Technical Report

Enhancing the Neuro-Controller Design Process for the Myon Humanoid Robot

Christian Rempis, Manfred Hild, Frank Pasemann

Neurocybernetics Laboratory

Institute of Cognitive Science
University of Osnabrück

Refers to the software and hardware state of May 2011. Published in 2013.

Enhancing the Neuro-Controller Design Process for the Myon Humanoid Robot

Christian W. Rempis, Manfred Hild, Frank Pasemann

Institute of Cognitive Science, Osnabrück University, Germany[**]
Neurorobotics Research Laboratory, Humboldt University of Berlin, Germany
{`christian.rempis | frank.pasemann`}`@uni-osnabrueck.de`
`hild@informatik.hu-berlin.de`,
`http://ikw.uni-osnabrueck.de/~neurokybernetik/`
`http://www.neurorobotics.eu`

Abstract. Developing neural networks for the behavior control of autonomous robots can be a time-consuming task. This is especially the case for the new generation of complex robots with many sensors and motors – such as humanoid robots –, for which the networks with hundreds of neurons can become comparably large. Looking at the corresponding controller design workflow, a number of properties can be identified that slow down the development process: (1) The difficulty to create, handle and comprehend the large neuro-controllers, (2) the intricate debugging of neuro-controllers on the hardware, (3) delays caused by frequent time-consuming uploads of controllers to the hardware, (4) potential damaging of the robot and (5) the overall maintenance effort. This article proposes several measures to improve this workflow with respect to the mentioned problems. Some proposed improvements are realized by using sophisticated evolutionary robotics development software and suitable graphical network design tools. Such software, here in particular the *Neurodynamics and Evolutionary Robotics Development Toolkit* (NERD), significantly improves the network design process, specifically by allowing the development partially in simulation, by allowing a visual design of controllers with graphical network editors and by using suited neuro-evolution algorithms. Other improvements are based on proper neuro-modules that can be used to increase the usability of existing controllers. Bundled together, the proposed measures lead to a faster development of neuro-controllers. The proposed methods are demonstrated exemplarily with the Myon humanoid robot, but they can be applied also to other robots with similar properties and thus can help to improve the workflow for the neuro-controller design on such robot hardware.

Keywords: evolutionary robotics, humanoid robots, recurrent neural networks

[**] Technical Report of the Neurocybernetics Group,
Institute of Cognitive Science, Osnabrück University, Germany.
May 2011, published April 2013.

1 Introduction

In current robotics research, an increasing number of humanoid robotics platforms have been developed [2, 6, 11, 14, 18, 20, 23, 28]. One of the recent robots is the Myon robot [11], built as part of the EU project ALEAR. The robot is designed especially for experiments in the context of language games, in which the evolution and development of languages between interacting agents is examined. Therefore, the robot has been designed to be similar to a human eight-year-old child concerning its body proportions, size and weight. Furthermore, the robot is designed as a modular machine being a collection of fully autonomous, communicating body parts. Building such a machine is a challenging task, but – as with most other humanoid platforms – making the robot actually do something, thus programming the behaviors of the robot, is a great challenge of its own. Many programming paradigms for behavioral robotics have been proposed [1, 37]. One approach – the control with recurrent neural networks – sticks out because of its exceptionally suited features for this robot:

The first reason to use neural networks is the modular robot design. Parallel, distributed processing is native to neural networks, so distributing and running such a control network on the more than 20 independent, communicating processor boards of the humanoid is quite simple. Through the communication bus between the processor boards, sub-networks on different boards can still be connected via synapses, which allows large, body-spanning networks without a central control. The Myon architecture – used with a properly designed neural controller – even supports the safe removal of body parts at runtime, leaving both, the removed body part and the remaining robot, intact and functional, each running with its remaining part of the control network.

A second reason for neural networks is its qualitative similarity to biological nervous systems in animals. Controlling a robot with neural networks may lead to insights into the paradigms and principles of animal control. And also the other way round, knowledge from the control of animals can be used as inspiration for the control of the robot [40]. This is especially interesting when looking at the behaviors from the perspective of dynamical systems theory [4, 15, 29, 30], or from an "embodied cognition" point of view, where one understands behavior as a result of the dynamical interaction between an agent (animal or robot), its environment, and its internal (neural) states [31]. The identification of basic principles of (neural) behavior control in this way is presumably much easier than doing animal experiments, because an artificial neuro-controller can be examined much more detailed.

A final good reason to use recurrent neural networks for robot control is the availability of computational optimization algorithms, like learning rules and evolutionary algorithms [5, 25, 33], to create effective behavior control for robots. Recurrent neural networks have been shown to generate astonishingly robust, sensor-driven behaviors for complex robots [16, 22, 41].

The development of neural controller networks (in the following called *neuro-controllers*) is not trivial, especially for a robot with many sensors and actuators, like the Myon robot. As mentioned, in the research community, many neuro-controllers are generated using evolutionary algorithms [8, 26]. This, however, usually only applies well for networks with fixed topologies or with relatively small target networks. This is due to the large search space that can quickly increase beyond the manageable. An alternative is the manual design of such networks. A problem here is that network structures can become quite complex, and many (also positive) effects that are possible with neural networks are often not intuitive and, therefore, are often neglected. A third approach is a hybrid usage of both previous approaches: Manual network design assisted by evolutionary algorithms. This approach has the advantage that controllers can be defined based on knowledge and reasoning as far as possible. Optimization algorithms then can complete and extend the networks, automating the otherwise manual, time-consuming and error-prone search for the correct network parameters. Using artificial evolution on pre-designed networks also reduces the search space, increases the success rate of the evolutionary search and allows to bias the search towards a desired control strategy [35]. This means that evolution does not simply find *any* solution (often the most simple one), but instead finds more specific solutions previously outlined by the network designer [32]. This can be used to confirm given, more elaborate control approaches or to get clues whether a certain strategy is applicable or not for generating a desired behavior.

This article starts with a description of the typical workflows when using manual design or a combination of manual design and evolutionary algorithms to create neuro-controllers for non-trivial robots, shown exemplarily for the Myon humanoid robot. When working with the robot hardware during the network design, many problems can arise, that slow down the controller design and even can damage the hardware. This is especially true for networks generated by evolutionary algorithms, because such networks can lead to behavior which may be harmful to both the robot and persons working with the robot. Also, debugging of faulty neuro-controllers, optimization of parameters on the hardware, and understanding the functionality of the often quite large networks are commonly underrated problems. These and further problems are addressed in the next sections and, as far as they have been implemented for the Myon robot, solutions are proposed. Some of these proposed improvements are network structures that can be used in neuro-controllers as building blocks to enhance the usability of the robot. Other enhancements are realized with extensions of the robot and the network design software. And further improvements are achieved by using sophisticated design software, such as the so-called NERD Toolkit [36], a software framework developed at the Osnabrück University for simulation, neuro-evolution and neuro-control, or hybrid architectures, as described in [38]. One statement of this article is that the use of software collections like the NERD Toolkit substantially improve the network design process for complex robots like the Myon robot. The basic NERD Toolkit, therefore, is described briefly in section 3, whereas the particular software features implementing the usability improvements are described in the subsequent sections.

Although the proposed methods are shown exemplarily for the Myon humanoid robot, most of them can be transferred to other similar systems. These measures, therefore, can improve the general workflow of neural behavior design on physical robots.

2 Neuro-Controller Design Scenarios

Developing neuro-controllers is usually a time-consuming task. If networks are designed manually, all neurons and synapses have to be assembled by the designer to a directed graph. The structure, synaptic weights, bias terms and other parameters of the neurons have to be chosen and adapted based on reasoning, experience and often by trial-and-error. With appropriate experience, this can be sufficient, especially for well known tasks. However, if structures and weights cannot be inferred based on experience, then finding suitable network structures and the corresponding weights easily becomes very time-consuming. This can be reduced by constructing and optimizing networks (partially) assisted by evolutionary algorithms [7, 9, 10, 19, 21, 24, 26]. On the other hand, an evolution experiment then has to be designed and a proper fitness function as performance measure for the evolution has to be defined. This can be as time-consuming as manual optimization, because often complex behaviors are difficult to describe by efficient fitness functions. However, the effort put into providing a good fitness function and a well structured evolution experiment often pays off, because – once good measures are found – the evolution experiment can be used multiple times to generate differently structured neuro-controllers. And because many evolution experiments only need little supervision, this optimization does not require much manpower, so its application is – in principle – only limited by the available computational power. This is especially useful if one does not only strive for an optimal controller for the behavior, but if the aim is also to identify new neural paradigms and control strategies to learn more about neural control [32], for which multiple, different network structures are required.

In this section, two typical workflows for the neuro-controller design process are described. The first workflow describes the manual design of controllers directly on a robot hardware. The second workflow describes how networks can be designed manually or assisted by evolutionary algorithms in simulation, followed by a final transfer to the robot hardware. It should become clear that there are bottlenecks slowing down the already time-consuming design process. After that, common problems and inconveniences of both workflows, that are possible reasons for a protracted design process, are highlighted and solutions are proposed. Details on these solutions are then described in sections 4 - 9.

2.1 Typical Workflow for Manual Controller Design On Hardware

A typical workflow for the direct, manual design of neuro-controllers for a complex robot, here exemplarily shown for the Myon robot, is shown in figure 1. The neuro-controllers here are designed and uploaded with the *BrainDesigner Software* [12], the official software interface of the Myon robot implemented at the Humboldt University of Berlin. The BrainDesigner provides a graphical, module centered network editor that allows the construction of neuro-controllers using the mouse. Currently, the BrainDesigner is the only officially supported interface to the Myon robot and required for the transfer of neuro-controllers to the robot and for monitoring of the communication bus.

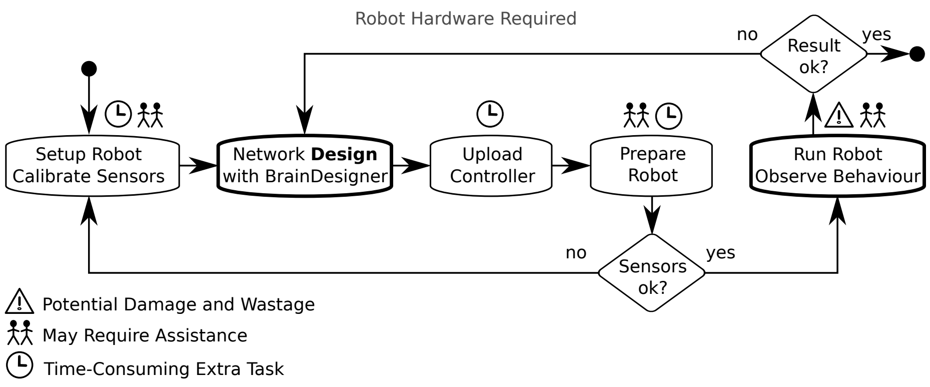

Robot Hardware Required
no
Result ok?
yes
Setup Robot Calibrate Sensors
Network **Design** with BrainDesigner
Upload Controller
Prepare Robot
Run Robot Observe Behaviour
no
Sensors ok?
yes
Potential Damage and Wastage
May Require Assistance
Time-Consuming Extra Task

Fig. 1. Workflow of the controller design on the hardware. Phases problematic with respect to time, manpower requirements and maintenance are indicated with icons. Extra tasks are assistive tasks to complement the unavoidable main tasks of network design and behavior observation (bold).

