach these extensions as functional constraints directly to a neuro-module. As a result, each neuro-module may provide its own set of extensions optimized for its use in the evolving network. If collected in a module library, such optimized neuro-modules can be reused including the associated extensions, usually without side effects to other parts of the network.

9.1.5 Control of Evolution Parameters

A major challenge in the field of evolutionary algorithms is still the identification of proper settings for the evolution operators. Clearly, these parameters have a strong influence on the evolution performance. Finding the correct settings is difficult because the 'optimal' settings strongly depend on the evolution experiment and often on the currently involved search space. Large mutation probabilities, for instance, are needed in the beginning of an evolution to start with a large variety of different initial networks and therefore a broad sampling of the search space. Lower mutation rates are required for fine optimizations when the search is close to an optimum. On the other hand, if such an optimum is an undesired local optimum, then larger mutation rates are mandatory to escape from such an optimum. Consequently, the main difficulty is to choose settings that are applicable for the *entire* evolution experiment, because these settings have to be a compromise between these scenarios.

In detail, it is even more difficult to find the proper settings. An example is the stepwise complexification of networks (Pasemann et al. 2001; Stanley and Miikkulainen 2002a), i.e. the stepwise extension of minimal networks by insertions of new neurons and synapses. If the probabilities for such insertions are set low, then the evolution requires many generations only to collect enough network elements to solve a task. If higher insertion probabilities are used, then the necessary network complexity is reached earlier, but before these network structures can be adapted to solve the task, the networks already comprise too many neurons and synapses, that hinder the further optimization.

A different, but also severe problem is that in larger networks, different network areas may be disparately sensitive to changes and thus would need different mutation settings to optimally evolve. So again, the global operator settings have to be a compromise between such highly sensitive and insensitive network areas, which can prevent both from evolving acceptably.

To avoid the selection of a single parameter set, different strategies have been proposed to adapt the settings over the course of the evolution experiment. Such adaptation is considered to significantly enhance the quality of evolved solutions (Back et al. 1991; Hinterding et al. 1997).

ICONE supports three measures to influence the settings of the evolution operators. The first is interactive evolution (section 5.2). Hereby, the user can adjust the evolution parameters interactively during the evolution. This measure is the most time-consuming one, but the results can be very good due to the involved domain knowledge and the ability to react directly on the observed evolution progress.

The second way to influence the evolution settings – at least in the reference implementation of the ICONE method – are automatic scripts, that adapt the evolution settings according to predefined rules (section 6.5). Certainly, this strategy is compatible with most NE methods, but with the optional interactivity of ICONE evolutions, it becomes easier to define such experiment-specific rules: The experiences gained from interactive evolutions for the same (class of) experiment(s) can be condensed to such sets of rules to enable a rough adaption also for unsupervised evolutions, e.g. when variants of an already successfully evolved controller are further explored with unsupervised experiments. These rules can still be specific for a certain experiment, which can be more effective than the use of standard rules.

The third measure, that is unique for the ICONE method, is to locally overwrite the evolution settings with network tags (section 3.2.4). This addresses the need to allow different evolution settings throughout the evolving networks. Overwriting evolution settings implies domain knowledge to decide, which network areas require a special treatment during mutations. This is especially given when using refined neuro-modules, whose function and desired mutation behavior is usually well understood.

Together, these measures allow a flexible adaptation of the mutation settings to reduce the problem of finding a single 'optimal' set of settings suitable for the entire networks and all phases of the evolution.

9.2 Evolvability and Performance

This section discusses the performance of the method regarding the successful evolution of neuro-controllers in the domain of mid-scale networks.

9.2.1 Increased Success Rate

The experiments performed with the ICONE method show that evolution often succeeds finding suitable neuro-controllers even in the domain of mid-scale networks. The success rate is, for this domain, comparably high, as long as the networks are sufficiently constrained. This is not surprising, because – presuming a suitable experiment with a proper fitness function – the success rate depends, to a large extent, on the dimensionality of the search space. The constraining of a mid-scale network does actually reduce that search space to a complexity commonly involved with the evolution of smaller networks and thus both kinds of evolution should roughly have a comparably high success rate. In the case of a constrained network, however, a new factor strongly influences the success rate. This factor is the *constraint mask* itself. A poorly designed constraint mask may not only lead to an unmanageable search space, but may also lead to search spaces that do not contain a suitable solution at all. This can happen inadvertently, e.g. when a constraint mask is faulty, but also when striving for a control approach that is not feasible in the anticipated way. This is why – when systematically exploring different control approaches – such bad configurations are almost inevitably encountered, demanding for a high frustration tolerance. On the other hand, a very restrictive constraint mask can already almost contain a solution, so that a success is virtually inevitable. In short, the success rate is highly depen-

dent on the experience of the experimenter and (as always with evolutionary experiments) on the way the experiment is conducted.

Therefore, it is very difficult to compare the success rate and performance of ICONE to other evolution methods. Comparing the algorithm on the basis of known benchmark problems has several problems. First, such problems do not cover the actual target domain of ICONE (evolution of mid-scale neuro-controllers), so either the required networks are too small or the task is too abstract (e.g. pattern recognition, classification). A second problem would be the choice of the constraint mask. Not using a constraint mask at all would not evaluate the full ICONE method. But since a constraint mask can focus the search by so many different degrees – also directly on an already known solution – all constraint masks may be vulnerable to criticism and in some cases considered 'cheating'. And a third problem is the (optional) interactivity of the method, which is also strongly dependent on the experience of the experimenter and therefore is not comparable.

Consequently, this thesis has to be content with the inconclusive statement, that the ICONE method does allow successful evolutions of non-trivial animat control in the domain of mid-scale networks, but nothing more.

Though, most experiments mentioned in this thesis have been conducted multiple times with different constraint masks to explore different kinds of controllers. A notable common observation was made concerning the evolvability of mid-scale controllers: Reducing the constraint mask too much or removing it completely, and thus turning the evolution into an (almost) unconstrained structure evolution, always lead to unsolvable configurations without successful solutions. This, at least, indicates the importance of the constraint masks for the evolution success in this domain. A proper proof for this, however, is still outstanding.

9.2.2 Performance of the ICONE Method

As described in the previous section, the performance of the ICONE method is difficult to compare to other neuro-evolution methods. In the literature, such comparisons often rate the method performance according to aspects like,

- the *success rate* of evolution experiments,
- the *number of generations* required to find a solution,
- the overall *number of evaluations* required to find a solution,
- or the *runtime* in seconds to complete.

Being designed with focus on the practical applicability and a good success rate when applied to mid-scale networks, rather than on performance, the ICONE method can, if it would be compared based on classical benchmark problems, certainly not be ranked as one of the top performing algorithms.

One reason is the reference implementation, the NERD Toolkit (Appendix D). This software was designed to allow rapid extensions (plug-ins, scripts) and an easy to use graphical user interface, needed for the preparation of initial networks, the analysis of neuro-controllers and the conduction of interactive evolution. This overhead naturally has

a negative impact on performance compared to compact console programs implementing only the plain algorithm in a native programming language such as C.

A second reason for a lower performance is the variation chain with the constraint resolver and the corresponding genome rejection mechanism (section 3.3). The mutation phases of other NE methods are confined to the rapid mutation of a few genes of each individual, whereas during the mutation phase of the ICONE method an often complex interplay of functional, computationally expensive constraints has to be resolved. Moreover, many constraints have to be executed multiple times to reach a resolved state for an individual, or worse, cannot be resolved due to a conflict. Unresolved genomes are then further mutated before a new constraint resolution is attempted, and the whole procedure can be repeated multiple times. In detail, the mutation of a single individual takes c_f non-trivial function calls, where

$$c_f = c_m(n_m + c_c n_c); \qquad c_m = 1, \ldots, max_m; \qquad c_c = 1, \ldots, max_c, \tag{9.1}$$

with c_m being the number of calls of the variation chain, n_m the number of mutation operators, c_c the needed number of constraint function calls per resolution attempt, and n_c the number of constraints in the genome. max_m and max_c are adjustable parameters of the algorithm and determine the maximal number of applications of the evolution operators per individual and the maximal number of constraint executions during a single constraint resolution attempt. So, depending on the settings of max_m, max_c and the number of involved constraints, the creation of a new generation can, in the worst case – i.e. involving individuals that cannot be satisfactory mutated and thus are discarded – require several *minutes* to complete. Of course, in the mean case the performance is much better, because the variation chain and often also the constraint resolver are only executed once per individual.

However, this should illustrate that, the more interacting constraints are used – and herewith the main search space restriction of ICONE –, the slower the creation of a new generation becomes, caused by a larger n_c, a higher probability for needing more constraint executions per resolution attempt (a higher c_c) and an increasing probability for irresolvable individuals.

In practice, this overhead only is considerable when the evaluation of individuals is very fast. In that case, the time for the generation creation is disproportionately high compared to the evaluation time and accordingly slows down the evolution progress significantly. In the domain of neuro-evolution for complex animats, this is usually not the case, because there, the evaluation of a single individual already requires – due to the hereby involved necessary physical simulations – substantially more time than the creation of a new generation.

In the context of mid-scale networks, the performance overhead originating from the constraint resolver also is outweighed by the increased focusing on promising search space areas, which helps to prevent many superfluous evaluations. And because evaluations are considered costly in this domain, the avoided unnecessary evaluations should more than compensate for the overhead of the constraint resolutions. The real strength of the

ICONE method is not its computational performance, but its ability to find successful neuro-controllers in the domain of mid-scale networks, where most other neuro-evolution methods have severe problems of finding solutions at all.

9.2.3 Modularization with the Network Editor

The modularization phase of the method with the definition of constrained initial networks may look complicated and time-consuming at first glance. So, one might expect that it takes a long time before a planned experiment can actually begin. It is clear, that the modularization requires experience, so – as always – the first steps are the most difficult ones. However, the creation of initial networks soon becomes an appreciated routine, once the main constraints and network tags are familiar.

The difficulty of the modularization, admittedly, depends strongly on the involved software tools. Trying to modularize a network using a plain matrix representation or a text file (e.g. XML) to describe groups, modules, network tags and constraints, would certainly be error-prone, time-consuming and frustrating. On the other hand, a graphical, feature rich network editor, like the one shipped with the reference implementation of the method, can significantly help to keep a clear view[1] on the network organization and on the effects of the used constraints. Ideally, an editor like this is not a separate tool, but instead tightly integrated with the animat simulator and the evolution system, so that networks can directly be tested in the context of a physically grounded animat and with respect to mutations during evolution. This helps to avoid conflicting constraints and misconfigurations of the constraint masks, so that invalid evolutions can be minimized. The inclusion of additional supportive tools, that help to cope with the larger networks, their configuration and their analysis, can significantly reduce the time needed for the modularization phase.

9.2.4 Bootstrapping the Evolution

Finding a promising search space region with adequate gradients in the corresponding fitness space as soon as possible is a crucial goal for any evolution experiment. If the individuals of a generation are distributed inappropriately, i.e. in regions of the search space without such gradients or with undesired local optima, it is very difficult and time-consuming to finally converge to the anticipated solution. Evolutionary algorithms belong to the gradient ascent algorithms and consequently do not work well without suitable gradients (see section 2.2). Accordingly, the strategy of how to move the evolutionary search initially into promising search space regions strongly influences the success of an evolution experiment. This initial localization of the first gradients is often called bootstrapping (compare section 2.4). In the context of neuro-control, this bootstrapping means to find neuro-controllers that show the first signs of the desired behavior and thus start to differ-

[1] Examples of the graphical network representations, as they are produced by the network editor of the NERD toolkit, can be found in sections 7 and 8. All depicted network graphs of that section have been taken from the editor with only minor modifications.

entiate in a meaningful way with respect to the fitness measure. Before this happens, the evolution cannot follow gradients and any progress is prevented.

Bootstrapping evolution experiments with ICONE is facilitated with several mechanisms. A major impact on the distribution of the initial individuals has obviously a properly prepared initial network with its constraint mask. Modularized and suitably constrained, such an initial network only produces new individuals that lay within the search space defined by its constraint mask, even in the first generation. This can significantly increase the chance that one of the created individuals is in a promising region of the search space, compared to an initial generation based on a pure random sampling. Notably useful here is the extensive use of peripheral structures with predefined functional subnetworks, which further reduces the number of mutations required to find a (partially) working network.

In this connection, initial networks should be configured such a way, that they spawn only individuals with *really* promising network structures, i.e. networks that at least provide all required network elements to actually be able to solve the problem. Different from most NE methods, networks in ICONE can be mutated multiple times per generation if required to resolve all constraints. This also holds for the first generation. As a result, with suitably strict constraints, the first generation can already contain individuals with a degree of mutation that normally would need several generations to develop. This acts as a kind of short-cut to instantly move the search to a valid search region, hereby avoiding the evaluation of several generations of 'defective' intermediate network configurations, that simply do not provide the necessary network elements for the task.

Consequently, it is not sufficient to outline the expected topological specifications and start with a *minimal* network within that constraint mask. It is better to provide additional constraints to enforce the existence of *plausible* (not minimal) initial network structures right in the beginning. This can be achieved with constraints affecting the connection density of modules, the specification of synaptic pathways, peripheral structures or by enforcing directed paths between neurons that are expected to be connected. This contradicts the wide-spread suggestion of complexifying algorithms to start with minimal networks (Stanley and Miikkulainen 2002a), i.e. without processing neurons. A better suggestion would be to start with (minimal) *plausible* networks, or even better, using a network configuration with a *minimal distance* to an *assumed* solution. This reduces the number of mutations needed to find a solution and thus can be expected to evolve faster compared to starting with a minimal network.

With these measures, the probability of finding promising networks in the first generation can be increased considerably. In such a configuration, an additional strategy can be used to increase the bootstrapping success. The population size of the first generation can be set very high, so that the initial search space is sampled densely. Starting with the second generation, the population size can be reduced again to the actually desired population size to reduce the evaluation effort for the remaining generations. This approach can also be combined with shorter evaluation times in the first generation, so that the overall evaluation time of that first generation is only marginally longer than that of the other generations. This is viable, because the first generation merely serves as a first initial

probing of the search space. So, a long evaluation is not required here. The evaluation just has to be long enough to identify those individuals that show the first signs of a suitable behavior and thus represent candidates for promising search regions.

Experience with the conducted experiments shows that the combination of these measures indeed helps to find solutions faster and to reduce the number of evolution runs that entirely fail.

9.3 Quality of Evolved Neuro-Controllers

This section discusses how and to what extent the ICONE method influences the quality of evolved neuro-controllers. The term *quality*, here, refers to the value a resulting neuro-controller has for the experimenter. This may include any of the following: The comprehensibility of the controller, the knowledge gained from it, its performance with respect to an experiment and the reusability of the results.