When designing neuro-controllers directly on the robot hardware, at least parts of the robot are permanently required. Every change of the neural network can only be tested with respect to the desired behavior by uploading the controller to the hardware.

The upload itself requires – depending on the number of connected body modules and the overall size of the neuro-controller – between 2 and 10 seconds. For a single upload this time requirement may seem passable, but since the upload has to be repeated after each network modification, this shortly sums up to a major bottleneck of the controller design process. This is especially problematic if a controller is faulty and the designer has to test many settings to find the cause of the problem. A second drawback is that for some experiments additional persons are required to prepare the robot for each test (e.g., to position the robot in a certain posture and to arrange the surrounding environment objects properly) or to supervise the robot during runtime (e.g., to prevent damage or to support the robot in early design phases). Thus, several persons might be needed to develop a single neuro-controller. And finally, as with every use of physical hardware, the robot is exposed to potential damage and wearout. This further increases the personnel effort and increases the costs.

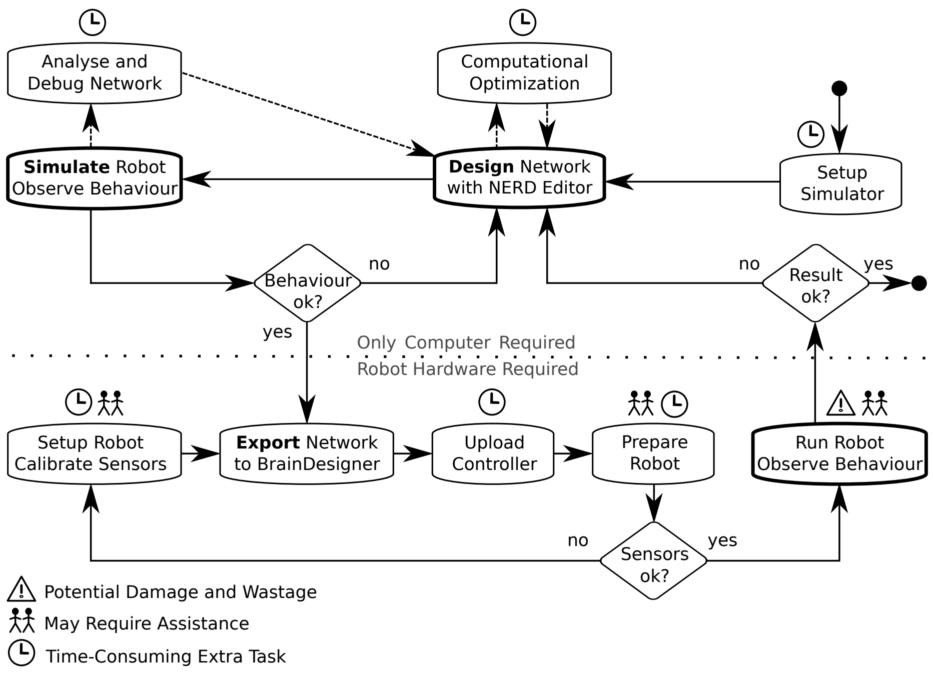
Though, developing the controller directly on the robot also has a great advantage: All developed controllers are guaranteed to work on the hardware, matching all limitations and attributes of the physical machine. A separate adoption phase after the major controller design, as is required for the workflow described in the next section, is unnecessary.

2.2 Typical Workflow for Hybrid Controller Design With Simulation, Evolution and Hardware

In difference to the previous workflow, a neuro-controller can also be designed in simulation, optionally supported by evolutionary algorithms. The neuro-controllers are in our context designed and simulated with the NERD Toolkit, implemented at the Osnabrück University in Germany (see section 3 for details). To upload a controller to the Myon robot, its network has to be exported as BrainDesigner project and can then be uploaded with the BrainDesigner application. Exporting controllers to the BrainDesigner format is fast and has the additional advantage, that networks can be extended on-the-fly with features that are usually very time-consuming to be done manually in the BrainDesigner. Such extensions are used for several of the proposed workflow improvements and will be described in the following sections. A typical workflow to design neuro-controllers with the NERD software is shown in figure 2.

In this diagram the lower part is very similar to the first approach. However, the loop between design and observation on the physical robot has been extended by a second loop in the upper part. The major controller design process now happens in this second loop, namely the controller design in the NERD editor and the testing of the controller with the simulated Myon robot. Optionally, an analysis phase can be added, that allows an in-depth examination of the neuro-controllers. Also optionally, the design process can be assisted with computational optimization algorithms, such as neuro-evolution. Neuro-evolution directly on the physical robot is very difficult, therefore, such optimization is – in practice – usually limited to simulation. Phases of computational optimization, manual design, analysis and tests on the physical machine can alternate arbitrarily.

As can be seen, the most problematic phases with respect to time, manpower requirements and maintenance in the mandatory part of the process (solid arrows) can be identified in the lower part of the diagram, where only minimal parts of the overall design takes place. However, the use of the robot cannot be avoided altogether because of the modeling inaccuracies inherent to the simulation. A final phase with adoptions to the physical robot usually is inevitable. But the modifications required for this are often limited to slight changes of synaptic weights and bias terms to account for the slightly different dynamical properties of the physical robot.



Analyse and Debug Network
Computational Optimization
Simulate Robot Observe Behaviour
Design Network with NERD Editor
Setup Simulator
Behaviour ok?
no
yes
Result ok?
no
yes
Only Computer Required
Robot Hardware Required
Setup Robot Calibrate Sensors
Export Network to BrainDesigner
Upload Controller
Prepare Robot
Run Robot Observe Behaviour
Sensors ok?
no
yes
Potential Damage and Wastage
May Require Assistance
Time-Consuming Extra Task

Fig. 2. Workflow of the controller design in simulation, optionally assisted by computational optimization algorithms. Most of the design phase happens in the upper part of the diagram, where a direct access to the physical robot is not required. Phases problematic with respect to time, manpower requirements and maintenance are indicated with icons. Extra tasks are assistive tasks to complement the unavoidable main tasks of network design and behavior observation (bold).

2.3 Problems and Inconveniences of Both Approaches

In both described workflows several problems and inconveniences can be identified that affect the development time of neuro-controllers in a direct or indirect way. This section *summarizes* these problems and gives solution approaches to improve the usability of the robot and the efficiency of the controller design process.

Coping with Large Neural Networks. Neuro-controllers for robots equipped with many motors and sensors, such as the Myon robot, frequently become quite large. The Myon robot provides about 180 sensor and 80 motor neurons that can be used all in the same network. Adding a few neurons for non-trivial processing structures and some synapses, one can easily get networks with over 2000 neurons and synapses. Such networks are difficult to handle.

A measure to cope with that is the usage of graphical design tools, such as the BrainDesigner or the NERD Toolkit. They provide many features to ease working with large networks, such as support for graphical network layouts (Sec.

4.1) and a structuring of the network into neuro-modules (Sec. 4.4). The NERD Toolkit additionally allows various constraints on the networks to reduce the number of free parameters (Sec. 4.5), provides a flexible concept of network layers and views to enhance the comprehensibility of the networks (Sec. 4.6), and contains a powerful and fast exporter for BrainDesigner projects (Sec. 7.2).

Network Delay and Reactivity. The neural networks on the Myon robot use time-discrete dynamics. All neurons of a network are updated simultaneously. Signals from the sensors, therefore, require several update steps to affect the motor neurons, because a signal has to propagate over several synapses on its path. This delay reduces the responsiveness of the robot, which becomes more severe the more synapses have to be passed to get from the sensors to the motors. To take account for this, neurons in the BrainDesigner can – with some work – be executed in a specific, asynchronous order to speed up the processing in critical subnetworks. To simplify this process, the NERD Toolkit provides *order dependent* neurons as a comfortable wrapper on this feature. This is an effective and easy to configure way to reduce the processing delay greatly (Sec. 5).

Network Optimization and Adaptation. The network structure and the network parameters can be optimized rapidly by hand when using the network editor in simulation (Sec. 6.1). The same parameters may also be optimized with the help of evolutionary algorithms (Sec. 6.2). But once a neuro-controller has to be adapted to the hardware, modifying a network becomes inefficient, because every modification has to be tested on the hardware, requiring a time-consuming network upload. As a countermeasure, Myon – also in combination with the NERD Toolkit or the BrainDesigner – supports the adjustment of synapses and bias terms at runtime directly on the robot, rendering many uploads unnecessary (Sec. 6.3).

Debugging. Finding flaws and errors in a neuro-controller can be difficult. When working with the simulator and the NERD network editor a number of analysis tools can greatly assist in identifying causes for problems (Sec. 7.1). But also on the Myon hardware activations of neurons can be observed at runtime with the BrainDesigner software. NERD minimizes the effort required to observe any neuron of a network using this feature (Sec. 7.2).

Maintenance Effort. The maintenance effort can be reduced on different scales. Firstly, the use of the simulator – where possible – relieves the robot hardware and avoids wastage and potential damage (Sec. 8.1). Secondly, manual sensor calibration often can be avoided with proper neural structures, that, for instance, automatically recalibrate the angle sensors as part of the behavior (Sec. 8.2).

Hardware Damage. Faulty adjusted neuro-controllers can easily damage the hardware and its users. The use of a simulator can help preventing such damage, because networks can be tested before they are transferred to the hardware (Sec. 9.1). Another problem in this context exists when starting a controller:

Many controllers abruptly approach their initial posture, leading to fast, strong motions, endangering people and the motors of the robot. This problem can be avoided with proper network structures to fade controllers smoothly in (Sec. 9.3). Overly strong and enduring motor activations also can damage the motors or joints. This can be avoided on the network level with motor protection sub-networks (Sec. 9.2).