9.3.1 Influencing and Biasing the Network Topology

Networks evolved with the ICONE method can be biased towards specific solution approaches, network topologies and organizational heuristics in many ways. Therefore, the resulting neuro-controllers are often closer to what the experimenter is looking for, which increases the value of the results for the experimenter. In most other NE methods, the experimenter has only limited control over the evolution process and can only hope for the emergence of certain kinds of solutions. So the role of the experimenter is a quite passive one. In difference, ICONE can – without losing the capability of evolving surprising solutions – also be used as a tool to confirm very specific neuro-control approaches, which can, in such cases, increase the quality of the evolved controllers.

The ability to bias the search also makes it easier to search for variants of controllers. Instead of waiting for the evolution to come up with different solutions, one can intentionally change the constraint mask (slightly) and influence the focus of the search. Starting in different subspaces of the search space increases the probability to converge to different solutions, so the variability of controllers can be increased. This is especially useful in the context of neuro-cybernetics, where the goal is not only to find a working controller for a particular application, but also to investigate different ways to realize this. Biasing the search also allows a search for variants in a rather systematic than a purely random way, as has been shown in the example in section 8. Furthermore, this leads to a larger number of different controllers (**?**) and therefore often to an increased quality.

9.3.2 Comprehension of Evolved Neural Structures

The quality of evolved neuro-controllers with respect to their comprehensibility is increased by the ICONE method as part of several (side-) effects. First, complex neuro-controllers evolved with ICONE often involve large parts of peripheral structures. These peripheral structures are usually well understood, so the – at first glance – complex networks often contain only quite manageable additional network structures. Subsequently, the experi-

menter can concentrate on understanding these new structures and their interaction with the already known peripheral structures.

Second, the constraint masks often keep the complexity of the evolved focus structures low, for instance by forcing symmetries, module repetitions, synaptic pathways and the like. Knowing the applied constraints simplifies the identification of network areas with relevance for a behavior and may be used to guide the analysis of a network. For instance, if repeating structures (or symmetries) are involved, it makes sense to analyze one instance of these structures first to understand its function, and then to examine the other instances (e.g. for their role), already knowing the underlying function. When using highly constrained, evolvable neuro-modules, the possible evolvable functions of such modules are also often limited. So, knowing the constraints used for a particular module can point one to the underlying neural principles and speed up the analysis.

In addition, many experiments are constrained towards a specific control approach. Herewith, the experimenter often has a certain expectation of the outcome that certainly helps the experimenter during the analysis.

A final positive effect should be mentioned. The analysis of larger neuro-controllers always requires a way to visualize a network. For ICONE, a comfortable neural network editor is provided (Appendix D.2), which allows a graphical visualization and layout of the networks and helps to understand the networks visually by providing suitable layouts for the controllers. In this context, the constrained neuro-evolution approach additionally helps to keep controllers comprehensible. Because all networks are based on a set of modularized initial networks, a basic layout is inherent to all evolved controllers. Accordingly, the locations of the sensor and motor neurons and the overall organization of the networks in modules and groups are preserved throughout the evolution process. In addition, many constraints also affect the layout of network elements (e.g. symmetries, cloning), which also applies to newly inserted network elements. This makes it easier to instantly get an overview on an evolved network, its basic organization and apparent regularities. Of course, newly inserted network elements are not neatly laid out – which is not possible because of the required domain knowledge –, but they are at least placed within the layout frame formed by modules and constraints. This frame usually reflects the principle network organization, so that the locations of new network elements still give clues about their function and their relation to other network parts.

9.4 Future Work

The experiments conducted with the ICONE method up to now are still only the first steps to explore the possibilities of network shaping and search space control. Many future experiments can be expected, not only with other complex animats, but also with a focus on novel questions. This is facilitated by the guidance abilities of the ICONE method that simplifies the focusing of the search on very specific questions concerning network topology, network organization and the realization of specific approaches.

Further experiments may also focus on new constraints and additional extensions of the ICONE method. Chapter 6 already introduced some of these extensions, which should

be thoroughly examined as future work. Especially the extensions regarding the further control over mutations on the neuro-module level, such as module insertions, module replacements, hidden elements and typed connections have a promising potential to enhance the expressiveness of the method.

Another wide field of experiments can be the combination of different neuron models in constrained, heterogeneous networks, so that the flexible combination of approaches like learning rules, higher-order synapses, different neuron models and adaptive synapses may lead to networks with specialized neurons in specialized subnetworks, as they can be observed in most biological nervous systems. Mixing such neuron models during evolution and allowing the free mutation of all related parameters of the models increases the search space severely and therefore requires a search space restriction like the constraint masks of ICONE to reduce the search space back to a feasible dimensionality.

In addition to these suggestions, many further applications, extensions and explorations of the ICONE method can be imagined. In this section, three of these suggestions are explicitly mentioned and are outlined briefly.

9.4.1 Increasing Control Over Weight Mutations

One of the most crucial problems of the evolution of larger networks is the search for suitable synaptic weights. Many controllers require quite large synaptic weights ($w \gg 1$) to amplify signals or to inhibit neuron activations. On the other hand, interesting neurodynamics often require comparably small weights ($w < 1$), because if only large weights are used, then the network dynamics are mostly generated as an interplay of predominantly saturated neurons, which only represents a very small class of possible network dynamics. So, both kinds of synapses are useful, but it is difficult to find proper settings of the evolution operators to generate and preserve both kinds of weights. In practice, if the development of larger weights is not explicitly prevented, then most weights converge to a medium strength over time, which is usually already very high, leading to networks with saturation-based dynamics (see many of the controllers in section 8). If, in contrast, large weights are explicitly forbidden, then the useful stronger amplification weights are prevented as well.

Another related problem is the selection of a proper variance setting for weight changes during evolution. Some parts of the networks are often more sensitive to weight changes than others, so that different settings would be required in those areas. A strategy to cope with this problem in *known* network areas was already discussed in section 9.1.5. But for synapses that are added during evolution, this measure does not work out.

Therefore, additional strategies should be examined for the ICONE method to gain more control over the weight distribution of an evolving network. One suggestion would be the introduction of new constraints with focus on the weight distribution. Such constraints could control the weights within a network according to some heuristics, e.g. to keep the sum of all weights within a module constant. Such a constraint could use network tags to influence the weight modification operator to prefer an increase or a decrease of weights. So, when a weight gets stronger, then the probability for a subsequent decrease of another

weight in the module is raised, so that a balance between strong and weak weights is maintained.

Another strategy would be to change weights proportional to their strength, so that small weights are only slightly changed and large weights by larger amounts. This would prevent the rapid change of a small weight into a large weight, but would also allow fast changes of strong weights. Then, allowing even very large weights ($w \gg 100$) would not be a problem.

A third strategy could be the introduction of different synapse classes. When a new synapse is added, a special type tag could be added, that groups the synapse into a group (e.g. strong, medium, small, fine). Then, corresponding mutation operators could mutate the different classes with distinct sets of parameters settings, so that the weight range, change probability and variance may be selected individually for each class. These operators could also allow a change of the weight class as a mutation variant, so that the classes can be varied as well. Such synapse classes then could also be enforced for certain neuron-groups with constraints.

9.4.2 Onion Skin Evolution

Working with constraint masks has indicated another interesting aspect. When looking for controller variants, it does not seem so important what kind of constraint mask is used, as long as the constraint mask is restricting the search space to a feasible size. Evolution will, if proper controllers are sufficiently numerous, find a solution. That solution, however, is in most cases the most likely, prevalent solution of that search space. So manipulating the search space changes the probability of the findable solutions and accordingly determines, which solutions are dominant and consequently most likely to be found.

Therefore, a possible hypothesis is, that for finding *variants*, it is not so important *what kind* of constraint mask is used, but rather that (differing) constraint masks are used at all, so that each search is performed in a sufficiently small configuration space with differing dominant outcomes.

As a result, a novel evolution strategy could be examined in future work that may be termed '*Onion Skin Evolution*'. The procedure would be as follows: The evolution takes place with an initial constraint mask that restricts the search space to a feasible size. After running multiple evolutions on that search space, solutions should have been found and the dominant approaches should have been identified. Then the constraint mask is altered such, that the dominant solutions are *excluded* or become unlikely to evolve. Then evolution is again run multiple times to find novel solutions in the remaining search space and to identify the now dominant approaches. This is repeated until the initially chosen major subspace of the search space is widely explored. So, like an onion, where skin after skin is removed to get to previously unreachable layers, the search space is more and more reduced by excluding already found approaches, so that other, less likely approaches can come to the surface.

In difference to most evolutions performed with the ICONE method so far, where the constraint masks have been primarily used to *focus on specific* approaches, constraint masks

then would be used primarily to *exclude* approaches without requiring much thought about the remaining possible outcomes.

9.4.3 Evolving More Controllers for Complex Robot Hardware

The described experiments in this thesis are only a small subset of the experiments that can be approached for complex animats with the ICONE method. In addition to the A-Series humanoid, a much more complex humanoid robot, the Myon robot (Hild et al. 2011, 2012), has been used as a hardware platform for numerous behaviors and functional neuro-modules. The Myon humanoid comprises 173 sensors (63 acceleration sensors, 8 force sensors, 62 angular sensors, 40 current consumption sensors), 80 motor neurons to control the torque of the 40 motors and to switch between motor modes (Siedel et al. 2011), and a camera with a corresponding neural interface. Without search space restrictions to drastically limit the number of evolvable parameters, the evolution of neuro-controllers for this class of robots is very unlikely to succeed. Up to now, several behaviors have been successfully developed for this robot, involving highly restrictive constraint masks and peripheral structures, for instance to hide the raw motor torque neurons behind lower-dimensional and easier to use angular controller modules. With the ICONE approach, it is comparably easy to focus the evolutionary search on specific questions without the overhead of evolving already known functions. Consequently, questions can focus on the motor control layer (e.g. position control, force control, antagonistic control, energy efficient control), on sensor processing (pose estimation, instability detection, reliable behavioral state detection, motor stall detection, collision handling), on the coordination and transition of behaviors, or on control paradigms (e.g. with perception control theory (Gräuler 2011; Powers 1973)). Only a small part of the possible (and realistically evolvable) experiments have been approached so far and are left for future work.

Another very interesting class of animats is the class of walking machines of all kinds, such as the Octavio robot (von Twickel 2011; von Twickel et al. 2012). Experiments with such machines have been conducted in several variants, for instance walking insects with six, eight and twenty legs, with a stiff body or with controllable joints between each pair of legs. Hereby, with ICONE, the involved search space does not necessarily grow with the number of legs, which makes the approach scalable even to animats with many legs. With proper constraints, most legs can be controlled with subnetworks that are dependent on other parts of the network, so that the evolvable search space remains quite small. With ICONE, different questions can be systematically approached, for instance by focusing solely on the development of suitable single leg-controllers (not on isolated legs, but merely as part of a fully working animat, e.g. using a clone constraint), on the specialization of leg controllers according to their position and role in the animat (e.g. with symmetry, structure symmetry, module classes), on the leg coordination (e.g. with fixed single leg controllers and connection symmetry), on the integration of body motions (turning and lifting of segments) in addition to the leg motions, on the sensor processing (pose estimation, obstacle detection), on internal states (e.g. short-term memory for obstacles to avoid collisions for the successive legs), or on the development of different gaits and behaviors. These

and many other experiments can be approached by explicitly focusing the search on the required properties with appropriate constraint masks.

To summarize, with the search space restriction and experiment focusing that is possible with the ICONE constraint masks, the animats involved in the evolution experiments can be equipped with more sensors, proprioceptors and actuators and still provide feasibly small search spaces. Therefore, neuro-evolution can also start now to explore the capabilities of sensor-rich, complex robots, which should allow many novel experiments and applications.

Chapter 10

Conclusion and Impact on the Field of Neuro-Evolution

ICONE provides a unified, generalized approach for the evolution of mid-scale neuro-controllers in the domain of evolutionary robotics. It provides measures for the major challenges of neuro-evolution in that domain (see section 2.4), including search space restriction, bootstrapping, network shaping, (interactive) guidance of the evolution, crossover, diversity maintenance, (systematic) variation exploration, exploitation of animat regularities and peculiarities, the incorporation of domain knowledge, and structure reuse / knowledge transfer between experiments. With these measures, larger networks focusing on very specific experimental questions can be successfully approached.

The focus hereby is clearly on the *practical* applicability of neuro-evolution for non-trivial neuro-controllers and specifically on the systematic, guided search for controller variants. The algorithm has shown its practicality in the domain of mid-scale neural networks, which enables novel evolution experiments that have so far been impractical due to the large involved search spaces. However, the features of the ICONE method are not only suitable to reduce the search space and to increase the chance of finding successful neuro-controllers, but they also provide a set of – in this context – novel techniques to structure and control the shape and organization of evolving neural networks. Among these techniques, the guidance of evolution towards specific network organizations and solution approaches has an especially high potential for interesting results, enabling the exploration of new scientific questions. This already allows for a class of novel experiments on its own.

The future experiments are not only expected to cover neural behavior controllers for increasingly complex physical animats equipped with many sensors and actuators, but also to focus on the further understanding of dynamical principles of neuro-control and on the realization of particular functions as neuro-modules, such as different motor control approaches, sensor processing, (short-term) memory and (behavior) coordination approaches. This has the potential to lead to novel insights regarding properties of neuro-controllers with richer neuro-dynamics and more complex neural organizations.

As part of this thesis, a reference implementation of the method has been published as open-source software. With this high-quality software framework the ICONE method is offered as ready-to-use tool for the evolutionary robotics community. Herewith, the ICONE technique can be used directly for experiments without bothering the experimenter with a prior laborious implementation of the method.

Appendix

Appendix A

Network Symbols

Neural networks in this thesis are visualized graphically as graphs. The symbols used in these diagrams are named in figure A.1. The different network elements are described in detail in the corresponding chapters of this thesis. Named neurons, neuro-modules and neuron groups used in the text are printed in a `Typewriter` font to highlight, that these names refer to network elements in a network graph.

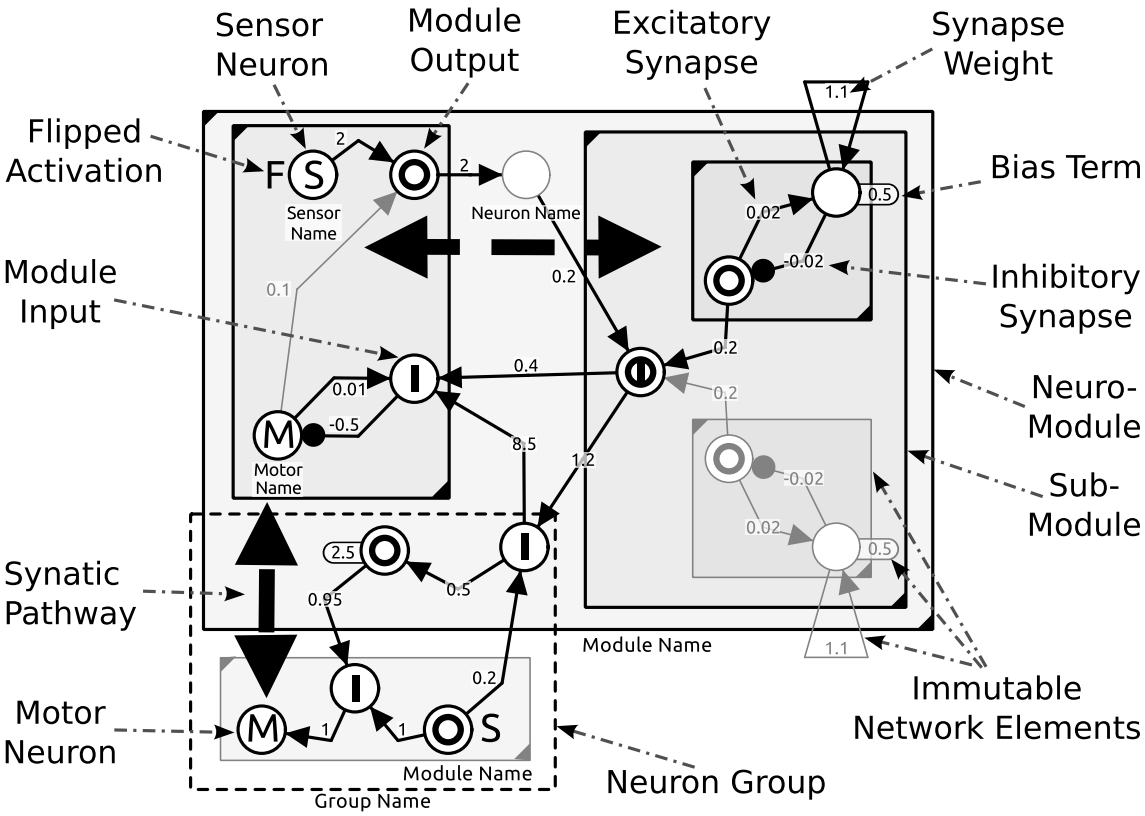


Figure A.1: Symbols used to visualize the neural networks in this thesis.

Appendix B

Mutation Operators

The variation chain of the ICONE algorithm (section 3.3.1) provides a number of mutation operators that modify the network genomes during the variation phase. The operators are executed in a fixed, ordered sequence during each mutation phase. This section describes the mutation operators of the ICONE method, as they are used with the reference implementation of the NERD Toolkit for the experiments of this thesis. The chapter gives details about their function and the various parameters that can be used to influence their behavior. The order, in which the operators are described here, corresponds to their order in the variation chain. In addition to the standard mutation operators, the chapter is concluded by a brief description of some experimental operators that are used to extend the ICONE method with new functions, such as module insertions, hidden elements and mixed neuron models (for an overview on ICONE extensions see section 6).

ICONE Standard Mutation Operators

Initial Network Creation

This operator is part of the genome creation phase, together with the modular crossover and the clone parent operator. This operator is executed for any new individual without parents. This usually only happens when the first initial generation is created.

The operator allows three different ways to describe the preferred network configuration for a new individual. The first two ways, the specification of an initial network or a directory with such networks, is the preferred way to start an ICONE evolution. An initial network is usually modularized and configured with a constraint mask, so that the search is biased and the search space is restricted. Most commonly, a single constrained network is given as a blue-print for the initial generation. That network is cloned for each new individual and hereafter slightly varied by the subsequent mutation operators, so that the new networks differ from each other. Instead of a single network, also a directory with initial networks can be specified. This directory can either contain different alternative initial networks, or more often, the generation of a previously stopped evolution experiment. This allows the continuation of the evolution with every generation of a previous experiment, as long as the networks of that generation are still available.

Network File or Directory	String	The file containing the initial network, or the directory containing multiple initial network files
Preserve Initial Network	Boolean	Determines whether an unchanged version of the initial network should be part of the first generation
Input Neurons	Integer	The number of input neurons (sensors) of the new network
Output Neurons	Integer	The number of output neurons (sensors) of the new network
Activation Function	String	The activation function used for all neurons of the new network
Transfer Function	String	The transfer function used for all neurons of the new network

Table B.1: Parameters of the **Create Initial Network** operator

Alternatively, this operator also can create minimal networks from scratch, which is – in the context of the ICONE method – not recommended because of the missing constraint masks and the herewith involved large search spaces. In this case, a number of parameters have to be specified to define the number of motor and sensor neurons and the used neuron model.

Parent Cloning

This operator is applied to any new individual with at least one parent. It simply copies the genome of the that first parent. Due to the subsequent mutation operators, that cloned network is then lightly modified hereafter.

Modular Crossover

The crossover operator is only applied for new individuals that provide at least two parents. The operator exchanges modules of the genome, which was cloned with the previous mutation operator, with modules of the second parent. Details of the modular crossover can be found in section 3.3.3.

The operator has two parameters: The first one specifies the probability of whether the crossover operator is applied to a new individual. If so, then the probability of each exchangeable module to be exchanged is given by the second parameter. This global probability setting can be overwritten locally with network tags on the modules, so that each module may have a differing probability to be exchanged.

Neuron Removal

This operator removes neurons from the network. This only affects neurons whose existence is not protected by network tags or by their owner module. The latter is the case when a

Crossover Probability	Float [0,1]	The probability that the crossover operator is executed
Probability per Module	Float [0,1]	Probability per module to perform the crossover with a compatible module

Table B.2: Parameters of the **Modular Crossover** operator

module itself is protected against structural changes, which also excludes the removal of neurons.

Number of Tries	Integer	The maximal number of neurons that can be removed during a single call of the variation chain
Probability per Try	Float [0,1]	Probability per try to remove a neuron

Table B.3: Parameters of the **Remove Neuron** operator

The parameters of the operator specify the maximal number of removal attempts for each execution of the operator, and the probability for a successful removal per attempt. If a removal should take place, then an unprotected neuron is chosen at random and is removed.

Synapse Removal

This operator removes synapses from the network. This only affects synapses whose existence is not protected by network tags or by their owner module. The latter is the case when a module itself is protected against structural changes, which also excludes the removal of synapses.

Number of Tries	Integer	The maximal number of synapses that can be removed during a single call of the variation chain
Probability per Try	Float [0,1]	Probability per try to remove a synapse

Table B.4: Parameters of the **Remove Synapse** operator

The parameters of the operator specify the maximal number of removal attempts for each execution of the operator, and the probability for a successful removal per attempt. If a removal should take place, then an unprotected synapse is chosen at random and is removed.

Bias Removal

This operator removes bias terms from neurons. This does only affect neurons whose bias is not protected by network tags.

Number of Tries	Integer	The maximal number of bias terms that can be removed during a single call of the variation chain
Probability per Try	Float [0,1]	Probability per try to remove a bias term

Table B.5: Parameters of the **Remove Bias** operator

The parameters of the operator specify the maximal number of removal attempts for each execution of the operator, and the probability for a successful removal per attempt. If a removal should take place, then an unprotected bias term is chosen at random and is removed.

Neuron Insertion

This operator inserts neurons to the network. For this, the operator first determines the number of new neurons that are actually inserted to the network during the current execution of the operator. Each new neuron is then placed at random positions in the network. New neurons can only be inserted to modules that are not protected against structural changes and that do not violate given size restrictions, which both can be specified with network tags on the modules. Neurons that cannot be placed at a valid place due to restrictions are discarded.

The distribution of new neurons can be influenced with network tags on modules. With such tags, modules can be configured to be more often or less frequently considered for a neuron insertion.

The transfer and activation function of a new neuron is derived from the default settings of the network. However, both functions can be changed by constraints of the neuro-module to which the new neuron was added to, so that heterogeneous networks with coexisting neuron models can also be evolved.

Number of Tries	Integer	The maximal number of neurons that can be inserted during a single call of the variation chain
Probability per Try	Float [0,1]	Probability per try to add a new neuron
Add to Module Probability	Float [0,1]	Probability per inserted neuron to add it to a module (default is 1.0)
Max Network Size	Integer	The maximal number of neurons in the network
Initial Connection Proportion	Float [0,1]	Determines the proportion of valid synaptic connections that can be added to the new neuron during the initial connection phase (default is 1.0)

Table B.6: Parameters of the **Insert Neuron** operator

A new neuron without synaptic connections is quite useless. To avoid a long delay between a neuron insertion and the development of new synapses connecting the new

neuron with the remaining network, each neuron is initially connected with the network. These initial synapses are inserted by the specialized `Synapse Insertion` operator. To trigger such an initial connection in that operator, each new neuron is tagged with a special network tag. This tag specifies the maximal desired connectivity of the new neuron, i.e. the maximal number of synapses that may develop during that initial connection step as a proportion of all valid synapses for that neuron.

Synapse Insertion

This operator inserts synapses to the network, considering module boundaries, neuron visibilities, protection tags, synaptic pathways and other limiting or heuristic rules.

Number of Tries	Integer	The maximal number of synapses that can be inserted during a single call of the variation chain
Probability per Try	Float [0,1]	Probability per try to add a new synapse
Initial Insertion Probability	Float [0,1]	Insertion probability per possible initial synaptic connection of new neurons

Table B.7: Parameters of the **Insert Synapse** operator

The operator runs in two phases. First, *initial connections* of new neurons are created. New neurons are identified by a special network tag that specified the proportion of initial connections for the neuron. For each such neuron, the operator determines, which synaptic connections are valid according to the rules and restrictions mentioned above. Based on these valid connections, the maximal number of new synapses for that neuron is derived as the proportion of the number of valid connections. The probability for each new synapse to be actually inserted is given as a parameter of this operator. That probability differs from the probability for the insertion of other synapses, so that the initial connections can be controlled independently from the normal synapse insertions. This is important, because the insertion of initial synaptic connections should usually be much more probable than the insertion of other synapses. This ensures, that new neurons are rapidly connected to the network without the need to have a high synapse insertion probability for existing network structures, which would lead to densely (or fully) connected networks in a short time. The network tag for the initial connection is removed by this operator, so that initial connections are only added once.

In the second phase, the operator inserts synapses to neurons that have not been newly inserted to the network. The maximal number of insertion attempts per operator execution can be specified as a parameter of the operator. For each such attempt, the operator decides based on the given insertion probability, whether a new synapse is inserted. If so, a valid synapse is added to the network. Hereby, the above mentioned rules and restrictions are considered.

The synapse function of new synapses is taken from the default settings of the network. Constraints on neuro-modules can modify the synapse function, so that also heterogeneous networks with mixed synapse functions are possible to evolve.

Bias Insertion

This operator adds new bias terms to neurons. It only affects neurons without a bias term, that are not protected against bias insertions with network tags.

Number of Tries	Integer	The maximal number of neurons that can get a new bias term during a single call of the variation chain
Probability per Try	Float [0,1]	Probability per try to add a new bias term

Table B.8: Parameters of the **Insert Bias** operator

The number of bias insertion attempts per operator execution can be specified as a parameter of the operator. For each attempt, the operator decides based on the probability parameter, whether a new bias term is inserted. If so, then a valid neuron is chosen at random and a new bias term is added. Each new bias is set to 0.0 and has to be initialized by a subsequent operator.

Synapse Initialization

This operator initializes all synapses with a weight of 0.0 with a random value in the given range of [Min, Max]. This range can be influenced for each separate neuro-module or neuron-group with network tags, so that synaptic weights can be sensitive to a context where domain knowledge is available.

Min	Double	The lower boundary of the range of initial weights
Max	Double	The upper boundary of the range of initial weights

Table B.9: Parameters of the **Initialize Synapse** operator

Bias Term Initialization

This operator initializes all bias terms with a weight of 0.0 with a random value in the given range of [Min, Max]. This range can be influenced for each separate neuro-module or neuron-group with network tags, so that bias terms can be sensitive to a context where domain knowledge is available.

Min	Double	The lower boundary of the range of initial weights
Max	Double	The upper boundary of the range of initial weights

Table B.10: Parameters of the **Initialize Bias** operator

Bias Term Modification

This operator modifies existing bias terms of neurons. It only affects neurons that are not protected against bias changes via network tags. For each neuron with a bias term the application of the operator is separately determined according to the application probability. If the bias term is modified, then the value of change is calculated using a Gaussian distribution with the given standard deviation. That value is then added to the bias term. The standard deviation and the change probability can be locally influenced per neuron or neuro-module with network tags.

Probability per Bias	Float [0,1]	The change probability for each bias
Deviation	Double	The standard deviation for bias changes
Min Bias	Double	Minimal bias of changeable bias terms
Max Bias	Double	Maximal bias of changeable bias terms
Re-initialization Probability	Float [0,1]	The probability to randomly re-initialize a bias term.

Table B.11: Parameters of the **Change Bias** operator

Parameters Min and Max define an absolute limit for bias terms that cannot be violated by changing a bias. This only applies when a bias term is changed, so protected or initial bias terms may violate these limits.

The `re-initialization probability` parameter defines a (low) probability that a bias term is randomly re-initialized with a new random value. This allows rare, but rapid bias changes.

Synaptic Weight Modification

This operator works similar to the `bias term modification` operator and changes the weight of synapses.

Probability per Weight	Float [0,1]	The change probability for each synaptic weight
Deviation	Double	The standard deviation for weight changes
Min	Double	The minimal weight of all changeable synapses
Max	Double	The maximal weight of all changeable synapses
Reinitialization Probability	Float [0,1]	The probability to randomly reinitialize a weight

Table B.12: Parameters of the **Change Synapse Weight** operator

Experimental Mutation Operators

Neuro-Module Insertion

Inserts a neuro-module from the neuro-module library (5.1.3). In the library, only neuro-modules that are relevant for the experiment should be enabled for insertions, so that the search space is kept small. Neuro-modules are only inserted as submodules, if their insertion is valid for the target module. This can be influenced with network tags on the modules. Each module can separately specify lists of permitted or prohibited module types (details on module types can be found in section 4).

As a variant, this operator can also exchange existing modules of the network by compatible neuro-modules from the module library. This allows the evolutionary search for the best suited alternative for a module, when multiple versions of that module or neural function are available in the module library.

Higher-Order Synapse Insertion

This operator inserts higher-order synapses (Koch and Poggio 1992), i.e. synapses that connect the output of a neuron with another synapse, instead of another neuron. These higher-order synapses then modulate the function of the target synapse, for instance to regulate the weight of a synapse with the dynamic output of the neuron.