Time Effort and Hardware Availability. The time effort can be greatly reduced with a combination of the previously mentioned approaches. Using a simulation and the network editor with its many usability tools makes it much faster to develop the general behavior (Sec. 8.1) and allows more designers to work simultaneously with fewer available robots. The fine-tuning of controllers on the hardware can be done rapidly with the help of the hardware extensions for network attribute adjustments at runtime (Sec. 6.3). During the entire controller development the neural networks remain clear and comprehensible (Sec. 4) and thus also speed up the development process.

3 The NERD Toolkit

The Neurodynamics and Evolutionary Robotics Development toolkit (NERD) [36] is a collection of platform independent libraries and applications for experiments in the field of evolutionary robotics, artificial life, neuro-dynamics and neuro-control. This open source project has been developed as part of the ALEAR project with a special focus on the development of larger neuro-controllers with up to approximately 5000 synapses and neurons. These are comparably large neuro-controllers in that field. Therefore, to use computational optimization in this domain, the new evolutionary method ICONE (Sec. 3.3) [34] has been developed that allows to work with such networks in a more effective way. As part of the evolutionary method, a comfortable graphical neural network editor has been implemented (Sec. 3.2). This editor can be used to construct entire networks manually, or – as its main function – to prepare network structures for evolution, to control constraints to reduce the search space and to analyze evolved neuro-controllers.

An important statement of this article is that the NERD Toolkit – with features partly specifically adapted to this class of complex robots – significantly enhances the described workflow and reduces the time effort for the controller design. The three main parts of this library that are most relevant for the neuro-controller development are described briefly in the next sections.

3.1 Simulation of the Myon Robot

The NERD simulator is a multi-purpose simulator supporting different physics engines and system calculation models to evaluate and test neuro-controllers. Objects and agents in this simulator can either be implemented flexibly with a powerful scripting language or as C++ plug-in. Simulation scenarios – usually involving one or more agents and a number of environmental objects – can be designed with the same scripting language. One feature of NERD is that all parameters of all objects in the simulation are collected in a *blackboard* repository so that all parameters can be interactively observed and modified at runtime. This is useful not only when designing a simulation scenario, but also to examine the behavior of agents closely and to define proper fitness functions for evolution.

For the Myon robot a detailed simulation model has been implemented. This model includes the physical properties of the robot, models of the multi-motor joints [12], models of the sensors, and a qualitative simulation of the vision system. For performance reasons, the outer shapes of the body parts have been simplified. Still, the simulation has to use a high update rate of 500 - 1000 Hz to remain stable and to produce sufficiently accurate results. Because of that, the simulation runs – on most computers – slower than real-time. To give an observer still a realistic impression on the behavior dynamics, a camera modus is available, that can be used to playback the observed behavior in real-time. The simulation speed itself can be increased by using only parts of the robot in simulation. Like the physical Myon robot, the simulated machine can be disassembled into its body parts. So for experiments requiring only the arms, legs or the head, only these parts can be used in simulation, which allows simulations running faster than real-time.

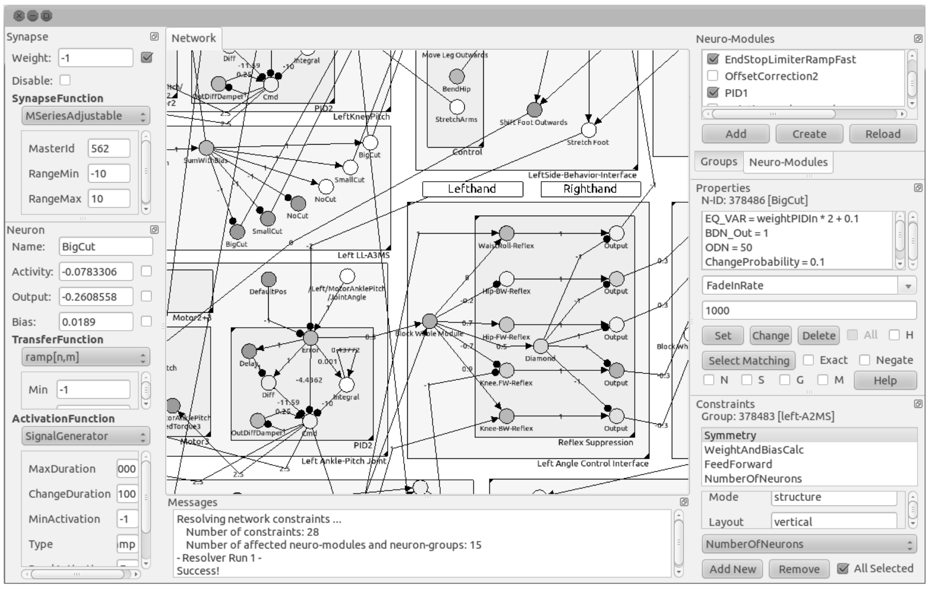
Other robots can be implemented with similar accuracy using custom plug-ins. This is important, because every robot has its own particularities that are crucial to be modeled exactly. With plug-ins, any custom model can be added to the simulator, so that no constraints are imposed by the simulator itself.

The simulator and the Myon model also support changes of the robot morphology via parameters. This enables evolution experiments with a morphology co-evolution. An example is the evolution of a basic controller for a behavior developed with a simplified Myon robot (e.g., with larger feet, stronger motors, shorter legs). Once the basic controller is found, an iterative optimization towards more and more realistic morphologies can be applied, ending up with a more complex controller working on the unmodified Myon model. Such evolutions can sometimes be easier than evolutions directly on the undoubtedly non-trivial machine. An interesting variant of this approach is the co-evolution of the morphology parameters and the neuro-controllers, leading to different search dynamics and, potentially, to hints for better robot morphologies.

3.2 Neural Network Editor

The NERD network editor (Fig. 3) is used to visualize and design neuro-controllers in close interaction with the simulation. The network editor automatically collects all sensors and motors of a simulated agent and provides these as sensor and motor neurons of the network. Neurons and synapses can be added, parametrized and modified using the mouse.

In difference to many common neural network models, NERD networks support two additional building blocks, the *neuro-modules* and the *neuron-groups*. These elements are used to logically and hierarchically structure a network.



Weight: -1 [✓]
Disable: ☐
SynapseFunction: MSeriesAdjustable
MasterId 562
RangeMin -10
RangeMax 10

Name: BigCut
Activity: -0.0783306
Output: -0.2608558
Bias: 0.0189
TransferFunction: ramp[n,m]
Min -1
ActivationFunction: SignalGenerator
MaxDuration 000
ChangeDuration 100
MinActivation -1
Type mp

[✓] EndStopLimiterRampFast
☐ OffsetCorrection2
[✓] PID1

N-ID: 378486 [BigCut]
EQ_VAR = weightPIDIn * 2 + 0.1
BDN_Out = 1
ODN = 50
ChangeProbability = 0.1
FadeInRate
1000
☐ All ☐ H
☐ Exact ☐ Negate
☐ N ☐ S ☐ G ☐ M

Group: 378483 [left-A2MS]
Symmetry
WeightAndBiasCalc
FeedForward
NumberOfNeurons
Mode structure
Layout vertical
NumberOfNeurons
[✓] All Selected

```
Resolving network constraints ...
  Number of constraints: 28
  Number of affected neuro-modules and neuron-groups: 15
- Resolver Run 1 -
Success!
```

Fig. 3. The neural network editor of the NERD Toolkit.

Neuro-modules encapsulate their member neurons and hide them from neurons outside of the module. To interact with other modules, selected neurons of that module can be marked as input or output (in the network diagrams used in this article these neurons are marked with I or O). Only these neurons can have synaptic connections to neurons outside of a module. Modules can also be added as sub-modules to other modules, resulting in a hierarchical structure. The benefit of this structuring is a clear, functional and hierarchical separation of the network, which increases comprehensibility. This leads to a strong restriction of the possible synaptic connections and thus to a reduction of the search space during evolution. *Neuron-groups*, on the other hand, are logical structure elements. Any subset of neurons and modules can be assigned to a group. The purpose of a neuron-group is to enforce certain constraints to its members, as will be described in section 4.5.

Another difference to other network models is that NERD allows additional, arbitrary properties for each network element to add further information to each element. Adding such a *network tag* is called *tagging.* Network tags can comfortably be managed with the editor, ranging from setting and changing, up to finding and visualizing such tags in the network. Network tags are important control elements, that can affect the network behavior (Sec. 5), the evolutionary search (Sec. 3.3) or the export of a network to other formats (Sec. 8.2 and 7.2).

The network editor runs in parallel with the simulation and allows the analysis and visualization of the network dynamics at runtime (Sec. 7.1). This can be combined with user interactions through the simulator or the network, such as pushing the robot around to observe the network response, or by manually overwriting neuron activities to test the reaction of the simulated machine.

An important feature of the network editor is its collection of tools that simplify the handling of large neural networks, commonly found when working with neuro-controllers for complex robots (Sec. 4.1).

3.3 Interactive Evolution Environment

Neuro-evolution in NERD is supported by different implemented evolution algorithms. The neuro-evolution method primarily used to evolve neuro-controllers for non-trivial, complex robots is the *Interactively Constrained Neuro-Evolution method* (ICONE) [33, 34]. This is due to the usually large networks and the corresponding high dimensional search space, that makes successful evolutions without massive restrictions difficult to achieve. The application of such massive search space restrictions is one of the main characteristics of the ICONE method.

ICONE is a structure evolution algorithm, that supports the evolution of both the structure and the attributes (synaptic weights, neuron bias terms, plasticity parameters) of a network. In addition to common modification operators, such as insertion and removal of neurons and synapses, changes of synaptic weights and changes of bias terms, ICONE provides a (customizable) rejection filter, a constraint resolver, a modular crossover operator and manipulation operators for modules.

Rejection Filter. This filter is a custom collection of functions that check the network for arbitrarily selectable attributes. These filter functions can reject a network, if certain characteristics are not met. A rejected network has to undergo the entire mutation chain again, until it is approved by all filter operators, or until a maximum number of mutations has been exceeded. In the latter case the network is discarded and replaced by a network that passes the filter. This ensures that networks known to be non-functional – such as networks where relevant sensors or motors are not properly connected – are not evaluated to avoid unnecessary computations.

Constraint Resolver. The most important operator of ICONE is the constraint resolver. This operator applies the constraints, which were set by the user in the network editor, and rejects all networks whose constraints cannot be resolved completely. Constraints, in difference to filters, can actively modify a neural network and, therefore, can restructure the network to meet specified criteria. Examples of constraints are described in section 4.5. The proper use of constraints greatly reduces the search space and increases the success rate of evolutions.

Modular Crossover. Crossover operators in evolutionary algorithms combine two successful solution candidates to produce a new one using parts of both candidates. This leads to more diversity during search and facilitates the mixture of different approaches. For neuro-evolution, crossover operators are difficult to realize, because exchanging arbitrarily chosen network parts usually leads to networks much less effective than their parents. This is because the network structure is often changed so much by the operation that the resulting network dynamics does not have much in common with both original networks. Modular crossover is an attempt to allow the exchange of partial solutions without producing networks that are too different from their parent networks. Instead of exchanging arbitrary parts of the network, only entire modules are exchanged. In addition, these modules have to be compatible with respect of their neural interface and an optional type tag. The network designer can determine which parts of the network may be exchanged, and can also specify the possible exchange partners. In the context of a properly constrained network this can lead to a non-destructive sub-network exchange with the mentioned advantages.

Module Manipulation Operators. ICONE is able to insert not only neurons and synapses, but also entire neuro-modules from a module library. For each evolution the network designer can select all suitable neuro-modules from that library and enable them for insertion. An advantage of this approach is, that previously identified or evolved (functional) neuro-modules do not have to be reinvented by evolution to be used in a network. Furthermore, the neuro-modules of the library usually are well prepared, so that, for instance, the function of a module is fixed. If evolution inserts such a prepared module, then the search space is only minimally extended, because most of the neurons in the neuro-module are fixed and cannot be changed by evolution. Furthermore, only the interface neurons of the new module are visible for other parts of the network, thus a module with one input, one output and an arbitrary number of internal neurons and sub-modules increases the search space only as much as a single neuron would do. Having also an operator to remove neuro-modules from a network, evolution can try to combine and connect known neuro-modules, instead of single neurons, leading to larger, functionally more complex networks.

Operators and Tagging. Network element *tagging*, as mentioned above, can influence the evolution process. Network tags can be used to protect network elements from being changed or to restrict the ranges of attributes (e.g., weight and bias). Tags can overwrite the global mutation parameters for certain network structures, for instance to enforce very small changes with a high probability in some areas and larger changes with low probability elsewhere. Tags can also give hints to the evolution operators, for instance to allow new synapses only between specified neuro-modules (*enforced synaptic pathways*), or to influence the distribution of neurons and synapses throughout the network. All these network tags further reduce the search space and bias the network towards desired network structures and hereby helps finding good solutions.

4 Coping with Large Networks

Neural networks for behavior control of the Myon robot often become quite large. Especially when networks with combined behaviors are created, as they are required for language games in the ALEAR context. Such networks can easily comprise over 2000 network elements. Working with and understanding such networks is not easy. In many applications, small neuro-controllers are often represented as $n \times n$ matrices with n being the number of neurons. The entries in the matrix then represent the synapses going from neuron A (row) to neuron B (column). This representation is sufficient for small networks, but does not work for sparsely connected networks with hundreds of neurons: The weight matrix would become too large and most entries of the matrix would be zero.

For larger networks graphical visualizations are more convenient. NERD visualizes the networks as graphs, indicating neurons as circles and synapses as edges with an arrow at its end. To increase readability, inhibitory synapses have a filled circle instead of an arrow. Non-zero bias terms are printed next to the neuron and synaptic weights next to the synapses (for an example see figure 7 on page 27). Neuro-modules are shown as rectangular boxes spanning around its member neurons and sub-modules. The entire network is drawn at the same plain area, so that every synaptic pathway can be traced from its origin to its target without changing the view. This is in contrast to the BrainDesigner software, where modules are black boxes, that can be opened to change the view to its inner parts. Because only the content of one module can be shown at a time, this visualization requires some navigation and searching to trace a synaptic pathway through a hierarchy of sub-modules.

Networks of this size, especially when visualized as a whole, can get unhandy and confusing. Therefore, NERD provides a number of features to assist the network designer when working with these large networks.

4.1 Network Layout

One problem of large networks is that it becomes difficult to identify the role of each neuron and to comprehend the effects of their interconnection. The NERD editor allows to layout all elements of a network with the mouse. This helps to structure the network so that its function becomes clearer. For many robots it makes sense to create a template network with a suitable layout as base for all neuro-controllers to be created for this robot. Such a layout can be a simple intuitive arrangement of the motor and sensor neurons according to their location on the robot. This often clarifies at first glance which part of the robot is involved in a neural structure. Also, related network structures, preferably encapsulated into named modules, should be arranged close to each other. Names can be given not only to modules, but also to individual neurons to point out their function.

4.2 Navigation and Search

A large network requires extensive navigation between the different network areas. The NERD editor supports bookmarks that can be set to any viewpoint and enables the user to switch to any stored location with a key stroke. To find specific network elements, NERD provides a powerful search function: Each neuron, neuron-group or neuro-module can be found by name, synapses can be found by their weight and every network element can be found by its unique id. Also, network elements can be found by their used neuron model functions, parameters and *network tags*. All search queries are interpreted as regular expressions, so elements can also be found knowing only parts of the desired parameter name.

4.3 View Modes

The elements of a neural network have may attributes, such as properties originating from *tagging* and network attributes (bias, weight, transfer function). Many of these attributes have to be available to the network designer in a convenient way. To draw all this information in the network simultaneously would be more confusing than helpful. Therefore, NERD supports different viewing modes. These modes can be switched on and off with the keyboard. Useful view modes, among others, are to mark the motor and sensor neurons, to identify the input and output neurons of modules, to hide all unselected elements, to hide the bias terms or synaptic weights, to show the transfer function of neurons, to hide element names, to show unconstrained network parts that can be modified by evolution, or to change the appearance of synapses. When designing a network the designer simply has to switch between the different view modes to have a fast access to all relevant information, without being distracted by too many details.

4.4 Neuro-Modules

Neuro-modules, as already described in section 3.2, are an important way to structure networks. Subdividing a large network into a hierarchy of neuro-modules according to their function, role or simply their corresponding robot body parts, makes the network structure much more comprehensible. Furthermore, the possible synaptic pathways are reduced, often also making the visualization clearer. With appropriate names on the modules such a modularized network can almost be read as conveniently as a computer program with functions.

4.5 Constraints

Constraints are a corner stone of the search space restriction of the ICONE evolution algorithm. However, constraints are not only useful during evolution, but also when manually constructing a network. The constraint resolver can be triggered manually, leading to the same constraint resolving attempt as there would be during evolution. In fact, when a network is prepared for evolution by the experimenter, all constraints should be tested manually before starting with the evolution to ensure that all constraints are resolvable without conflicts.

Resolving the constraints modifies a network until all constraints are met. Examples of such constraints are module cloning, symmetries, attribute dependencies, prevention or enforcing of mutual connections in a group, specific connectivity patterns, and more. Such constraints are also helpful doing manual network design.

The *cloning constraint*, for instance, can be used to have a single prototype module (e.g., a neural position controller) and use modules with the clone constraint wherever such a module is needed. The structure of the module, its layout and all attribute changes are automatically applied to all cloned modules when the constraint is resolved. This makes it very fast to apply the same modification to all parts of the network where the module is used.

As another example, *symmetries* can often be applied to neuro-controllers. Because of the symmetric robot body, symmetry can be assumed for most controllers where a specific function is required for both sides of the robot, e.g., grasping with the left and right arm or the simultaneous movement of both legs during squatting. Using a *symmetry constraint* on one side of the network reduces the design work to the unconstrained side only, whereas the other side is – mirror-inverted – completed automatically by the constraint. The symmetry constraint can also be configured to work only on the network structure, allowing differences at the weights and bias terms. NERD also supports special network tags to make the neuron interface more compatible with symmetries. An example is the `FLIP` tag, that mirrors the output of a neuron leading to the same behavior as reversing the corresponding motor or sensor. The exporter of NERD automatically extends the network with the necessary structures to realize this effect also on the Myon hardware when a BrainDesigner project for the upload procedure is generated.

A third constraint for manual design is the *attribute dependency constraint*. This constraint allows the definition of variables and equations on neurons and synapses, by *tagging* the elements with equation (`EQ`) and variable (`EQ_VAR`) tags. Synapses or neurons tagged with an equation calculate their weight or bias term based on the weights and bias terms of neurons and synapses that have been tagged with variable names. These variable names can be used in equations to represent the corresponding weight or bias value (such as in the equation $w = (2 * InputSynapse + MotorBias/4))$. Synapses that, for instance, are expected to have the same weight relation may be automatically calculated in that way based on the weight of a master synapse. Adjustments of such dependent weights and bias terms (manually or during evolution) only require a change of the few network elements that are *tagged* to provide variables.

To summarize, constraints not only restrict the search space for evolution greatly, but also reduce the design effort during manual construction.

4.6 Display Layers

In very large networks with hundreds of neurons the visualization can become less reactive (due to the high computational effort of drawing so many elements) and the network structures are more difficult to comprehend. To cope with this NERD provides network layers. Each network element can be *tagged* with the `Layer` tag to belong to one or more network layers. The user then can decide which combination of layers is actually shown in the editor. With this strategy, all network parts that are irrelevant for the current task can be hidden, so that it becomes easy to focus on the current goal. Layers can, for instance, be used to hide the recommended motor protection and auto-calibration networks of sections 8.2 and 9.2, so that the designer can benefit from their function without added complexity to the network visualization. Each network element can belong to an arbitrary number of layers simultaneously. This allows very specific and flexible view selections which are superior to a visibility control on the module level only.

5 Reducing Network Delays

The neurons of discrete-time neural networks usually are updated *simultaneously*, meaning that their activation is calculated based on the neuron activations of the previous execution step. Therefore, the result of a network update is by default independent of the order at which the neurons are actually updated. This implies that a sensor signal, to have an effect on a motor neuron, requires as many network updates as there are synapses to cross between the sensor and the motor neuron. These delays can become quite long and affect the reactivity of the robot. The network update rate on the Myon robot is 100 Hz, therefore, a delay between three and ten update steps can – e.g., for a position control network – lead to severe reactivity problems.

Neural networks on the Myon robot are represented and executed as a sequential byte code [13]. This byte code is not restricted to a specific neuron model, but instead is able to execute any kind of sequential machine code. Such code sequences can be arranged to a specific execution order. In case of the machine code representing a neural network, the code representing the synapses is always executed before the code representing the neuron activations. This mechanism can be used to model neurons that are executed in a specific order, hereby not using the neuron activations of the previous update step, but instead using the current, partially already updated activations of the other neurons. In network parts where the delay has to be as short as possible, that delay can be reduced by executing the neurons in a suitable order. That order has to ensure that each neuron only gets inputs from neurons that already have been updated in the current update step. With this mechanism, a sensor signal may be propagated from the sensor neuron to the motor neuron in a single update step.