Hide / Uncover Network Elements

This operator is part of the *Hidden Elements* extension (section 6.3) and hides (disables) or uncovers (enables) network elements, such as synapses, bias terms or entire neuro-modules. The effect of hiding is similar to that of the removal operators, with the difference, that the hidden network elements remain being (a disabled) part of the network and thus can be uncovered again later. So the main difference is when the operator *uncovers* hidden elements, which is – as effect for the network – similar to the insertion operators. In difference to a random insertion, the uncovered structures are not new network elements, but instead the previously defined, fully configured network elements. This makes it possible to explore different permutations of predefined (complex) network structures, instead of trying to find the network structures with random mutations.

Change Synapse Functions, Activation Function and Transfer Function

These three operators can be used to evolve attributes of the network models for neurons and synapses of the network. This leads to heterogeneous networks, in which the neurons and synapses may have different underlying network models (activation function, transfer function, synapse function).

Appendix C

Network Tags

Network tags (introduced in section 3.2.4) are simple string-string pairs that can be attached to network elements to influence the network function, the evolution process, the function of mutation operators and other aspects of the network handling in various ways. This section lists the major network tags used in the NERD toolkit – and herewith for the experiments of this thesis – and describes them in detail. The list ignores a number of additional minor network tags that are used to exchange or leave information on the network genome, for instance during the mutation phase, the constraint resolving phase or for analysis purposes.

Each network tag can have a parameter string to specify required attributes. Although every network element can, in principle, be tagged with any network tag, most tags only make sense when used with a specific network element. Therefore, these valid target elements are listed for each network tag. Network tags are given in the following format:

Tag Name(s)	**Parameter Name (Type)**	**Valid Network Elements**
	Description	

Network Tag Prefixes

Network tags support an extensible, simple prefix system to influence, how a network tag is treated when constraints manipulate tagged network elements or when conflicts between network tags arise. Because network tags are only simple attached information that is used by various external program parts, such as mutation operators, the network editor, code exporters and functional constraints, such conflicts can in rare cases lead to unexpected behavior. In these cases, tag prefixes can help to preserve a deterministic network setup without unexpected side-effects.

_	Hidden	Network tags starting with an _ are hidden and not shown directly to the user. Such tags are often just ancillary tags that are usually not directly defined or manipulated by the user.
+	Essential	When using the tagged element as reference for a copy procedure (e.g. cloning, symmetry), the tag is copied to the dependent elements.
-	Singular	When using the tagged element as reference for a copy procedure, this tag is not allowed in the dependent elements.
!	Major	If more than one tag with the same name is found in a group of elements, then the setting of this tag is prefered.
%	Permanent	If a module is replaced during evolution, then this property is copied to the replacement of the module.

Table C.1: Network Tag Prefixes

Protection Tags

Protected		Any Element
	Protects all attributes of the network element, including its existence. For neurons, no synapses can be directed to that neuron. Modules do not allow the insertion of additional modules and neurons.	

ProtectExistence		Any Element
	Prevents the removal of the network element during evolution.	

ProtectBias ProtectWeight		Neuron, Synapse
	Prevents changes to the neuron bias or synapse weights.	
ProtectActivationFunction ProtectTransferFunction ProtectSynapseFunction		Neuron, Synapse
	Prevents all changes to the specified aspect of the neuron model.	

NoSynapseSource NoSynapseTarget		Neuron, Synapse
	Prevents a neuron from being the source of a synapse, and prevents a neuron (or higher order synapse) from being the target of a new synapse.	

Hints for Mutation Operators

InitConnectionProportion	Proportion [0,1]	Neuron
	Specifies the proportion of new *initial* connections that may maximally be added. This tag is automatically added to any new neuron to initially connect it to the network, but it can also be used manually to foster stronger initial connections. The tag is removed right after the synapse insertion phase, so it affects a neuron only once.	

SynapticPathways	List of Ids	Network
	Specifies a list with pairs of module ids that specify pathways at which synapses are valid to be added. Once this tag is present in a network, then connections between modules are confined to those between the given pairs. Each pair consists of the source followed by the target module, so to allow bidirectional connections between two modules, two separate pairs have to be specified.	

MaxNumberOfNeurons	Maximum (int)	Module
	Specifies the maximal number of neurons that are allowed in the neuro-module.	
MaxNumberOfModules	Maximum (int)	Module
	Specifies the maximal number of submodules that are allowed in the neuro-module.	

ModuleType	List of Type Names	Module
	Defines the types the module belongs to. This information is used to find exchangeable modules during crossover or for module replacements.	
CompatibilityList	List of Type Names	Module
	Lists all types the module is compatible with. This information is used for crossover and other module replacements.	
Role	Type Name	Module
	Specifies a type that has to be used in the specific place the module is used at. If replaced, the new module is tagged with the same role tag, so that this tag remains at the specific location.	

Evolution Control Tags

ChangeProbability	Probability [0,1]	Any Element
	Overwrites the global mutation probability for the main parameter of a network element (bias term, weight, neuron insertion) with the new probability. This probability may be either given as an absolute value or as percentage of the global setting.	

ChangeDeviation	Deviation (double)	Any Element
	Overwrites the global deviation setting for mutations of the main parameter of a network element (bias term, weight, neuron insertion) with the new deviation. This deviation may be either given as an absolute value or as percentage of the global setting.	

Min Max	Limit (double)	Synapse, Neuron
	Overwrites the global range limitations for a weight (synapse) or a bias term (neuron).	
InitBiasRange InitWeightRange	Range [min,max]	Module
	Overwrites the global range for the initialization of newly inserted weights and bias terms for all neurons and synapses of a module.	

NeuronInsertionProbability NeuronRemovalProbability	Probability (double)	Module
	Increases of decreases the neuron insertion or removal probability with respect to the other modules.	
SynapseAdditionProbability SynapseRemovalProbability	Probability (double)	Neuron, Module
	Increases of decreases the addition or removal probability for the tagged neuron or the neurons of a tagged module with respect to the other neurons.	

Functional Tags

ODN	Level (int)	Neuron
	Converts the affected neuron into an *order dependent neuron* that is executed on the given level.	

Flip		Neuron
	Reverses the output of the neuron.	

Input Output		Neuron
	Marks the neuron as sensor (input) or motor (output) neuron. Such neurons are directly connected to the sensors and actuators of the animat.	

ModuleInput ModuleOutput	Visibility Depth (int)	Neuron
	Marks the neuron as interface neuron (input and output) for a module. The visibility depth of the interface neurons can be further specified. The default visibility level is 1.	

Identification and Tracing Tags

EQ_VAR	Variable Name	Neuron, Synapse
	Defines a named variable within the scope of a *Network Equation* constraint. The variable refers to the bias term (neuron) or weight (synapse) of the tagged element.	
EQ	Equation	Neuron, Synapse
	Defines an equation within the scope of a *Network Equation* constraint to calculate a synapse weight or bias term. The equation may use any arithmetic operation and access all variables defined with EQ_VAR in the same equation context.	

_RDOF	List of Dependencies	Any Element
	This (hidden) tag (reduced degrees of freedom) is automatically updated by the constraint resolver and holds a list of all dependent parameters for each network element. This tag is useful to visualize the search space and to derive the number of evolvable parameters of the network. The dependency list simply contains a character per dependent parameter, such as e (existence), c (content), b (bias), w (weight), t (transfer function), a (activation function), s (synapse function).	

Accelboard SpinalCordAddress	Identifier	Module
	The Accelboard tag associates a neuro-module with a specific AccelBoard on the A-Series robot (see section 7). The SpinalCordAddress associates a neuron with a hardware address for a motor or sensor variable on the hardware. Both are required for an export of NERD networks to the native code of the A-Series hardware.	

BDN_In BDN_Out	ID	Neuron
	Defines an input or output address to convert a NERD network into a BrainDesigner network, used to operate physical robots like the A-Series or the Myon humanoids. The hardware addresses are required to connect the network to the physical hardware.	

_Location	Coordinates (x,y,z)	Any Element
	This (hidden) tag holds the current position of the network element in the network editor. The position tag is automatically adapted when the network element is manipulated in the editor. The position information is not only needed for the visualization, but also for mutation operators working on the relative positions of network elements.	
_ModuleSize	width, height (double, double)	Module
	This (hidden) tag holds the size of a module. This is primarily used for layout purposes, but can also be used by mutation operators to decide where to put new neurons (e.g. by adding more neurons to larger modules)	

_CreationDate	Generation (int)	Any Element
	This (hidden) tag is automatically added and keeps track on the generation at which the element has been added to the network.	

_ModuleOrigin	List of Individual IDs	Module
	This (hidden) tag maintains a list of individual ids that can be used to reproduce the migration path of a module from individual to individual through crossover.	

Auxiliary Network Tags

Layer	Layer (int)	Any Element
	Assigns a network element to one or more named layers. The NERD network editor allows to hide selected layers to increase the readability and handling of large networks.	

FadeInRate	Milliseconds	Motor Neurons
	Only for exported networks. Replaces the motor neuron with a special neuron type that allows a slow fading in of the torque to protect the robot hardware from harmful initial conditions.	

Appendix D

Neurodynamics and Evolutionary Robotics Development Toolkit

As part of this thesis, the *Neurodynamics and Evolutionary Robotics Development Toolkit* (NERD Toolkit) has been developed since 2008. The toolkit is still under constant development and available as an open source project at nerd.x-bot.org under the general public license (GPL). At the time of this writing, the NERD Toolkit is available in version 3.5 Rev.2790. As project lead, the overall concepts and software organization have been developed and supervised by myself. The implementation of the now over 150K lines of code was assisted by Verena Thomas (03.2008 - 02.2009), Ferry Bachmann (09.2008 - 01.2010) and other contributors (a complete list of contributors and details on their contributions can be found in the project folder at nerd.x-bot.org).

The NERD toolkit was designed to provide libraries and applications to rapidly design experiments in the context of evolutionary robotics and neuro-control. Such experiments can either be developed within a set of given applications, that allow the evolution of neuro-controllers for simulated animats, or can be implemented as custom applications based on the NERD libraries. The first approach is fast and provides all common tools needed for the typical experiments in the context of our workgroup. Due to a scripting interface, the experiments can be designed solely with QScript (ECMA-262 Standard 2009), so no compilation of C++ code is required. The second approach for more experienced users allows very specific, non-standard experiments in C++ without the need to (re-)implement and test common software parts, like visualizations, monitoring tools, graph plotters, neural networks, evolution algorithms or the simulation of rigid body systems. In the current version, the libraries comprise a scriptable simulator for animats, an extensible neural network library with a comfortable network editor, a flexible (neuro-)evolution framework with several evolution algorithms and diagram plotters for the analysis of neural networks with methods of dynamical-systems theory. The next sections describe the main components used for the experiments of this thesis. Further details can be found at the nerd homepage at nerd.x-bot.org and in Rempis et al. (2010).

D.1 NERD Simulator

A simulator is an essential part of the evolution of neuro-controllers for animats. Because of the many necessary evaluations, it is unpractical to test controllers directly on hardware during the evolution. The effort for maintenance and (comparable) setup of the experiments would be very high and time-consuming. With a simulator, thousands of evaluations can be performed parallel and often much faster than in real-time without the need to have physical robots at all. Also, even such animats can be simulated, that cannot practically be built as physical machines, thus increasing the range of possible experiments (e.g. by including morphology co-evolution (Bongard 2010; Bongard and Paul 2001; Dellaert and Beer 1994) or artificial ontogeny (Bongard and Pfeifer 2003)).

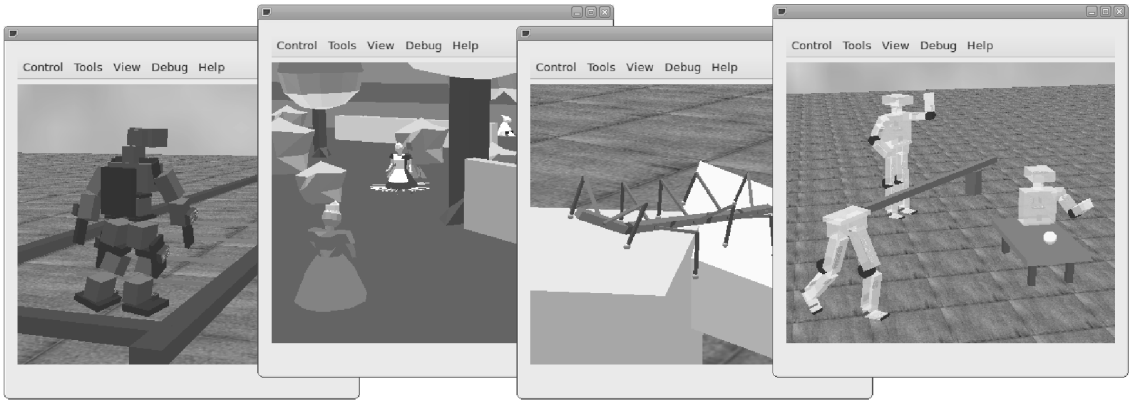


Figure D.1: Examples of scripted evolution experiments with the NERD simulator.

The performance of a simulator strongly depends on the underlying physics engine, the level of simulated detail, and the degree of accuracy of the simulation. Obviously, the more accurate and the more complex a simulated scenario is, the longer its computation takes. The evolution of controllers for physical robots requires highly accurate simulations, because most often evolved neuro-controllers do not work on the physical robot if the simulation details differ too much. On the other hand, many experiments, for instance ones that examine network organizations or qualitative tasks (e.g. short-term memory or behavior coordination), allow a much less accurate simulation or sometimes do not require a physical simulation at all. Such experiments can be simulated much faster, which reduces the runtime of the evolution experiment. Therefore, the NERD simulator does not include a fixed physics engine, but instead implements a physics abstraction layer, that allows a simple switching between physics engines without the need to change the descriptions of the experiment scenarios. The simulation performance can be adapted to the accuracy requirements by switching between different rigid body physics engines (IBDS (Bender 2007), ODE (Smith 2001)), a simple 2D physics engine, a simple geometric collision model or the representation of simulations by equations only.

The simulator provides a visualization engine with different camera perspectives based on the *Open Graphics Library* (OpenGL, Segal and Akeley 1993), that works with the

representations of the physics abstraction layer and therefore is compatible with all physics options. All parameters of the simulation, including all parameters of the simulated objects, can be viewed, changed and plotted over time (see figure D.2), so that the manipulation, analysis and setup of simulations is fast and simple.