When choosing the execution order of neurons great care has to be taken to avoid undesired side effects. In a feed-forward structure without parallel pathways (Fig. 4a) setting an appropriate execution order is trivial. Problems may arise as soon as parallel pathways (Fig. 4b) and recurrent connections (Fig. 4c) are present and timing becomes relevant for the function of the structure. The choice of the desired order, therefore, has to be done manually to ensure that the output behavior of a subnetwork is not changed compared to the synchronous update; the evolutionary algorithm cannot do this because of the lack of required knowledge about the function of each affected structure. In figure 4c, for instance, every other choice of the execution order leads to a different outcome and thus changes the original function of the module. Order dependencies have to be neutral with respect to the *desired* function, so an understanding of the

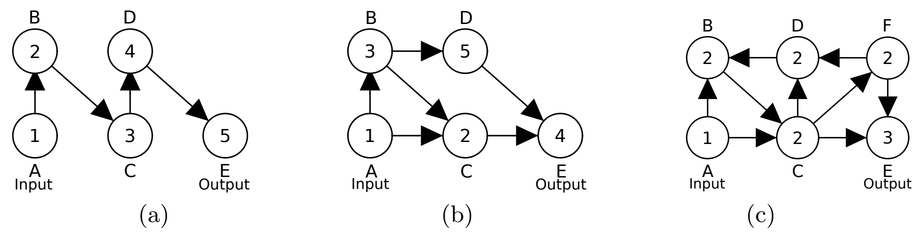


Fig. 4. Delay reduction through order dependent updates. The output of the modules remains similar, only the processing speed is increased. The examples indicate the chosen execution order (shown as numbers in the neurons) for (a) a simple feed-forward structure without parallel pathways (delay reduced from 5 to 1), (b) a feed-forward structure with parallel pathways (delay reduced from 3 to 1) and (c) a recurrent structure (delay reduced from 3 to 1).

relation between network structure and that desired function is mandatory. The additional effort, however, often pays off. As an example, the delay in a position control module of five update steps can be reduced to a single update step, which increases the reactivity of the controller significantly.

Making neurons order dependent is simple with the NERD network editor. Neurons have to be *tagged* with the `ODN` (Order Dependent Neuron) tag set to an arbitrary number. Neurons are then updated from lower to higher ODN numbers during a network update. Neurons without this tag are supposed to have an ODN number of zero. A neural network with these tags works as well in the NERD simulator as on the Myon robot. When a network is exported as BrainDesigner project, then the neuron order is transformed into appropriate model neurons with specific machine code accounting for the execution order.

6 Controller Optimization

After having obtained a basic network structure with a reasonable performance, synaptic weights and bias terms have to be fine-tuned to optimize the desired behavior. Often, also the network structure is not irrevocable and thus also part of the optimization process. In neural networks even small changes in synaptic weights or bias terms – not to mention structural changes – can have a strong effect on the resulting behavior. Hence, this optimization can be difficult and time-consuming, especially when done on the hardware without additional tools.

6.1 Optimization with Network Editor and Simulator

A fast way to optimize networks, that works for many optimization scenarios, is again to use the robot simulator and the NERD network editor. Attributes of the network can rapidly be changed manually and the outcome can be observed immediately. Different scenarios can be tested reliably and reproducible with the simulator. Combining these adaptations with the constraints described in section 4.5 can further speed up the optimization process. And by using the NERD tools to ease working with large networks (Sec. 4) also the optimization of rather complex networks is feasible.

6.2 Neuro-Evolution with the ICONE Method

Optimization can be done with computational optimization methods. This relieves the network designer from tedious try-and-error search and may lead to less intuitive but highly interesting new control paradigms. The preferred method in NERD is the use of the evolutionary method ICONE (Sec. 3.3). Because of the usually large networks and the corresponding high dimensional search space, evolution from scratch, starting without a given network structure, will likely fail. Hence, a hybrid, partially interactive approach is proposed. To bootstrap evolution and to reduce the search space, a (set of) starting network(s) has to be chosen and constrained based on reasoning, experience and domain knowledge. The more constraints such a starting network has, the smaller the search space becomes, which usually has a positive effect on the evolution outcome. Designing a starting network also has the advantage, that the network designer can decide on many aspects of the network, such as the involved sensors and motors, the overall joint control strategy (e.g., position control, force control, velocity control), the requirement of a sensor preprocessing, underlying utility modules (as shown in Sec. 9.2) or a specific control paradigm (e.g., central pattern generators, reflex loops, hierarchical approaches). This leads to a larger variety of control approaches.

The preparation of an evolutionary optimization requires additional work. Evolution requires an evaluation scenario in which the performance of solution candidates is rated. These scenarios usually involve a (partial) Myon robot and objects in its environment to interact with. A proper choice of the evaluation scenario is an important requirement to afford successful evolutions. Strongly related to the evaluation scenario is the definition of a performance measure, the so-called fitness function. The design of the fitness function often decides about success or failure of an evolution experiment. The fitness measure should provide strong gradients in the fitness space, because evolutionary algorithms belong to the gradient descent optimization algorithms [3]. Combining this requirement with a good performance description of the desired behavior can become quite difficult. Sometimes it is easier to evolve a network stepwise, starting with simple evaluation scenarios and increasing the complexity of the scenarios incrementally [8]. Neuro-controllers then do not have to cope with performance criteria that may be too difficult in the beginning.

Once defined, the evaluation scenario and the fitness function can be used multiple times to develop different approaches to solve the same behavior task, which justifies the design effort. To be able to do many evolution runs, one needs adequate computational power. Accordingly, NERD supports the use of computer clusters to reduce the run-time of evolution experiments using parallel evaluations on multiple processors or computers.

6.3 Direct Optimization on the Myon Robot

For every neuro-controller there is a phase when it has to be adapted to the physical hardware. As mentioned earlier, the simulator cannot be absolutely precise, so final changes are commonly required. This phase usually is costly in terms of time, because after every modification of synapses and bias terms the controller has to be uploaded to the robot. To avoid these frequent uploads, Myon – in combination with the BrainDesigner or NERD – provides a convenient way to adapt synapses and bias terms at runtime directly on the hardware.

For this a useful feature of the Myon robot is exploited: The robot supports hardware extensions that can link to the communication bus and participate in the data exchange. That way, such extensions can provide their own set of neurons and communicate their activations on the bus. Accordingly, network parts running on other control boards of the Myon robot can access these neurons like to any other local neuron. A useful class of hardware extensions are potentiometer boards (examples are shown in figure 5) providing a number of adjustable potentiometers to directly control the activation of corresponding local neurons. This, in fact, realizes globally accessible neurons whose activation can be adjusted at runtime. Potentiometer boards are usually used to control the function of a running neuro-controller, for instance by activating or suppressing different sub-behaviors or modes.

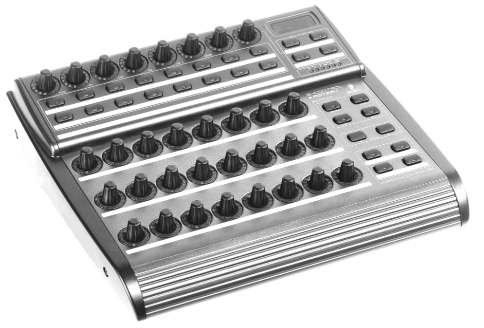


Fig. 5. Examples of a potentiometer board. Midi controller boards, such as the one shown here, can be modified so that their rulers (here potentiometers) directly change the neuron activity of associated local neurons. These neurons are accessible from all control boards of the Myon robot via communication bus and can be used to influence the network dynamics.

Potentiometer boards are also suitable to adjust bias terms and synaptic weight at runtime, given suitable extensions. To adjust a bias term a simple synapse has to be added from such a controllable neuron to the neuron whose bias term is to be optimized. At runtime the activation of the controllable neuron can be adjusted using the corresponding potentiometer until the behavior is acceptable. Then the activation setting of the controllable neuron has to be read out with the BrainDesigner software. The obtained activation now can be set as bias term to the neuron that had to be optimized.

For synaptic weights this is more difficult. NERD provides a specific adjustable synapse type, that requires additional parameters. The first parameter denotes the id of the adjustable neuron whose activation is used to calculate the weight. The other two parameters determine the range in which the synapse can be adjusted. The synaptic weight then is calculated by mapping the activation from the activation range $[-1, 1]$ of the controllable neuron to the specified adjustment range of the synapse. The weight of the synapse then is calculated anew at each network update based on the current activation of the controllable neuron.

Any synapse can be turned into an adjustable synapse with a single click in the network editor and changed back to a standard synapse. This makes it very efficient to select a set of synapses and neurons to be adjusted on the Myon robot. When networks are finally exported as BrainDesigner projects, an additional adjustable synapse model is added that behaves similar to the simulation. Once uploaded to the robot, the adjustable neurons can be modified at runtime by the potentiometer boards leading to an adaptation of the corresponding synaptic weights. In common cases, where the adjustment involves several synapses that are expected to have the same weight or a fixed weight relation, these synapses can be adjusted with a single control neuron.

Because the networks behave similar in simulation and on the hardware, the transfer of the identified synaptic weights back to the network is trivial. The activity of the controllable neuron is obtained with the BrainDesigner and set as bias term to the corresponding neuron in simulation. All dependent synapses automatically change their weight according to their specified adjustment range. The synapses now can be changed back to standard synapses and the new weights become persistent.

7 Debugging Neuro-Controllers

Neural networks, especially large ones, are not easy to comprehend and to handle. If the robot does something unexpected it may become difficult and time-consuming to find the reason for this. This is particularly difficult if the neuro-controller is running on the hardware, which makes it complicated to monitor internal neural activities. Myon, in principle, supports the observation of neural activities through the BrainDesigner software, but this requires quite some work. Nevertheless, the observation of neural activities at runtime is an essential part of any debugging process, because otherwise there is no way to verify and analyze the network dynamics.

7.1 Analysis Tools in Simulation

Debugging in simulation is much faster than on the hardware. Wherever possible the analysis should be done in simulation. The first reason is that it is usually easy to create repeatable simulation scenarios, in which the observed problem can be reliably replicated. The second reason is that all neural activities are available in the network editor and can be analyzed and visualized with many tools. Thirdly, to see effects of changes applied to probe the network, no time-consuming uploads to the hardware are necessary. Also, the activities of neurons can be manually influenced at runtime to further get an idea about the involved coherences, which is not possible on the hardware.

Plotting and Visualizations. NERD provides a number of ways to visualize and analyze neural networks. The default setting of the editor is to draw all neurons in colors, representing their current activity (compare Fig. 3 on page 12). Red tones represent positive activations, blue tones negative ones. With this the network designer can have an overall view of the network activities and can often spot problematic areas, for instance areas with unexpected activation patterns, such as unchanging, overly strong or too variable activations. To analyze the activity of a neuron over time, any set of neurons can be plotted with time series plots (compare Fig. 6, left). This makes even rapid activation developments visible and the activations of related neurons comparable. Time series plots can be created for both, the internal activation (before applying the transfer function) and the output activity (after applying the transfer function) of a neuron. Additional information can be obtained using first return map plots and phase space plots (Fig. 6, right). These plots give additional insights into the characteristics of an activation pattern.

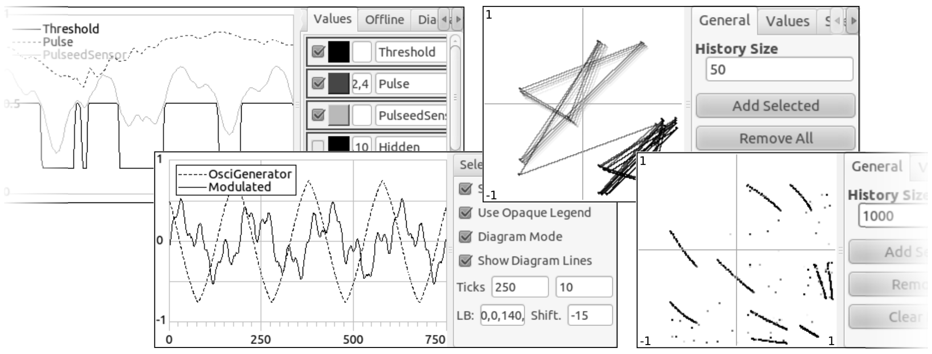


Fig. 6. A selection of plotters to analyze the neural networks at runtime: Time series plots for neuron output and activation in different configurations (left), first return maps in different modes (right).

Networks can also be analyzed numerically as isolated dynamical systems. For this a number of analysis tools from dynamical systems theory are available, such as bifurcation diagrams, iso-periodic plots, basin of attraction plots and trajectory plots [17, 27, 39].

Pruning Experiments. To identify problematic network areas and to avoid side-effects from supposedly unrelated network parts, virtual pruning experiments are helpful. Synapses in NERD can be temporarily disabled, making sure that certain network areas are fully isolated or disconnected. Because the synapses do not have to be deleted for this, all their configurations are preserved.

7.2 Activation Monitoring on the Hardware

The BrainDesigner software is able to create time series plots, but this holds only for activations available on the communication bus. To make the activity of a neuron visible on the communication bus, the user has to add and configure a new virtual sensor neuron, find a free address on the communication bus and connect a synapse from the observed neuron to the new sensor neuron. Then, after uploading the network to the Myon robot, the user can choose the sensor neuron in the BrainDesigner to be monitored and plotted as time series. Alternatively the entire bus can be recorded into a text file during the test and then be analyzed later. Both options are expensive with respect to time and effort.

NERD provides a simpler mechanism for networks created in the network editor. These networks can be exported by the editor to BrainDesigner projects including the necessary modifications required to add neurons to the communication bus and to configure the BrainDesigner to start monitoring the affected neurons with time series plots. A neuron simply has to be *tagged* with the `BDN_Out` tag. The optional value of this tag specifies a plotter id and herewith allows to assign the neuron to one of multiple plotter windows to create custom collections of time series plots. If the `BDN_Out` tag is used on a neuron not available on the communication bus, a free address on the bus is automatically chosen and the required network structures are added by the exporter. The choice of the bus address can also be overwritten manually with the `BDN_BOARD_INTERFACE` tag. This is rarely needed to optimize the distribution of the network on the controller boards. Tagging neurons in that way makes it easy to specify the observed neurons via NERD without the need of changing the network structure or to configure the BrainDesigner manually.

7.3 Probing Networks on the Hardware

If a network cannot be analysed using the simulator – for instance if the problem only occurs in combination with the hardware – then the technique described in section 6.3 combined with the time series plots presented in the previous section can be used to probe the network at runtime with a potentiometer board and to analyse its resulting activation patterns. This avoids time-consuming uploads of the network and allows detailed analysis of the active controller on the hardware.

8 Reducing Maintenance Efforts

When working with hardware, significant effort is required for maintenance and repair. Robots prototypically built for scientific experiments are in this respect much more affected than commercial robotics platforms, that already had many design iterations to increase robustness and to reduce wastage. And even there, robustness is often not at a level that is standard for most consumer electronics. The Myon robot is still in its early release phase and the durability with respect to wastage and the overall robustness is still constantly improved. However, the mechanics is still not optimized for fail-safe robustness, so improper handling can damage the machine[1]. Examples of such improper handling are strong impacts on the body, moving joints too fast into their dead stops, overheating of the motors through exhaustive use, or producing shortcuts due to careless use of metallic objects near the electronic boards. The robot also has some wear parts, that regularly have to be replaced or readjusted after a longer usage. Furthermore, some sensors have to be calibrated before usage, such as the angular position sensor of the joints. And when using battery packs as power supply, these also have to be maintained and charged. All these aspects, ranging from sensor calibration over routine maintenance to damage repair are here summed up with the term maintenance.

Because a noticeably fraction of the resources are engrossed by maintenance – especially when multiple robots are used simultaneously – and because required maintenance often interrupts network designers during their work, a reduction of this effort enhances the workflow, increases the time available for the network design and, therefore, leads to a faster controller development.

8.1 Behavior Development with the Simulator

One way to greatly reduce the maintenance effort is simply not to use the hardware. Major parts of a neuro-controller can also be developed with the simulator. This especially makes sense when new control paradigms are tested. In this case, using a simulator prevents accidental improper behavior (as described above) that is likely to occur when using new or less reliable control strategies. Once a promising neuro-controller is found and tested with the simulator, this controller gets transferred to the physical Myon robot. As a result, only a small part of the controller development, namely the optimization of a controller for the physical machine, takes place on the hardware.

In some cases a behavior controller cannot be fully developed by simulation only. This affects all controllers that are very sensitive to differences between the simulated and the physical robot. But also here, at least the basic neuro-controller structure can be developed with the NERD simulator, leaving only the sensitive network parts to be further developed on the physical machine.

An additional advantage of using the simulation is that more network designers can work in parallel with less dependency on hardware resources (except of a computer), at the same time reducing the number of required robots. This additionally diminishes the maintenance effort.

[1] Since 2011 there has been significant progress in making the Myon robot durable and reliable. In its current mature state, the robot requires much less sensor calibrations, the joints and cables are protected against damage and the overall robustness is comparable to many commercial robots.

8.2 Automatic Sensor Calibration

In some cases, certain maintenance can also be taken care of by a neuro-controller itself. Some sensors of the Myon robot, especially the angular sensors of the joints, have to be carefully calibrated before usage. Slight variations in the sensor calibration can also occur between different robots or exchanged body parts. The sensors can also be altered by improper handling during experiments, for instance when the sensors are accidentally touched. Nevertheless, many neuro-controllers require a reliable adjustment of the sensors to work properly. As a result, a frequent sensor calibration is inevitable.

The designer of a neuro-controller can relieve this situation by using suitable neuro-modules that allow the network to be used also with (slightly) inaccurate sensor calibration. One way to do this is, when possible, not to work with absolute, but instead with relative sensor values. Such controllers only need to know how sensor values change, but not what their exact output is. An example would be a vision-guided grasping behavior, where the positions of the joints are controlled – without needing the exact joint angles – based on the relative distance between the hand and the approached object.

However, many controllers do not allow such a relative sensor usage, for instance position controlled motions. But also for these networks, a suitable neuro-controller can make a manual sensor calibration superfluous. Figure 7 shows a neuro-module that can be used to calibrate the sensors at the network level. This strategy works for behaviors that have phases with known posture states, for instance when standing straight or sitting on a chair. In such situations the network can activate its calibration modules to determine and readjust suitable bias terms for the sensors, so that the known pose and the sensor output matches. When the calibration module is deactivated afterwards, the calculated bias term remains active and permanently corrects the misaligned sensor signal. As error measure other, more reliable sensors can be used, for instance the acceleration sensors. In the example of a standing behavior, the acceleration sensors can be used to align the limbs in a desired angle relative to gravity. For some joints it can also make sense to drive the joints *gently* to one of their limits and to calibrate the sensors at their dead stops, where the true angle is known (e.g., suitable for the hands and the head).

The main strategy of the sensor calibration network is to influence an angular sensor with a corrective activation, that shifts the activation of the potentially misaligned sensor to the correct level. This is done by the *Correction* neuron, whose corrective activation is summed up with the measured sensor, so that the corrected sensor signal is provided by neuron *CorrectedAngularSensor*. Instead of using the raw sensor signal, all neuro-modules in the network have to use this corrected sensor signal. All neurons of the network have linear transfer functions: The inputs and the *Correction* neuron have a range of $[-1, 1]$, all other neurons of the module a range of [0,1]. Because of this and the self-coupling of weight 1.0, the activation of the *Correction* neuron can be kept constant if the neuron is not otherwise influenced. If the *DisableCalibration* neuron is not active, then calibration takes place and the activation of the *Correction* neuron

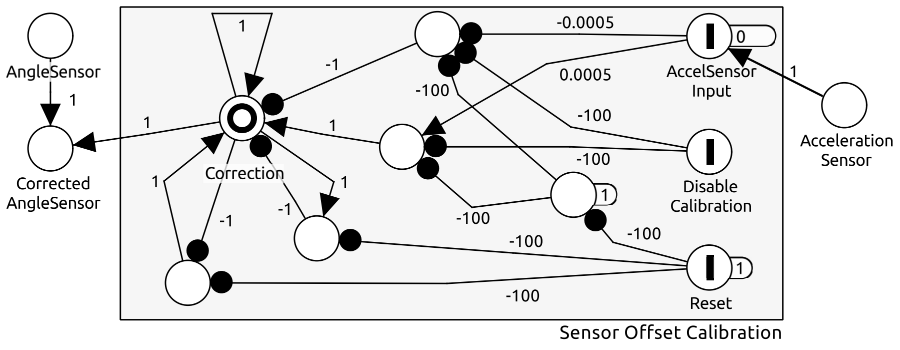


Fig. 7. A neuro-module used to determine and store a correction bias for an angular sensor. Inputs I and outputs O provide the interface of the module. The diagram does not show the position control module, that is required to move the corresponding joint to the desired default angle, hereby closing the calibration loop through the affected acceleration sensor.

is slowly changed. The direction of the change is determined by the difference between a reference input (bias term of *AccelSensorInput*) and the actual input (*AccelSensor*). For this, any sensor can be used that is directly dependent on the angular position. This can be the angular sensor itself (for instance if calibration is performed at the dead stops of a joint), or – as in the example – an acceleration sensor, assuming the arm is driven by a neural position controller. The difference between the reference and the actual input now shifts the corrective bias slowly until this difference becomes zero. The angular sensor is then calibrated. The calibration module can also be reset by inhibiting the *Reset* neuron.