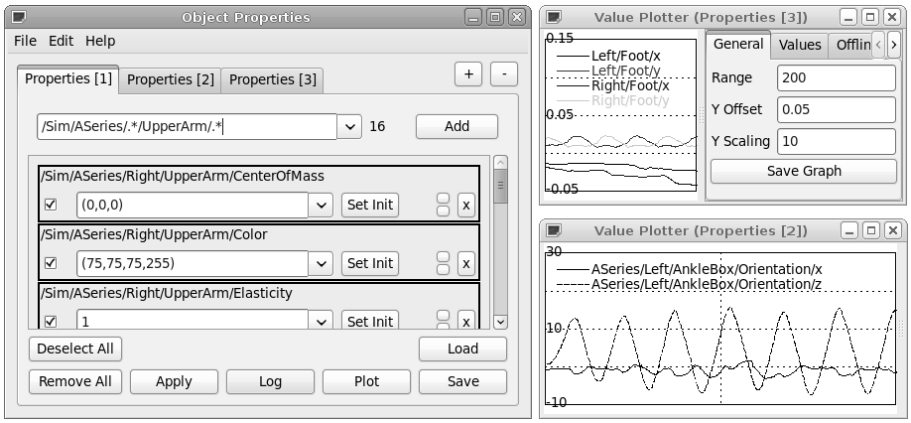


Figure D.2: The graphical user interface to manipulate, observe, plot, log and examine all parameters of the application at runtime.

The simulation scenarios can be scripted with a language similar to QScript. The language allows the definition of simulation experiments, including the specification of rigid or dynamic objects, and the definition of (parametrized) animat models as assemblies of primitive building blocks. The script also includes descriptions of randomizations, that are needed to vary the simulation during the evolution in a comparable, deterministic way. Herewith, experiments can be defined and varied rapidly. New physical objects, such as specific motors and sensors, can be added with a simple plug-in mechanism, so that the NERD standard applications can be adapted to very specific robot hardware.

All experiments described in sections 7 and 8 have been realized with the NERD simulator.

D.2 Neural Network Editor

For the ICONE method, an interactive graphical network editor with appropriate functionality is mandatory. Without such an editor, it becomes very tedious and difficult to modularize networks and to keep track of constraints and network tags. The network editor implemented as part of the NERD toolkit eases the construction, modularization, analysis, comprehension and handling of neural networks in many ways and is a valuable part of the NERD toolkit.

Network Construction. When designing an evolution experiment with ICONE, not only the simulation experiment, but also the initial networks have to be prepared. This preparation involves the construction of *peripheral structures*, that are needed for the net-

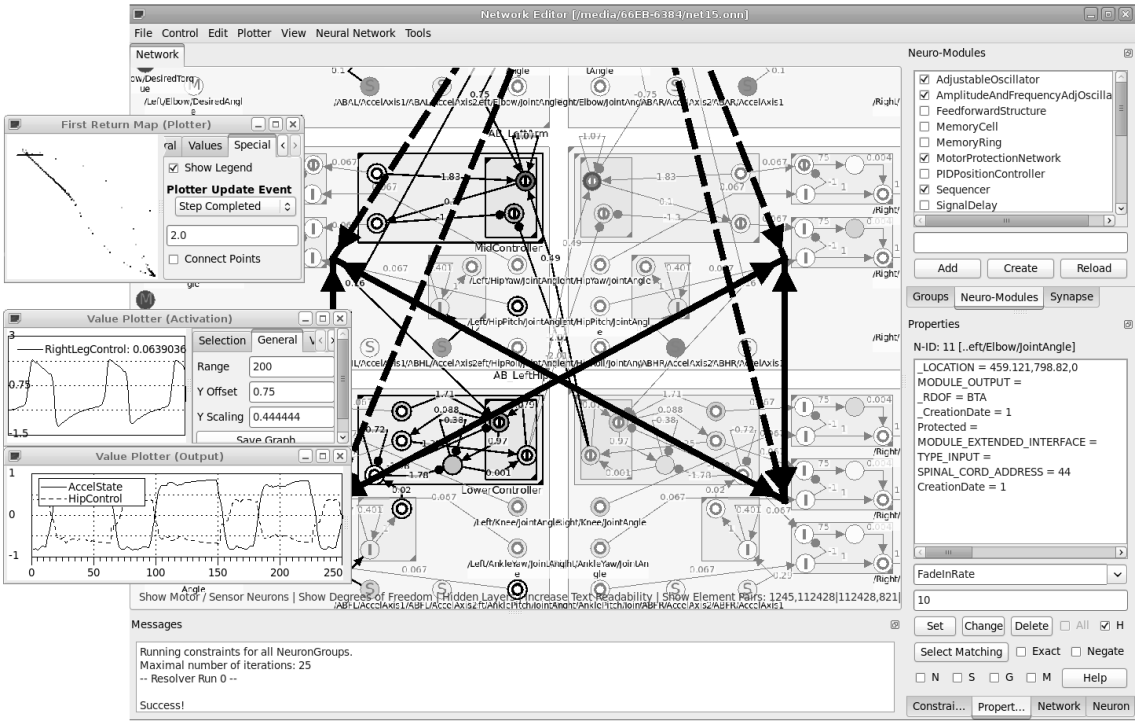


Figure D.3: Screenshot of the NERD neural network editor.

work to function, but that are not a (primary) target of the evolution experiment. But also when insights gained from experiments should be transferred to new applications, it is important to design an often complex neuro-controller by hand.

The NERD network editor provides a number of tools to construct, visualize and layout networks. Especially for large networks, where it becomes difficult to keep a clear view on the network, NERD provides handy features, like different zoom settings, bookmarks for a fast navigation between network areas, search and highlighting of network elements by selected properties (network tags, weights, names, constraints) and hiding of network elements according to freely definable layers. Networks can be constructed and aligned intuitively with mouse and keyboard while simultaneously observing the effects of network changes on the network dynamics and on the behavior of an animat in real-time.

Network Modularization. Also, the modularization of neural networks is assisted by a number of tools. Modules and groups can be defined, organized and configured to structure networks and to manage constraints on the network. The editor can visualize effects of constraints, for instance to show the defined synaptic pathways (section 6.1) or to highlight all independently evolvable parameters of the constrained network to get an impression on the remaining search space. Also – because many constraints reference network elements by their unique identifier – a number of tools help to ease the handling of such ids. Herewith,

the separation of large networks with respect to regularities (e.g. similar subnetworks of an animat with many legs) into modules becomes fast and robust.

Constraints can be defined and parametrized through comfortable user interfaces. To test the constraints, all or a selection of constraints can be executed manually. Their effects and their optional error messages can be examined to find a proper, conflict-free configuration for a given task.

Module Library. As part of the network editor, a neuro-module library (section 5.1.3) can be accessed. Thus, fully configured neuro-modules from previous experiments can be inserted to the network as easy as single neurons or synapses. Each module instance can be separately configured and constrained further to integrate an inserted module smoothly to the network. Also, it is simple with the editor to create new neuro-modules for the library. The modules can be prepared and configured with the editor like any other network structure. Then, the corresponding network structures – thoroughly constrained and tagged – simply have to be selected and defined as a named module with a single mouse click to add it as a new prototype to the module library.

Analysis of Neuro-Controllers. One goal of the experiments in the context of evolutionary robotics and neurocybernetics is to understand the organization and dynamical principles of neuro-controllers. Therefore, the NERD network editor provides many tools for the examination and analysis of neural networks. The activations and many other dynamical aspects of the neurons, synapses, weights and other parameters can be observed in real-time while the animat interacts with its simulated environment. Different types of graph plotters can be used to analyze and compare the activations or output of the neurons. Such plotters are not only valuable during the analysis phase, but also during the manual construction, to debug networks or to shape the output of neurons or modules.

With its undo mechanism, modifications of the network (including virtual pruning) can be undone so that it becomes easy to play around with a network to examine the impact of changes.

Finally, NERD also provides an extension to create plots useful for the analysis from the perspective of dynamical systems theory, such as bifurcation diagrams, first-return maps, attractor plots and iso-periodic plots. These diagrams allow an in-depth analysis of the dynamics of network structures and a systematic examination of their limitations and application ranges for certain tasks.

D.3 Evolution Framework and ICONE Implementation

General Evolution Framework. The NERD toolkit provides a flexible evolution framework that can be extended with various evolution algorithms. The main focus is clearly on neuro-evolution, but the framework also supports any other kind of evolution that requires a physical simulation. This includes not only morphology (co-)evolutions, but also the evolution of functional controllers for animats (e.g. with genetic programming or parameter optimization). The frameworks allow many different evaluation scenarios, such as local

evolutions, evaluations distributed on a computer cluster, the use of external simulators via custom interface plug-ins, and more. Because these evaluation methods are provided by the framework, a developer of a new evolution method does not have to cope with these issues and can directly use these evaluation types. Therefore, all evolution methods automatically support a distributed evaluation on multiple computers or processing cores to speed up the evolution process.

Neuro-Evolution. For the evolution of neuro-controllers, several evolution methods have been implemented. Each evolution method can reuse the graphical user interface to configure, adjust and analyze the evolution process. Figure D.4 shows some of the main windows of that user interface.

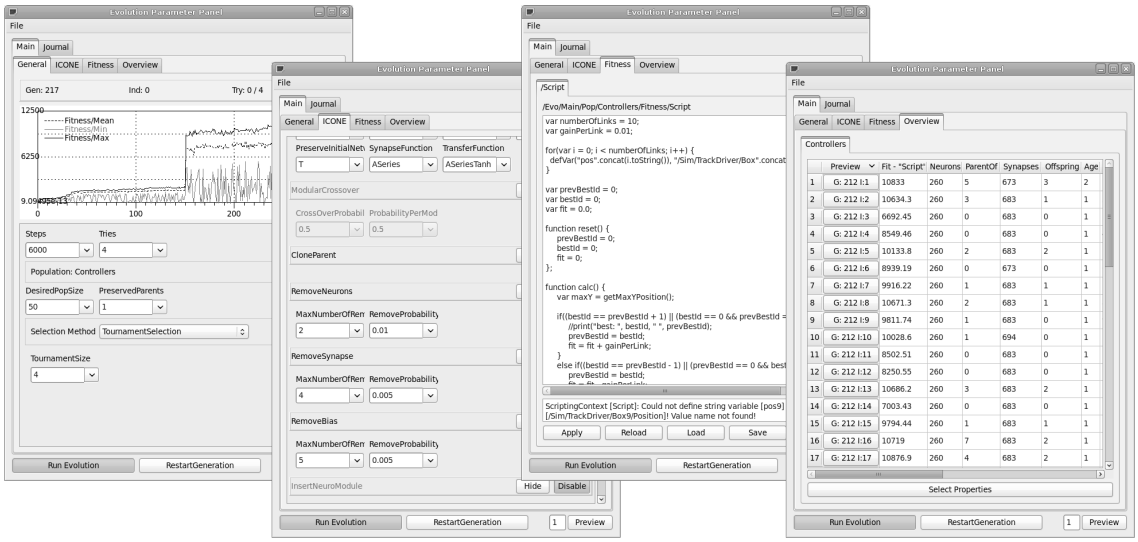


Figure D.4: Screenshots of the main evolution parameter panel, which can be used to interactively control and monitor evolution experiments.

The user interface allows, in addition to the configuration of experiments, also the observation of the evolution progress in various ways. The fitness progress can be monitored on-line, for instance to react on the fitness progress during an interactive evolution run. The fitness function can also be directly implemented in an own editor window to develop and change the fitness function (not only parameters of the function) interactively.

In a summary window, various statistics on the evolving networks are listed. This makes it easy to keep an overview on the size, performance, inheritance relations and network properties of each network. Furthermore, any network of the lastly completed generation can be re-run with a single mouse-click in the simulator under the same conditions that were used to evaluate the individual during the evaluation phase. Subsequently, the user can pick networks with significant statistical properties and examine these networks directly without interrupting the running evolution. This is especially important in combination

with evaluations on a computer cluster, because then this is the only way to examine the networks visually, which is a prerequisite for interactive evolution.

During the evolution, all the various statistics on the networks and the evolving generations are stored in files, so that the evolution process itself can be analyzed in detail after each experiment. All parameter changes of an interactive evolution run are hereby stored together with the statistics and optional user comments, so that each evolution run can be replicated in detail. This also allows the identification of useful parameter changes of an interactive experiment to derive parameter adaption rules for similar, unsupervised experiments (see section 6.5).

ICONE Implementation. The ICONE method has been implemented as part of the NERD toolkit in a way that allows a flexible extension of the method by new functional constraints, network tags and mutation operators. That way, the algorithm can be adapted to many new experiments and approaches. The neural network genomes are implemented as object structures, following the object oriented programming paradigm. This provides a fast manipulation of the network elements (e.g. by mutation operators and functional constraints) and keeps the genome open for future extensions. In addition, a scripting interface was implemented that allows the manipulation of networks via simple QScript programs, so that new constraints and mutation operators can also be developed (prototypically) without C++ programming.

The mutation chain operator is realized with an *ordered* list of mutation operator objects, that are executed during the mutation phase. New operators can be inserted to that list to define their proper execution position within the mutation chain.

Both, constraint objects and mutation operators support an error reporting mechanism, which collects all problems and warnings during their execution. These errors can then be logged or displayed in a human readable form. So, if problems are detected, e.g. when a constraint could not be resolved, then the reason for that problem can be accessed by the user to allow a fast rectification of the problem.

Extensibility. The NERD toolkit implements multiple ways to extend the evolution applications without the need to implement an own, separate evolution program. Additional constraints, mutation operators, statistics generators, experiment schedulers and even entire new evolution methods can be implemented as plug-ins and can be loaded at runtime to extend the standard NERD applications. Some of these extensions can also be realized with simple QScript programs, so that these extensions do not even require programming skills in C++.