9 Hardware Damage Avoidance

Section 8 already described potential risks of how the Myon robot may be damaged by improper handling and harmful control. Damaging the robot may not be the most frequent cause for delayed controller development, but one that, once it occurs, can halt an entire workgroup. And because not every workgroup has the resources and experts to do the repairs themselves, this can significantly impact the whole development process. Consequently, damage should be avoided right from the beginning. The next sections show some measures how to prevent some of the more common problems leading to damaging.

9.1 Exhaustive Testing With the Simulator

Again, the first recommendation is to develop the neuro-controllers primarily using the simulator. Potentially harmful control here is harmless until controllers are actually transferred to the physical machine. This is especially important for those design phases, where new control paradigms are applied and only marginal experience with the resulting behavior is available. Before such controllers are transferred to the physical machine and also after every major changes during the transfer, the behavior should be exhaustively tested in simulation with respect to movement speeds, problematic postures, overly strong or prolonged motor activations and too abrupt direction changes of the motors. For this the robot should be confronted with different simulation scenarios, covering both typical and potentially problematic situations. The robot behavior should be observed with the real-time camera extension of NERD to be able to judge the resulting dynamics adequately. Furthermore, NERD allows to create time series plots of variables of the simulation system, including – apart from neural activities – also all parameters of the simulated robot, such as forces, joint positions and contact points of collisions. Using time series plots for such critical variables reveals also rapidly and only briefly occurring potential problems, such as single peaks and fast oscillations of motor activities. Such problems can be missed on the physical machine because their effect on the observable behavior can be minimal, even though their negative effect on the motors can be severe.

9.2 Motor Protection Networks

Overly strong motor activations cannot be avoided in general, because in some cases very strong motor forces are required to realize a certain behavior. Such (preferably short) durations of strong motor activations are common when the joints in the legs have to lift the body or when objects are carried with the arms. The developer of a neuro-controller then has to take care that the robot preferably resides in postures minimizing the required motor activations to avoid overheating of the motors. Otherwise the motors shut off or, with some other robots, motors may even get damaged. The Myon robot provides the temperature measurements of all motors on its communication bus, so that controllers can react on imminent overheating. In a few obvious cases, however, overheating can be prevented right from the beginning, e.g., by using special neuro-modules to prevent overly strong motor activations near the dead stops. Because the dead stops of the joints are known hard limits, all approaches to reach angles beyond that limit (e.g., with a position control module) lead to a continuous, strong motor activation pressing the joint strongly into its dead stop, leading to a high current flow and increased heat.

The neuro-module shown in figure 8 is an example of a motor protection module that prevents this known cause of overheating. The network allows an increasing suppression of the motor output when approaching the upper or lower dead stop of a joint. The neurons in this network have linear transfer functions: The left input and the right output neuron of the module have a linear range of $[-1, 1]$, all other neurons in the module a linear range of [0,1]. The network can – with some modifications – also be realized with hyperbolic tangent transfer functions, if more appropriate. The overall principle of this network is simple. The desired motor activation (neuron *DesiredMotorActivation*) is split into its positive and negative activation component (neurons *Input(+)* and *Input(-)*) and subsequently merged again to be forwarded to the motor neuron. Ignoring

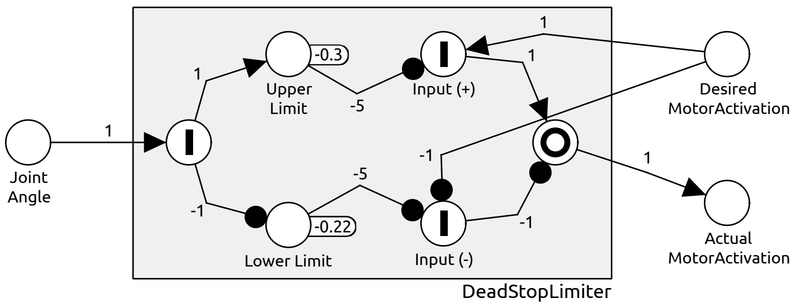


Fig. 8. Neuro-module to limit the motor activation near the dead stops of a joint.

the left part of the network, this results in a simple forwarding of the unchanged desired motor activation to the motor neuron. The additional delay can be countered with *Order Dependent Neurons* (Sec. 5), so that the entire module can be executed in a single update step. The left part of the module is used to detect the joint limits and to gradually suppress the *Input(+)* and *Input(-)* neurons when the upper or lower limit is reached. For this, the positive and negative component of the angular sensor neuron of that joint are separately compared to fixed bias terms in the *UpperLimit* and *LowerLimit* neurons. The bias term of each limit has to be chosen according to the actual dead stop angles of the joint. The negative bias terms ensure that the corresponding limit neurons only becomes active, if the joint angle is beyond the specified working interval of the joint. This leads to a suppression of the corresponding negative or positive component of the desired motor activation, effectively limiting further activation towards the problematic direction. Forces in the opposite direction are not reduced. The weights of the inhibitory synapses in the middle can be used to adjust the steepness of the suppression curve. Together with the negative bias terms in the limiter neurons any linear suppression curve can be modeled (sharp or smooth). The settings for each joint have to be specified independently according to the characteristics of the typical usage of each joint. As a variation of this module, the limits of this module may also be dynamically changed by adding neurons to influence the limiter neurons depending on the current behavior or motor conditions.

9.3 Behavior Fading

A common problem with many neuro-controllers is that these behaviors often only work smoothly in a narrow posture range, particularly when controllers use central pattern generators, direct reflex loops or fixed joint positions. Once these behaviors are running, the motor control is smooth and unproblematic. But during the start phase, i.e. when the robot is switched on, the motors have to approach the starting positions for the behavior. This often leads to undesired, strong movements, carried out without considering the current robot posture. This is problematic not only because of the strong motor activities harming the robot, but also because of the fast movements that potentially may hit or crush persons. Furthermore, when the robot is switched on, then its joints can be in any (relaxed) posture. Thus, starting the robot from inappropriate postures can lead to collisions or deadlocks of body parts.

A countermeasure is to fade behaviors smoothly in, so that the motions are slower and the electric current flow is low. A smoothly starting behavior can also be stopped in time when the initial posture of the robot turns out to be problematic.

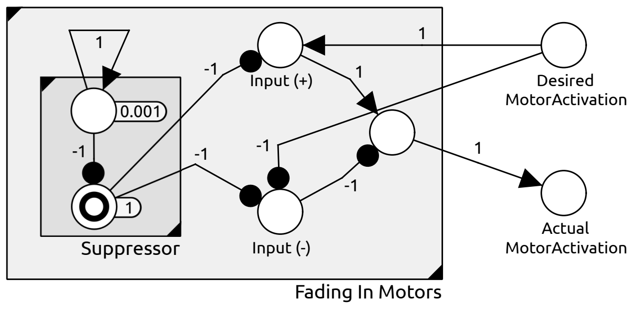


Fig. 9. Neuro-module to fade a motor slowly in when the robot is switched on.

Fading in via Neuro-Module. Figure 9 shows one way to fade in a behavior. Such modules can be added to all motors of the robot so that any behavior based on such a network automatically fades in during its starting phase. The neuro-module works similarly to the one described in section 9.2 used to protect the motors from strong activations near its dead stops. The source of suppression now comes from the *Suppressor* module. This module produces a linearly decaying activation that is used to suppress the motor activations gradually. Starting strongly, hereby suppressing the motor activations completely, the suppression fades off over time until no suppression takes place. The duration of the fading phase can be adjusted with the bias of the upper neuron in this module.

This network can also be combined with the previous network for motor protection. The *Suppression* module can be added to the motor protection network and the inhibitory synapses simply have to be connected to the *Input(+)* and *Input(-)* neurons of the motor protection module in figure 8.

Fading in via Synapse Models. NERD also provides a more convenient method to turn any neuro-controller into a smoothly starting network. For this the network simply has to be *tagged* with the `FadeInRate` tag set to a value that specifies the duration of the fading phase. When the network is exported to a BrainDesigner project, then all motor neurons automatically are equipped with a special synapse type that slowly increases its weight from 0 to 1, thus leading to a smooth and global fading of the behavior.

10 Summary

This article described the workflow for the development of neural behavior control for complex robots, such as the humanoid Myon robot. It furthermore identified measures to improve this workflow significantly. Although demonstrated exemplarily for the Myon robot in combination with the BrainDesigner and the NERD Toolkit, most of these measures can also be applied with little adaptation to other robots.

Two different approaches to neural network development are demonstrated to point out problems and inconveniences with respect to design time, manpower requirements, robot usability and maintenance. The identified main problems are:

– the handling and understanding of the often large neuro-controllers
– a decrease of the robot's reactivity due to long delay lines that come with non-trivial control structures
– the time-consuming controller optimization on the robot hardware requiring many 'costly' uploads to the hardware
– the difficult debugging and analysis of neuro-controllers, especially when working directly on the hardware
– potential risk of damage for both the robot and its users
– the requirement of additional staff when designing complex behaviors and the high number of required machines when working directly on the robot
– the time and effort required for maintenance and repair that frequently interrupts the design process when working heavily with the hardware

The first approach uses the official interface software of the Myon robot (BrainDesigner) to design networks directly on the hardware. The listed problems can be observed here very clearly.

The second approach uses the robot hardware and its physical simulation together with an alternative neuro-controller design environment (the NERD Toolkit). As described, this reduces many of the problems and facilitates additional design support, for instance by allowing the use of evolutionary algorithms to optimize or construct controllers with effective analysis tools, and also by reducing the use of the hardware. The latter leads to a reduction of potential damage, less frequent maintenance, lower demands on manpower and a better utilization of capacities, allowing more developers to work simultaneously with a fewer number robots.

To simplify the controller design on the hardware – which is to some extend required for both approaches – additional supportive measures have been implemented, like monitoring neuron activities on the hardware or adjusting synaptic weights and bias terms at runtime. With the NERD environment, the use of these otherwise quite labor intensive features becomes very efficient, because the required additional network structures are added automatically on demand. Such automatic extensions are also used to cope with other described problems, for instance to increase the robot reactivity by reducing network delays.

In addition to improvements in the network design software, some problems can be avoided and reduced with appropriate neuro-controllers. Such useful neural structures, for instance to avoid damage to the robot or to automatically recalibrate the robot's sensors, are described to be used for more robust neuro-controllers.

Bundled together the proposed workflow enhancements significantly improve the neuro-controller design process and foster a faster development of more diverse, more robust behaviors for this class of robots.

11 Resources

11.1 Myon Robot

Fully assembled Myon robots can be purchased at the Institute of Neurorobotics at the Humboldt University in Berlin. Instructions and blue-prints for the construction the robot are open source and can also be obtained there:

`http://www.neurorobotics.de/robots/myon_en.php`

11.2 BrainDesigner

The BrainDesigner ships with the Myon Robot and can also be separately obtained at the Institute of Neurorobotics at the Humboldt University in Berlin.

`http://www.neurorobotics.de/`

11.3 NERD Toolkit

The *Neurodynamics and Evolutionary Robotics Development Toolkit* (NERD Toolkit) is open source and free to use under the *GNU General Public License* (GPL). The software can be downloaded at the homepage of the Neurocybernetics workgroup of the Institute of Cognitive Science at the Osnabrück University:

`http://nerd.x-bot.org`
`http://ikw.uni-osnabrueck.de/~neurokybernetik/`

The documentation of the software can be found at

`http://nerddoc.x-bot.org`

12 Acknowledgements

This work was funded by EU-Project Number ICT 214856 (ALEAR Artificial Language Evolution on Autonomous Robots. http://www.alear.eu). Thanks go to the Neurorobotics Team of the Humboldt University of Berlin, especially

Torsten Siedel, Christian Benckendorff, Benjamin Werner and Christian Thiele. Thanks also to Ferry Bachmann and Verena Thomas for their contributions to the NERD Toolkit.

References

1. Arkin, R. *Behavior-based robotics.* The MIT Press, 1998.
2. Asfour, T., Regenstein, K., Azad, P., Schroder, J., Bierbaum, A., Vahrenkamp, N., and Dillmann, R. Armar-iii: An integrated humanoid platform for sensory-motor control. In *Humanoid Robots, 2006 6th IEEE-RAS International Conference on* (2006), IEEE, pp. 169–175.
3. Bäck, T. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, New York, 1996.
4. Beer, R. The dynamics of adaptive behavior: A research program. *Robotics and Autonomous Systems 20*, 2-4 (1997), 257–289.
5. Eiben, A. E., and Smith, J. E. *Introduction to evolutionary computation.* Natural computing series. Springer-Verlag, 2003.
6. Erbatur, K., Seven, U., Taskran, E., Koca, O., Ylmaz, M., Unel, M., Kzltas, G., Sabanovic, A., and Onat, A. Suralp: a new full-body humanoid robot platform. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on* (2009), IEEE, pp. 4949–4954.
7. Floreano, D., Dürr, P., and Mattiussi, C. Neuroevolution: from architectures to learning. *Evolutionary Intelligence 1*, 1 (2008), 47–62.
8. Floreano, D., Husbands, P., and Nolfi, S. Evolutionary robotics. In *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer, 2008, pp. 1423–1451.
9. Harvey, I., Husbands, P., Cliff, D., Thompson, A., and Jakobi, N. Evolutionary robotics: the sussex approach. *Robotics and Autonomous Systems* (1997).
10. Harvey, I., Paolo, E. D., Wood, R., Quinn, M., and Tuci, E. Evolutionary robotics: A new scientific tool for studying cognition. *Artificial Life 11*, 1-2 (2005), 79–98.
11. Hild, M., Siedel, T., Benckendorff, C., Kubisch, M., and Thiele, C. 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).
12. Hild, M., Siedel, T., Benckendorff, C., Thiele, C., and Spranger, M. Myon, a new humanoid. *Language Grounding in Robots* (2012), 25–44.
13. Hild, M., Thiele, C., and Benckendorff, C. The Distributed Architecture for Large Neural Networks (DISTAL) of the Humanoid Robot MYON. In *International Conference on Neural Computation Theory and Applications (NCTA 2011)* (2011).
14. Hirose, M., and Ogawa, K. Honda humanoid robots development. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 365*, 1850 (2007), 11–19.
15. Hülse, M., Wischmann, S., Manoonpong, P., von Twickel, A., and Pasemann, F. Dynamical systems in the sensorimotor loop: On the interrelation between internal and external mechanisms of evolved robot behavior. In *50 Years of Artificial Intelligence* (2007), M. Lungarella, F. Iida, J. C. Bongard, and R. Pfeifer, Eds., vol. 4850 of *Lecture Notes in Computer Science*, Springer, pp. 186–195.

16. Ijspeert, A. Central pattern generators for locomotion control in animals and robots: a review. *Neural Networks 21*, 4 (2008), 642–653.
17. Jost, J. *Dynamical Systems.* Springer Verlag Berlin, 2005. ISBN: 978-3-540-22908-7.
18. Kajita, S., Kaneko, K., Kaneiro, F., Harada, K., Morisawa, M., Nakaoka, S., Miura, K., Fujiwara, K., Neo, E., Hara, I., et al. Cybernetic human hrp-4c: A humanoid robot with human-like proportions. *Robotics Research* (2011), 301–314.
19. Kodjabachian, J., and Meyer, J.-A. Evolution and development of control architectures in animats. *Robotics and Autonomous Systems 16*, 2-4 (1995), 161–182.
20. Lohmeier, S., Buschmann, T., Ulbrich, H., and Pfeiffer, F. Humanoid robot lolaresearch platform for high-speedwalking. *Motion and Vibration Control* (2009), 221–230.
21. Lungarella, M., Mettay, G., Pfeifer, R., and Sandiniy, G. Developmental robotics: a survey. *Connection Science 15*, 4 (2003), 151–190.
22. Manoonpong, P., Wörgötter, F., and Pasemann, F. Biological inspiration for mechanical design and control of autonomous walking robots: Towards life-like robots. *The International Journal of Applied Biomedical Engineering (IJABME) 3*, 1 (2010), 1–12.
23. Metta, G., Natale, L., Nori, F., Sandini, G., Vernon, D., Fadiga, L., Von Hofsten, C., Rosander, K., Lopes, M., Santos-Victor, J., et al. The¡ i¿ icub¡/i¿ humanoid robot: An open-systems platform for research in cognitive development. *Neural Networks 23*, 8 (2010), 1125–1134.
24. Miikkulainen, R. *Neuroevolution.* Springer, New York, 2010.
25. Mitchell, T. M. *Machine Learning.* MxGraw-Hill, 2003.
26. Nolfi, S., and Floreano, D. *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines.* Bradford Book, 2004.
27. Nusse, H., Yorke, J., Hunt, B., and Kostelich, E. *Dynamics: numerical explorations.* Springer, 1998.
28. Ogura, Y., Aikawa, H., Shimomura, K., Morishima, A., Lim, H., and Takanishi, A. Development of a new humanoid robot wabian-2. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on* (2006), IEEE, pp. 76–81.
29. Pasemann, F. Neuromodules: A dynamical systems approach to brain modelling. In *Workshop on Supercomputing in Brain Research: from tomography to neural networks* (1995), H. J. Herrmann, D. E. Wolf, and E. Poppel, Eds., World Scientific Publishing Co., pp. 21–23.
30. Pasemann, F., Steinmetz, U., Hülse, M., and Lara, B. Robot control and the evolution of modular neurodynamics. *Theory in Biosciences 120*, 3-4 (December 2001), 311–326.
31. Pfeifer, R., and Scheier, C. *Understanding intelligence.* MIT Press, 1999.
32. Rempis, C., and Pasemann, F. Evolving variants of neuro-control using constraint masks. In *From Animals to Animats 12*, T. Ziemke, C. Balkenius, and J. Hallam, Eds., vol. 7426 of *Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, 2012, pp. 187–197.
33. Rempis, C. W. *Evolving Complex Neuro-Controllers with Interactively Constrained Neuro-Evolution.* PhD thesis, Osnabrück University, October 2012.
34. Rempis, C. W., and Pasemann, F. Search Space Restriction of Neuro-Evolution Through Constrained Modularization of Neural Networks. In *Proceedings of the 6th*

International Workshop on Artificial Neural Networks and Intelligent Information Processing (ANNIIP), in Conjunction with ICINCO 2010. (Madeira, Portugal, June 2010), K. Mandai, Ed., SciTePress, pp. 13–22.

35. Rempis, C. W., and Pasemann, F. An Interactively Constrained Neuro-Evolution Approach for Behavior Control of Complex Robots. In *Variants of Evolutionary Algorithms for Real-World Applications*, R. Chiong, T. Weise, and Z. Michalewicz, Eds. Springer, 2012, pp. 305–341.
36. Rempis, C. W., Thomas, V., Bachmann, F., and Pasemann, F. NERD - Neurodynamics and Evolutionary Robotics Development Kit. In *SIMPAR 2010* (2010), N. A. et al., Ed., vol. 6472 of *Lecture Notes in Artificial Intelligence*, Springer, Heidelberg, pp. 121–132.
37. Siciliano, B., and Khatib, O., Eds. *Springer handbook of robotics.* Springer-Verlag New York Inc, 2008.
38. Spranger, M., Thiele, C., and Hild, M. Integrating high-level cognitive systems with sensorimotor control. *Advanced Engineering Informatics 24*, 1 (2010), 76–83.
39. Strogatz, S. H. *Nonlinear Dynamics and Chaos: With Applications to Physics, Chemistry and Engineering.* Perseus Books, U.S., 2001. ISBN: 978-0738204536.
40. von Twickel, A., Büschges, A., and Pasemann, F. Deriving neural network controllers from neuro-biological data – implementation of a single-leg stick insect controller. *Biological Cybernetics Online First* (2011).
41. von Twickel, A., and Pasemann, F. Adaptive behaviour of single legs with evolved neural control. In *Dynamical principles for neuroscience and intelligent biomimetic devices* (2006), A. J. Ijspeert, J. Buchli, A. Selverston, M. Rabinovich, M. Hasler, W. Gerstner, A. Billard, H. Markram, and D. Floreano, Eds., EPFL, pp. 137–138. ISBN 978-2-8399-0134-5.