Bibliography

David G. Amaral. The anatomical organization of the central nervous system. *Kandel E. Schwartz J, Jessell T (eds): Principles of Neural Science. New York, McGraw Hill*, 2000. 155

Peter J. Angeline, Gregory M. Saunders, and Jordan P. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5 (1):54–65, January 1994. 16

Martin Anthony. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 2009. 10

William Ross Ashby. *An Introduction to Cybernetics*. Chapman & Hall, New York, 1956. 1

Gasser Auda and Mohamed Kamel. Modular neural networks: a survey. *International Journal Neural Systems*, 9(2):129–151, Apr 1999. 19

Thomas Back, Frank Hoffmeister, and Hans-Paul Schwefel. A survey of evolution strategies. 1991. 12, 158

James Mark Baldwin. A new factor in evolution. *The American Naturalist*, 30:441–451, 1896. 74

Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications (3rd edition)*. Morgan Kaufmann, dpunkt.verlag, 2001. 65

Danielle Smith Bassett and Ed Bullmore. Small-world brain networks. *The neuroscientist*, 12(6):512–523, 2006. 155

Mark A. Bedau. Artificial life: organization, adaptation and complexity from the bottom up. *Trends in cognitive sciences*, 7(11):505–512, 2003. 2

Randall D. Beer. A dynamical systems perspective on agent-environment interaction. *Artificial intelligence*, 72(1-2):173–215, 1995. 3, 10

Randall D. Beer. Dynamical approaches to cognitive science. *Trends in Cognitive Sciences*, 4(3), March 2000. 3, 10

Randall D. Beer. Parameter space structure of continuous-time recurrent neural networks. 2005. 11

Richard K. Belew, John McInerney, and Nicol N. Schraudolph. Evolving networks: Using the genetic algorithm with connectionist learning. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, pages 511–547. Addison-Wesley, Redwood City, CA, 1992. 16

Jan Bender. Impulse-based dynamic simulation in linear time. *Computer Animation and Virtual Worlds*, 18(4-5):225–233, 2007. ISSN 1546-4261. doi: http://dx.doi.org/10.1002/cav.v18:4/5. 194

Peter Bentley and Sanjeev Kumar. Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 35–43, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann. ISBN 1-55860-611-4. 16

Josh C. Bongard. *Incremental Approaches to the Combined Evolution of a Robot's Body and Brain*. PhD thesis, University of Zürich, 2003. 3, 16, 17

Josh C. Bongard. The utility of evolving simulated robot morphology increases with task complexity for object manipulation. *Artificial Life*, 16(3):201–223, 2010. ISSN 1064-5462. 194

Josh C. Bongard and Hod Lipson. Automating genetic network inference with minimal physical experimentation using coevolution. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 333–345. Springer, 2004a. 97

Josh C. Bongard and Hod Lipson. Once more unto the breach: Co-evolving a robot and its simulator. In *Artificial life IX: proceedings of the Ninth International Conference on the Simulation and Synthesis of Artificial Life*, page 57. The MIT Press, 2004b. 97

Josh C. Bongard and Chandana Paul. Making evolution an offer it can't refuse: Morphology and the extradimensional bypass. *Advances in Artificial Life*, pages 401–412, 2001. 74, 194

Josh C. Bongard and Rolf Pfeifer. Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 829–836, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann. ISBN 1-55860-774-9. 16

Josh C. Bongard and Rolf Pfeifer. Evolving complete agents using artificial ontogeny. *Morpho-functional machines: The new species (designing embodied intelligence)*, pages 237–258, 2003. 16, 194

Stefan Bornhofen and Claude Lattaud. Outlines of artificial life: A brief history of evolutionary individual based models. In *Artificial Evolution*, pages 226–237. Springer, 2006. 3

Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, Cambridge, 1984. 15

Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M. Bower, Markus Diesmann, Abigail Morrison, Philip H. Goodman, Frederick C. Harris Jr., et al. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of computational neuroscience*, 23(3):349–398, 2007. 9

Ed Bullmore and Olaf Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(3):186–198, 2009. 46

Stefano Cagnoni, Riccardo Poli, Yun Li, George D. Smith, David Corne, Martin J. Oates, Emma Hart, Pier Luca Lanzi, Egbert J. W. Boers, Ben Paechter, and Terence C. Fogarty, editors. *Real-World Applications of Evolutionary Computing, EvoWorkshops 2000: EvoIASP, EvoSCONDI, EvoTel, EvoSTIM, EvoROB, and EvoFlight, Edinburgh, Scotland, UK, April 17, 2000, Proceedings*, volume 1803 of *Lecture Notes in Computer Science*, 2000. Springer. ISBN 3-540-67353-9. 13

Raffaele Calabretta, Stefano Nolfi, Domenico Parisi, and Günter P. Wagner. Duplication of modules facilitates the evolution of functional specialization. *Artificial Life*, 6(1): 69–84, 2000. 19

Angelo Cangelosi, Domenico Parisi, and Stefano Nolfi. Cell division and migration in a 'genotype' for neural networks. *Network: Computation in Neural Systems.*, 5(4): 497–515, 1994. 20

Genci Capi and Kenji Doya. Evolution of neural architecture fitting environmental dynamics. *Adaptive Behavior*, 13 (1):53–66, 2005. 3

Claudio Moraga Christoph M. Friedrich. An evolutionary method to find good building-blocks for architectures of artificial neural networks. In *Proceedings of the 6th International Conference on Information Processing and Management of Uncertainty in Knowledge Based Systems (IPMU'96)*, pages 951–956, 1996. 19

Jeff Clune, Benjamin E. Beckmann, Robert T. Pennock, and Charles Ofria. Hybrid: A hybridization of indirect and direct encodings for evolutionary computation. In *In Proceedings of the 10th European Conference on Artificial Life*, Budapest, Hungary, September 2009. 17

Carlos A. Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11-12):1245–1287, 2002. 32, 39

Stephen Coombes. Waves, bumps, and patterns in neural field theories. *Bilogical Cybernetics*, 93(2):91–1008, August 2005. 53

Holk Cruse. *Neural Networks as Cybernetic Systems (3rd and revised edition) - Part I.* Digital Peer Publishing (DiPP), 2009. 3

David B. D'Ambrosio and Kenneth O. Stanley. A novel generative encoding for exploiting neural network sensor and output geometry. In *Genetic And Evolutionary Computation Conference: Proceedings of the 9 th annual conference on Genetic and evolutionary computation.* Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 2007. 19, 157

Charles Darwin. *On the Origin of Species.* John Murry, London, 1859. 3, 12

Hugo de Garis. *Genetic Programming: GenNets, Artificial Nervous Systems, Artificial Embryos".* PhD thesis, Universite Libre de Bruxelles, 1991. 16

Jean-Baptiste de Lamarck. *Histoire naturelle des animaux sans vertebres*, volume 2. Meline, Cans et Compagnie, 1819. (in french), Translated and published as "Zoological Philosophy: An Exposition with Regard to the Natural History of Animals", 1984, University of Chicago Press, Chicago/IL, USA. 74

Kalyanmoy Deb. An efficient constraint handling method for genetic algorithms. *Computer methods in applied mechanics and engineering*, 186(2-4):311–338, 2000. 32

Rina Dechter. *Constraint processing.* Morgan Kaufmann, 2003. 39

Frank Dellaert and Randall D. Beer. Co-evolving body and brian in autonomous agents using a developmental model. 1994. 194

Ulf Dieckmann. Coevolution as an autonomous learning strategy for neuromodules. In Hans J. Herrmann, D. E. Wolf, and Ernst Poppel, editors, *Workshop on Supercomputing in Brain Research: from tomography to neural networks, HLRZ, KFA Julich, Germany, November 21–23.* World Scientific Publishing Co., 1995. ISBN 981-02-2250-5. 18, 20

Stephane Doncieux and Jean-Arcady Meyer. Evolving neural networks for the control of a lenticular blimp. *Lecture notes in computer science*, pages 626–637, 2003. 16

Stephane Doncieux and Jean-Arcady Meyer. Evolving modular neural networks to solve challenging control problems. In *Proceedings of the Fourth International ICSC Symposium on engineering of intelligent systems (EIS 2004).* Citeseer, 2004a. 18, 19, 20, 75

Stephane Doncieux and Jean-Arcady Meyer. Evolution of neurocontrollers for complex systems: alternatives to the incremental approach. In *Proceedings of The International Conference on Artificial Intelligence and Applications (AIA 2004)*. Citeseer, 2004b. 18

Stephane Doncieux and Jean-Arcady Meyer. Evolving pid-like neurocontrollers for nonlinear control problems. *Control and Intelligent Systems 2005*, 33(1), 2005. 18

Stéphane Doncieux, Benoît Girard, Agnès Guillot, John Hallam, Jean-Arcady Meyer, and Jean-Baptiste Mouret, editors. *From Animals to Animats 11, 11th International Conference on Simulation of Adaptive Behavior, SAB 2010, Paris - Clos Lucé, France, August 25-28, 2010. Proceedings*, volume 6226 of *Lecture Notes in Computer Science*, 2010. Springer. ISBN 978-3-642-15192-7. URL `http://dx.doi.org/10.1007/978-3-642-15193-4`. 14

Marco Dorigo and Marco Colombetti. *Robot Shaping: An Experiment in Behavior Engineering*. MIT Press/Bradford Books, 1998. 67

Keith L. Downing. Supplementing evolutionary developmental systems with abstract models of neurogenesis. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 1, pages 990–996, London, 2007. ACM Press. ISBN 978-1-59593-697-4. 16

Keith L. Downing. Computational explorations of the baldwin effect. In *Norwegian Artificial Intelligens Symposium (NAIS)*. Tapir Akademisk Forlag, 2009. 74

Keith L. Downing. The baldwin effect in developing neural networks. In Martin Pelikan and Jürgen Branke, editors, *GECCO*, pages 555–562. ACM, 2010. ISBN 978-1-4503-0072-5. 74

Adam Dziuk and Risto Miikkulainen. Creating intelligent agents through shaping of coevolution. In *Proceedings of the Congress on Evolutionary Computation*, New Orleans, LA, 2011. IEEE. URL `http://nn.cs.utexas.edu/?adziuk:cec11`. 17

ECMA-262 Standard. Ecmascript language specification, 5th edition. ECMA International, Rue de Rhone 114, CH-1204 Geneva, December 2009. 193

Peter Eggenberger. Cell interactions as a control tool of developmental processes for evolutionary robotics. In *From Animals to Animats 4*, pages 440–448, Cambridge, MA, 1996. MIT Press. 16

Jeffrey L. Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48:71–99, 1993. 15

Dario Floreano and Joseba Urzelai. Evolution of neural controllers with adaptive synapses and compact genetic encoding. *Advances in Artificial Life*, pages 183–194, 1999. 73

Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008a. 15

Dario Floreano, Phil Husbands, and Stefano Nolfi. Evolutionary robotics. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 1423–1451. Springer, 2008b. ISBN 978-3-540-23957-4. 3, 21, 74

David B. Fogel and Lawrence J. Fogel. An introduction to evolutionary programming. *Lecture Notes in Computer Science*, 1063:21–33, 1996. ISSN 0302-9743. 12

Lawrence J. Fogel, Alvin J. Owens, and Michael John Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, 1966. 12

Bernd Fritzke. A growing neural gas network learns topologies. In *Advances in Neural Information Processing Systems 7*, pages 625–632. MIT Press, 1995. 9

Jason Gauci and Kenneth O. Stanley. Generating large-scale neural networks through discovering geometric regularities. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, page 1004. ACM, 2007. 19

Mario Giacobini, Anthony Brabazon, Stefano Cagnoni, Gianni A. Di Caro, Anikó Ekárt, Anna Esparcia-Alcázar, Muddassar Farooq, Andreas Fink, Penousal Machado, Jon McCormack, Michael O'Neill, Ferrante Neri, Mike Preuss, Franz Rothlauf, Ernesto Tarantino, and Shengxiang Yang, editors. *Applications of Evolutionary Computing, EvoWorkshops 2009: EvoCOMNET, EvoENVIRONMENT, EvoFIN, EvoGAMES, EvoHOT, EvoIASP, EvoINTERACTION, EvoMUSART, EvoNUM, EvoSTOC, EvoTRANSLOG, Tübingen, Germany, April 15-17, 2009. Proceedings*, volume 5484 of *Lecture Notes in Computer Science*, 2009. Springer. ISBN 978-3-642-01128-3. 13

C. Lee Giles and Christian W. Omlin. Pruning recurrent neural networks for improved generalization performance. *IEEE transactions on neural networks/a publication of the IEEE Neural Networks Council*, 5(5):848, 1994. 15

David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989. 12

Faustino J. Gomez. *Robust Non-Linear Control through Neuroevolution*. PhD thesis, August 1 2003. Tue, 6 Jan 104 19:10:41 GMT. 17, 67

Faustino J. Gomez and Risto Miikkulainen. Incremental evolution of complex general behavior. Technical Report AI96-248, The University of Texas at Austin, Department of Computer Sciences, June 1 1997. Tue, 7 Nov 106 21:26:08 GMT. 19, 47

Faustino J. Gomez and Jürgen Schmidhuber. Evolving modular fast-weight networks for control. In Wlodzislaw Duch, Janusz Kacprzyk, Erkki Oja, and Slawomir Zadrozny,

editors, *ICANN (2)*, volume 3697 of *Lecture Notes in Computer Science*, pages 383–389. Springer, 2005. ISBN 3-540-28755-8. 3

Faustino J. Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Efficient non-linear control through neuroevolution. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 654–662. Springer, 2006. ISBN 3-540-45375-X. 17

Faustino J. Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9:937–965, May 2008. ISSN 1533-7928 (electronic); 1532-4435 (paper). 14, 17, 19

Frédéric Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD Thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, 1994. 14, 19

Frédéric Gruau. Automatic definition of modular neural networks. *Adaptive Behaviour*, 3 (2):151–183, 1995. 16, 19

Frédéric Gruau and L. Darrell Whitley. Adding learning to the cellular development of neural networks: Evolution and the Baldwin effect. *Evolutionary Computation*, 1(3): 213–233, 1993. 73

Frédéric Gruau, Darrell Whitley, and Larry Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, Stanford University, CA, USA, 28–31 1996. MIT Press. 24

Hannes Gräuler. Hierarchical neural control of a humanoid robot arm in the sensorimotor loop. Master's thesis, Department of Computer Science, university of Osnabrück, 2011. 168

Peter J. B. Hancock. Genetic algorithms and permutation problems: A comparison of recombination operators for neural net structure specification. In *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*, pages 108–122. IEEE, 1992. 16

Peter J. B. Hancock. An empirical comparison of selection methods in evolutionary algorithms. In *To appear in the proceedings of the AISB workshop on Evolutionary Computation*, page 1, 1994. 37

Bart L.M. Happel and Jacob M.J. Murre. Design and evolution of modular neural network architectures. *Neural Networks*, 7(6):985–1004, 1994. 19

Inman Harvey, Phil Husbands, Dave Cliff, Adrian Thompson, and Nick Jakobi. Evolutionary robotics: the sussex approach. *Robotics and Autonomous Systems*, 1997. 3, 15, 21

Inman Harvey, Ezequiel Di Paolo, Rachel Wood, Matt Quinn, and Elio Tuci. Evolutionary robotics: A new scientific tool for studying cognition. *Artificial Life*, 11(1-2):79–98, 2005. 3, 15, 21

Simon Haykin. *Neural Networks and Learning Machines: A Comprehensive Foundation.* Prentice Hall International, 2008. 10

Donald O. Hebb. The organization of behavior: A neuropsychological approach. *NewYork: John Wiley & Sons. Hinton, GE (1989). Deterministic Boltzmann learning performs steepest descent in weightspace. Neural Computation*, 1:143–150, 1949. 9

Francisco Herrera, Manuel Lozano, and José L. Verdegay. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artificial Intelligence Review*, 12(4):265–319, 1998. 16

Manfred Hild, Matthias Meissner, and Michael Spranger. Humanoid Team Humboldt Team Description 2007 for RoboCup 2007. *Atlanta USA*, 2007. 77

Manfred Hild, Thorsten Siedel, Christian Benckendorff, Matthias Kubisch, and Christian Thiele. Myon: Concepts and design of a modular humanoid robot which can be reassembled during runtime. In *Proceedings of the 14th International Conference on Climbing and Walking Robots*, 2011. 168

Manfred Hild, Torsten Siedel, Christian Benckendorff, Christian Thiele, and Michael Spranger. Myon, a new humanoid. In Luc Steels and Manfred Hild, editors, *Language Grounding in Robots*, pages 25–44. Springer US, 2012. ISBN 978-1-4614-3064-3. URL `http://dx.doi.org/10.1007/978-1-4614-3064-3_2`. 168

Michael L. Hines and Ted Carnevale. The neuron simulation environment. *Neural computation*, 9(6):1179–1209, 1997. 9

Robert Hinterding, Zbigniew Michalewicz, and Agoston E. Eiben. Adaptation in evolutionary computation: A survey. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 65–69. IEEE, 1997. 158

John H. Holland. Genetic algorithms. *Scientific American*, 267(1):44–50, July 1992. 12

Jeffrey Horn. *The Nature of Niching: Genetic Algorithms and the Evolution of Optimal, Cooperative Populations.* PhD thesis, University of Illinois, 1997. 17

Gregory S. Hornby and Jordan B. Pollack. Body-brain co-evolution using L-systems as a generative encoding. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 868–875, San Francisco, California, USA, 7-11 2001. Morgan Kaufmann. ISBN 1-55860-774-9. 16, 74

Gregory S. Hornby and Jordan B. Pollack. Creating high-level components with a generative representation for body-brain evolution. *Artificial Life*, 8(3):223–246, 2002. 74

Gregory S. Hornby, Hod Lipson, and Jordan B. Pollack. Generative representations for the automated design of modular physical robots. *IEEE transactions on Robotics and Automation*, 19(4):703–719, 2003. 3, 14

Jonathan C. Horton and Daniel L. Adams. The cortical column: a structure without a function. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 360 (1456):837, 2005. 19

Martin Hülse. *Multifunktionalität rekurrenter neuronaler Netze: Synthese und Analyse nichtlinearer Kontrolle autonomer Roboter*. PhD thesis, University of Osnabrück, Germany, 2006. 67

Martin Hülse, Steffen Wischmann, and Frank Pasemann. Structure and function of evolved neuro-controllers for autonomous robots. *Connection Science*, 16(4):249–266, December 2004. 3, 15, 18, 20, 31

Martin Hülse, Steffen Wischmann, and Frank Pasemann. The role of non-linearity for evolved multifunctional robot behavior. In Juan Manuel Moreno, Jordi Madrenas, and Jordi Cosp, editors, *ICES*, volume 3637 of *Lecture Notes in Computer Science*, pages 108–118. Springer, 2005. ISBN 3-540-28736-1. 15, 17

Martin Hülse, Steffen Wischmann, Poramate Manoonpong, Arndt von Twickel, and Frank Pasemann. Dynamical systems in the sensorimotor loop: On the interrelation between internal and external mechanisms of evolved robot behavior. In Max Lungarella, Fumiya Iida, Josh C. Bongard, and Rolf Pfeifer, editors, *50 Years of Artificial Intelligence*, volume 4850 of *Lecture Notes in Computer Science*, pages 186–195. Springer, 2007. ISBN 978-3-540-77295-8. 11

Benjamin Inden. Stepwise transition from direct encoding to artificial ontogeny in neuroevolution. In Fernando Almeida e Costa, Luis Mateus Rocha, Ernesto Costa, Inman Harvey, and António Coutinho, editors, *ECAL*, volume 4648 of *Lecture Notes in Computer Science*, pages 1182–1191. Springer, 2007. ISBN 978-3-540-74912-7. 16

Benjamin Inden. Neuroevolution and complexifying genetic architectures for memory and control tasks. *Theory in Biosciences*, 127(2):187–194, 2008. 16

Benjamin Inden, Yaochu Jin, Robert Haschke, and Helge Ritter. Neatfields: Evolution of Neural Fields. In *Genetic and Evolutionary Computation Conference*, pages 645–646, 2010. doi: 10.1145/1830483.1830601. 16, 20

Nick Jakobi. Evolutionary robotics and the radical envelope-of-noise hypothesis. *Adaptive behavior*, 6(2):325, 1997. 97

Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. *Advances in artificial life*, pages 704–720, 1995. 97

Kenneth A. De Jong. *Evolutionary computation - a unified approach.* MIT Press, 2006. ISBN 978-0-262-04194-2. 13, 37

Eric R. Kandel, James Schwartz, Thomas Jessell, et al. *Principles of neural science*, volume 4. McGraw-Hill New York, 2000. 1

Igor V. Karpov, Vinod K. Valsalam, and Risto Miikkulainen. Human-assisted neuroevolution through shaping, advice and examples. In *Proceedings of the 13th Annual Genetic and Evolutionary Computation Conference (GECCO 2011)*, Dublin, Ireland, July 2011. URL `http://nn.cs.utexas.edu/?ikarpov:gecco11`. 67

Yohannes Kassahun, Jakob Schwendner, Jose de Gea, Mark Edgington, and Frank Kirchner. Learning complex robot control using evolutionary behavior based systems. In Franz Rothlauf, editor, *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, QuÃ©bec, Canada, July 8-12, 2009*, pages 129–136. ACM, 2009. ISBN 978-1-60558-325-9. doi: http://doi.acm.org/10.1145/1569901.1569920. 67

Christof Koch and Tomaso Poggio. Multiplying with synapses and neurons. In *Single neuron computation*, pages 315–345. Academic Press Professional, Inc., 1992. 184

Jérôme Kodjabachian and Jean-Arcady Meyer. Evolution and development of control architectures in animats. *Robotics and Autonomous Systems*, 16(2-4):161–182, 1995. 15

Jérôme Kodjabachian and Jean-Arcady Meyer. Evolution and development of neural controllers for locomotion, gradient-following, and obstacle-avoidance in artificial insects. *Neural Networks, IEEE Transactions on*, 9(5):796–812, 1998. 3, 18, 30

Sylvain Koos, Jean-Baptiste Mouret, and Stephane Doncieux. Automatic system identification based on coevolution of models and tests. In *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*, pages 560–567. Institute of Electrical and Electronics Engineers Inc., The, 2009. 97

John R. Koza. *Genetic Programming.* MIT Press, 1992. 12, 65

John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994. ISBN 0-262-11189-6. 12, 65

John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane. Genetic programming III: darwinian invention and problem solving [book review]. *IEEE Trans. Evolutionary Computation*, 3(3):251–253, 1999. 12, 65

William B. Langdon and Riccardo Poli. *Foundations of genetic programming.* Springer Verlag, 2002. 13, 16

Christopher G. Langton. *Artificial Life*. Addison-Wesley, Redwood City, CA, 1989. 2

Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011. 19

Hod Lipson, Josh Bongard, Victor Zykov, and Evan Malone. Evolutionary robotics for legged machines: from simulation to physical reality. *Intelligent autonomous systems 9: IAS-9*, page 9, 2006. 3, 97

David J. Livingstone. *Artificial Neural Networks: Methods and Applications (Methods in Molecular Biology)*. Humana Press, 1 edition, 2008. ISBN 9781588297181. 10

Francisco López-Muñoz, Jes'us Boya, and Cecilio Alamo. Neuron theory, the cornerstone of neuroscience, on the centenary of the nobel prize award to santiago ramón y cajal. *Brain research bulletin*, 70(4-6):391–405, 2006. 1

Simon M. Lucas. Growing adaptive neural networks with graph grammars. *Proceedings of European Symposium on Artificial Neural Networks (ESANN'95)*, pages 235–240, 1995. 16

Sascha Ludwig. Performance evaluation of modular crossover for the neuro-evolution method icone. Bachelor's thesis, Universitäy of Osnabrück, Germany, 2011. 42

Max Lungarella, Giorgio Mettay, Rolf Pfeifer, and Giulio Sandiniy. Developmental robotics: a survey. *Connection Science*, 15(4):151–190, 2003. 3, 15

Samir W. Mahfoud. Niching methods for genetic algorithms. *Department of Computer Science, University of Illinois at Urbana-Champaign*, 1995. 17

Dave Cliff Maja J. Matarić. Challenges in evolving controllers for physical robots. *Robotics and autonomous systems*, 19(1):67–83, 1996. 3, 14

Thomas Martinetz and Klaus Schulten. A "Neural-Gas" network learns topologies. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Proc. International Conference on Artificial Neural Networks* (Espoo, Finland), volume I, pages 397–402, Amsterdam, Netherlands, 1991. North-Holland. 9

Warren McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5:115–133, 1943. ISSN 0092-8240. URL `http://dx.doi.org/10.1007/BF02478259`. 10.1007/BF02478259. 9

Lisa A. Meeden and Deepak Kumar. Trends in evolutionary robotics. In L. C. Jain and T. Fukuda, editors, *Soft Computing for Intelligent Robotic Systems*, pages 215–233. Physica-Verlag, New York, 1998. 21

Jean-Arcady Meyer. The animat approach: simulation of adaptive behavior in animats and robots. *in Actes de la conférence neurosciences pour l'ingénieur, F. Alexandre and J. D. Kant, Eds, Publication Loria*, 1:1–21, 1998. 2

Jean-Arcady Meyer and Agnès Guillot. Biologically inspired robots. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 1395–1422. Springer, 2008. ISBN 978-3-540-23957-4. URL `http://dx.doi.org/10.1007/978-3-540-30301-5_61`. 2

Jean-Arcady Meyer and Stewart W. Wilson, editors. *From Animals to Animats 1. Proceedings of the First International Conference on Simulation of Adaptive Behavior (SAB90)*, 1990. A Bradford Book. MIT Press. 14

Jean-Arcady Meyer, Stephane Doncieux, David, and Agnes Guillot. Evolutionary approaches to neural control of rolling, walking, swimming and flying animats or robots. *Biologically Inspired Robot Behavior Engineering*, pages 1–43, 2003. 3, 16, 18, 19, 20, 24

Jean-Arcady Meyer Meyer, Phil Husbands, and Iman Harvey. Evolutionary robotics: A survey of applications and problems. In *Evolutionary Robotics*, pages 1–21. Springer, 1998. 3, 21

Zibigniew Michalewicz and Mark Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary computation*, 4(1):1–32, 1996. 32

Risto Miikkulainen. *Neuroevolution*. Springer, New York, 2010. URL `http://nn.cs.utexas.edu/?miikkulainen:encyclopedia10-ne`. 15

Risto Miikkulainen, Bobby D. Bryant, Ryan Cornelius, Igor V. Karpov, Kenneth O. Stanley, and Chern Han Yong. Computational intelligence in games. In Gary Y. Yen and David B. Fogel, editors, *Computational Intelligence: Principles and Practice*. IEEE Computational Intelligence Society, Piscataway, NJ, 2006. URL `http://nn.cs.utexas.edu/?miikkulainen:wcci06`. 17

Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Urbana*, 51:61801, 1995. 87, 108

Melanie Mitchell. An introduction to genetic algorithms. 1996. 40

Melanie Mitchell and Stephanie Forrest. Genetic algorithms and artificial life. *Artificial Life*, 1(3):267–289, 1994. 74

David E. Moriarty. *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. PhD thesis, The University of Texas at Austin, 1, 1997. 14

David E. Moriarty and Risto Miikkulainen. Efficient reinforcement learning through symbiotic evolution. Technical Report AI94-224, The University of Texas at Austin, Department of Computer Sciences, September 1 1994. Thu, 27 Jul 106 16:45:30 GMT. 17

Jean-Baptiste Mouret and Stephane Doncieux. Incremental Evolution of Animats' Behaviors as a Multi-objective Optimization. In *Proceedings of the 10th international*

conference on Simulation of Adaptive Behavior: From Animals to Animats, pages 210–219. Springer-Verlag, 2008. 17, 18

Jean-Baptiste Mouret and Stephane Doncieux. Overcoming the bootstrap problem in evolutionary robotics using behavioral diversity. In *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*, pages 1161–1168. Institute of Electrical and Electronics Engineers Inc., The, 2009a. 19

Jean-Baptiste Mouret and Stephane Doncieux. Using behavioral exploration objectives to solve deceptive problems in neuro-evolution. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 627–634. ACM, 2009b. 19

Mark E. J. Newman. Models of the small world. *Journal of Statistical Physics*, 101(3Â4), 2000. 155

Stefano Nolfi and Dario Floreano. *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines.* Bradford Book, 2004. 3, 15, 21

Stefano Nolfi and Domenico Parisi. Growing neural networks. Technical Report PCIA-91-15, Institute of Psychology, December 1991. 16, 18, 20

Stefano Nolfi and Domenico Parisi. Evolving Artificial Neural Networks that Develop in Time. In *Proceedings of the Third European Conference on Advances in Artificial Life*, page 367. Springer-Verlag, 1995. 16

Frank Pasemann. Neuromodules: A dynamical systems approach to brain modelling. In Hans J. Herrmann, D. E. Wolf, and Ernst Poppel, editors, *Workshop on Supercomputing in Brain Research: from tomography to neural networks*, pages 21–23. World Scientific Publishing Co., 1995. ISBN 981-02-2250-5. 19

Frank Pasemann. Repräsentation ohne Repräsentation - Überlegungen zu einer Neurodynamik modularer kognitiver Systeme. In G. Rusch, S. J. Schmidt, and O. Breidbach, editors, *Interne Repräsentationen - Neue Konzepte der Hirnforschung.* Frankfurt: Suhrkamp, 1996. 19

Frank Pasemann. Structure and dynamics of recurrent neuromodules. *Theory in Biosciences*, 117:1–17, 1998. 3, 11

Frank Pasemann. Complex dynamics and the structure of small neural networks. *Network: Computation in Neural Systems*, 13(2):195–216, May 2002. ISSN 0954-898X. doi: doi:10.1088/0954-898X/13/2/303. 11

Frank Pasemann and Ulf Dieckmann. Balancing rotators with evolved neurocontrollers. 1997. 3

Frank Pasemann, Ulrich Steinmetz, and Ulf Dieckman. Evolving structure and function of neurocontrollers. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin

Yao, and Ali Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 3, pages 1973–1978, Mayflower Hotel, Washington D.C., USA, 6-9 1999. IEEE Press. ISBN 0-7803-5537-7 (Microfiche). 20

Frank Pasemann, Ulrich Steinmetz, Martin Hülse, and Bruno Lara. Robot control and the evolution of modular neurodynamics. *Theory in Biosciences*, 120(3-4):311–326, December 2001. 158

Frank Pasemann, Manfred Hild, and Keyan Zahedi. So(2)-networks as neural oscillators. *Computational Methods in Neural Modeling*, 2686/2003:144–151, 2003a. 53

Frank Pasemann, Martin Hülse, and Keyan Zahedi. Evolved neurodynamics for robot control. In *European Symposium on Artificial Neural Networks*, volume 2. Citeseer, 2003b. 3, 81

Frank Pasemann, Christian W. Rempis, and Arndt von Twickel. Evolving Humanoid Behaviors for Language Games. In Luc Steels and Manfred Hild, editors, *Language Grounding in Robots*, pages 67–86. Springer US, 2012. ISBN 978-1-4614-3064-3. 112, 141

Chandana Paul and Josh C. Bongard. The road less travelled: Morphology in the optimization of biped robot locomotion. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 1, pages 226–232. IEEE, 2001. 74

Luca Peliti. Fitness landscapes and evolution. *Physics of Biomaterials: Fluctuations, Selfassembly and Evolution*, pages 287–308, 1996. 13

Simon Perkins and Gillian Hayes. Incremental acquisition of complex behaviour using structured evolution. In *Proc. Int. Conf on Genetic Algorithms and Neural Networks*, 1997. Also available from: http://www.dai.ed.ac.uk/students/simonpe/. 67

Robert J. Peterka. Comparison of human and humanoid robot control of upright stance. *Journal of Physiology-Paris*, 103(3–5):149–158, 2009. 80

Rolf Pfeifer. On the role of embodiment in the emergence of cognition: Grey walter's turtles and beyond. In *Proc. of the Workshop "The Legacy of Grey Walter*, 2002. 15

Rolf Pfeifer and Josh C. Bongard. *How the body shapes the way we think.* MIT Press, November 2006. ISBN-13: 978-262-16239-5. 3

Jordan B. Pollack, Hod Lipson, Sevan Ficici, Pablo Funes, Greg Hornby, and Richard A. Watson. Evolutionary techniques in physical robotics. *Evolvable Systems: from biology to hardware*, pages 175–186, 2000. 74, 97

William T. Powers. *Behavior: The control of perception.* Aldine de Gruyter, New York, 1973. 168

John G. Proakis and Dimitris K Manolakis. *Digital Signal Processing.* Prentice Hall, 2007. 75

Ingo Rechenberg. *Evolutionsstrategie '94.* Frommann-Holzboog, 1994. ISBN: 3-7728-1642-8. 12

Joseph Reisinger, Kenneth O. Stanley, and Risto Miikkulainen. Evolving reusable neural modules. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 69–81, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag. ISBN 3-540-22343-6. doi: doi:10.1007/b98645. 16, 19

Christian W. Rempis. Short-Term Memory Structures in Additive Recurrent Neural Networks. Master's thesis, University of Applied Sciences Bonn-Rhein-Sieg, Germany, 2007. 3, 81

Christian W. Rempis and Frank Pasemann. Search Space Restriction of Neuro-Evolution Through Constrained Modularization of Neural Networks. In K. Mandai, editor, *Proceedings of the 6th International Workshop on Artificial Neural Networks and Intelligent Information Processing (ANNIIP), in Conjunction with ICINCO 2010.*, pages 13–22, Madeira, Portugal, June 2010. SciTePress. v, 23

Christian W. Rempis and Frank Pasemann. An Interactively Constrained Neuro-Evolution Approach for Behavior Control of Complex Robots. In Raymond Chiong, Thomas Weise, and Zbigniew Michalewicz, editors, *Variants of Evolutionary Algorithms for Real-World Applications*, pages 305–341. Springer, 2012a. ISBN 978-3-642-23423-1. v

Christian W. Rempis and Frank Pasemann. Evolving variants of neuro-control using constraint masks. In Tom Ziemke, Christian Balkenius, and John Hallam, editors, *From Animals to Animats 12*, volume 7426 of *Lecture Notes in Computer Science*, pages 187–197. Springer Berlin-Heidelberg, 2012b. ISBN 978-3-642-33092-6. v

Christian W. Rempis, Verena Thomas, Ferry Bachmann, and Frank Pasemann. NERD - Neurodynamics and Evolutionary Robotics Development Kit. In N. Ando et al., editor, *SIMPAR 2010*, volume 6472 of *Lecture Notes in Artificial Intelligence*, pages 121–132. Springer, Heidelberg, 2010. v, 193

Sebastian Risi, Sandy D. Vanderbleek, Charles E Hughes, and Kenneth O. Stanley. How novelty search escapes the deceptive trap of learning to learn. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 153–160. ACM, 2009. 19

Raul Rojas. *Neural Networks. A Systematic Introduction.* Springer, Berlin, 1996. ISBN: 978-3450605058. 10

Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. 9

Ralf Salomon. The evolution of different neuronal control structures for autonomous agents. *Robotics and Autonomous Systems*, 22(3-4):199–213, 1997. 15

Bruno Sareni and Laurent Krähenbühl. Fitness sharing and niching methods revisited. *Evolutionary Computation, IEEE Transactions on*, 2(3):97–106, 1998. ISSN 1089-778X. 17

J. David Schaffer, Darrell Whitley, and Larry J. Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Proceedings of COGANN92*, pages 1–37, 1992. 12, 16

Mark Segal and Kurt Akeley. The OpenGL graphics system: A specification. Technical report, Silicon Graphics Computer Systems, Mountain View, CA,USA, 1993. 194

Torsten Siedel, Manfred Hild, and Mario Weidner. Concept and Design of the Modular Actuator System for the Humanoid Robot MYON. In *International Conference on Intelligent Robotics and Applications (ICIRA 2011)*, 2011. 168

Karl Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM New York, NY, USA, 1994a. 74

Karl Sims. Evolving 3d morphology and behavior by competition. In *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, MIT, Cambridge, MA, USA*, pages 28–39, 1994b. 74

Russel Smith. The open dynamics engine (ode) http://www.ode.org (last visited dec. 2011), 2001. 194

William Spears and Vic Anand. A study of crossover operators in genetic programming. *Lecture Notes in Computer Science*, 542:409–??, 1991. ISSN 0302-9743. 16

John P. Spencer and Gregor Schöner. An embodied approach to cognitive systems: A dynamic neural field theory of spatial working memory. In *Proceedings of the 28th Annual Conference of the Cognitive Science Society (CogSci 2006)Society (CogSci 2006)*, pages 2180–2185, Vancouver, Canada, 2006. URL `ftp://ftp.neuroinformatik.rub.de/pub/manuscripts/articles/SpencerSchoner2006.pdf`. 53

Olaf Sporns and Christopher J. Honey. Small worlds inside big brains. *Proceedings of the National Academy of Sciences*, 103(51):19219, 2006. 155

Kenneth O. Stanley. *Efficient evolution of neural networks through complexification.* PhD thesis, The University of Texas at Austin, 2004. 14, 15, 17, 19, 20

Kenneth O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, 2007. 16, 19

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural network through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002a. 16, 158, 163

Kenneth O. Stanley and Risto Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In William B. Langdon, Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant Honavar, Günter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund K. Burke, and Natasa Jonoska, editors, *GECCO*, pages 569–577. Morgan Kaufmann, 2002b. ISBN 1-55860-878-8. 16

Kenneth O. Stanley and Risto Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130, 2003a. 16

Kenneth O. Stanley and Risto Miikkulainen. Evolving adaptive neural networks with and without adaptive synapses. In Bart Rylander, editor, *Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 275–282, Chicago, USA, 12–16 July 2003b. 73

Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009. 16, 17, 19

Esther Thelen and Linda B. Smith. *A dynamic systems approach to the development of cognition and action*. MIT Press, Cambridge, MA, 1994. 3

Joseba Urzelai and Dario Floreano. Evolution of adaptive synapses: Robots with fast adaptive behavior in new environments. *Evolutionary Computation*, 9(4):495–524, 2001. 73, 157

Vinod K. Valsalam. *Utilizing Symmetry in Evolutionary Design*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 2010. URL `http://nn.cs.utexas.edu/?valsalam:phd10`. Technical Report AI-10-04. 19, 72

Vinod K. Valsalam and Risto Miikkulainen. Evolving symmetric and modular neural networks for distributed control. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 731–738. ACM, 2009. 19, 20, 72

David A. van Veldhuizen and Gary B. Lamont. Multiobjective evolutionary algorithms: Analyzing the state-of-the-art. *Evolutionary Computation*, 8(2):125–147, 2000. 18

Nikola A. Venkov. *Dynamics of neural field models*. PhD thesis, School of Mathematical Sciences, University of Nottingham, 2009. 53

Arndt von Twickel. *Embodied Modular Neural Control of Walking in Stick Insects – From Biological Models to Evolutionary Robotics*. PhD thesis, Universität zu Köln, 2011. 22, 168

Arndt von Twickel and Frank Pasemann. Reflex-oscillations in evolved single leg neurocontrollers for walking machines. *Natural Computing*, 6(3):311–337, 2007. 3

Arndt von Twickel, Ansgar Büschges, and Frank Pasemann. Deriving neural network controllers from neuro-biological data – implementation of a single-leg stick insect controller. *Biological Cybernetics*, Online First, 2011. doi: 10.1007/s00422-011-0422-1. 10, 81

Arndt von Twickel, Manfred Hild, Torsten Siedel, Vishal Patel, and Frank Pasemann. Neural control of a modular multi-legged walking machine: Simulation and hardware. *Robotics and Autonomous Systems*, 60(2):227–241, 2012. 168

Joanne Walker, Simon Garrett, and Myra Wilson. Evolving controllers for real robots: A survey of the literature. *Adaptive Behavior*, 11(3):179, 2003. 3

John F. Walker and James H. Oliver. A survey of artificial life and evolutionary robotics. 1997. 21

Xiao Fan Wang and Guanrong Chen. Complex networks: small-world, scale-free and beyond. *Circuits and Systems Magazine, IEEE*, 3(1):6–20, 2003. 46

Jon M. Watts. Animats: computer-simulated animals in behavioral research. *Journal of Animal Science*, 76(10):2596–2604, 1998. 2

Paul J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. 10

Darrell Whitley, Keith E. Mathias, and Patrick A. Fitzhorn. Delta coding: An iterative search strategy for genetic algorithms. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms (ICGA'91)*, pages 77–84, San Mateo, California, 1991. Morgan Kaufmann Publishers. 47, 157

Norbert Wiener. *Cybernetics: or Control and Communication in the Animal and the Machine*. MIT Press, Cambridge, MA, 1948. 1

Claudia Wilimzig and Gregor Schöner. The emergence of stimulus-response association from neural activation fields: Dynamic field theory. 2004. 157

Stewart W. Wilson. The animat path to AI. In *Proceedings of the first international conference on simulation of adaptive behavior on From animals to animats*, pages 15–21. MIT Press, 1991. 2

Steffen Wischmann and Frank Pasemann. The emergence of communication by evolving dynamical systems. *From Animals to Animats 9*, pages 777–788, 2006. 3, 10

Xin Yao. *A Review of Evolutionary Artificial Neural Networks*. Commonwealth Scientific and Industrial Research Organization., Victoria, Australia, 1992. 12

Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999. 24, 25

Keyan Zahedi and Frank Pasemann. Adaptive behavior control with self-regulating neurons. In *50 years of artificial intelligence*, pages 196–205. Springer-Verlag, 2007. 73, 157

Glossary

Bootstrapping
The first phase of an evolution experiment, in which the first promising gradients have to be identified based on random mutations. Details can be found in chapter 2.4 and 9.2.4.

Complexification
Starting with a small network, neurons and synapses are incrementally added a little at a time, in contrast to fixed-topology evolutions, where all (or most) network elements are given in advance. Complexification starts the search with fewer mutable parameters and often leads to faster, simpler solutions (see section 2.4 at page 19).

Constrained Modularization
See Modularization.

Constraint Mask
A network structured with network tags and a layer of constrained neuron-groups and neuro-modules. The functional constraints of the groups and modules, as well as the limiting network tags, together define a specific subspace of the search space, in which the search exclusively takes place. This reduces the search space and biases the search towards specific topologies. Details can be found in section 5.1.

Evolvability
Describes how well the evolution performs with respect to the success rate, the run-time performance and the quality of the evolved solutions.

Fitness Landscape
The characteristics of the fitness with respect to the provided fitness gradients for every point in the search space (see sections 2.2 and 2.4).

Focus Structures
The part of the neural network that is the focus of the experiment and therefore the unknown part of the network. This is in contrast to peripheral structures, that are only of supportive nature (see section 5.1.1.2).

Generation
In the context of evolutionary algorithms, a generation is a set of individuals that directly compete against each other in terms of performance and therefore for the chance to produce offspring.

ICONE
Abbreviation for *Interactively Constrained Neuro-Evolution*, the name for the proposed neuro-evolution method.

Individual
In the context of evolutionary algorithms, an individual is a single genome, i.e. one specific set of parameters.

Mid-Scale Networks
A class of networks with a network size of approx. 50 - 500 neurons. This class was defined as target domain for the proposed method to distinquish it from the class of small networks and from very large networks (with thousands of neurons), which are used in a very different research area (e.g. vision, statistical analysis, modelling of biological data). See also section 2.3.

Modularization
The process of preparing a neural network for an ICONE evolution experiment by structuring the network and by defining a constraint mask on that network. For details see section 5.1.

NERD Toolkit
The Neurodynamics and Evolutionary Robotics Development Toolkit is the reference implementation of the ICONE method. Details can be found in appendix D.

Network Element
In the context of this document, a network element is one of the major building blocks of the neural networks: a synapse, a neuron, a neuron-group, a neuro-module and the network instance itself.

Network Shaping
The procedure to shape and restrict the set of valid network topologies and weight distributions. With ICONE, network shaping is a cornerstone of the search space restriction measures and is realized by defining constraint masks. For details see section 5.1.

Network Tag
A string-string pair added to a network element to annotate it with additional information. Network tags are interpreted by mutation operators, constraints and other program parts to adapt their behavior accordingly. Details can be found in section 3.2.4.

Parameter Dependency
: During evolution, some parameters of network elements are dependent on other parameters of the network and therefore are fully defined by these parameters. As a result, a dependent parameter is not part of the search space any more.

Peripheral Structures
: Supportive network structures that are known in advance and that are used to prepare the frame for the focus structures of the evolution experiment. Peripheral structures are part of the constraint mask in that they contribute to shaping the search space towards a specific network topology (see section 5.1.1.2).

Phenotype and Genotype
: The genotype is the genetic representation of an individual, the phenotype the actual instantiation of an individual. Phenotypes are generated based on genotypes. In static, deterministic evolutionary methods, every genotype represents a single phenotype. However, a genome can, in principle, also produce different phenotypes, for instance when the mapping function involves stochastic elements or when an adaption of the individual to its environment is supported. Also, a single phenotype can often be encoded with different genotypes. Therefore, individuals with the same genotype may differ in their phenotype from each other and vice versa.

Search Space
: The set of all valid parameter settings of a search. The search space is limited by the number of variable parameters and by the number of different values of a parameter (e.g. limited by a range, or given by a set of values).

Tagging
: The process of adding network tags to a network element.

Index