Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

TOWARDS AUTOMATED DESIGN OF COMPLEX MODULAR SYSTEMS INSPIRED BY NATURE

Doctoral Thesis

Jaroslav Vítků

Prague, September 2014

Ph.D. Programme: Electrical Engineering and Informatics
Branch of Study: Artificial Intelligence and Biocybernetics

Supervisor: Doc. Ing. Pavel Nahodil CSc.
Supervisor-Specialist: Doc. Ing. Zuzana Komínková Oplatková Ph.D.

Acknowledgement

I would like to thank to Doc. Ing. Pavel Nahodil CSc. for his leadership and help during my Diploma and PhD studies. I do not think that there are many supervisors willing to respond to student's emails even during weekends or after midnight, if necessary. I noticed that I've heard the sentence "Oh, so your supervisor is really good" from various people multiple times during my PhD study. And I know that they were right. Also, I would like to thank to my supervisor-specialist Doc. Ing. Zuzana Komínková Oplatková Ph.D. for her valuable suggestions how to make this thesis better.

Last, but not least, I want to thank to my mom and my wife for their support and patience during my study.

Abstrakt

Výzkum v těchto dnech se skládá z malých, izolovaných a často vysoce specializovaných oblastech. Jeden z hlavních cílů této práce je vytvoření platformy, která usnadní kombinování výsledků výzkumu ostatních laboratoří, jejichž výsledky mohou být prezentovány ve formě software. Toto je docíleno umožněním kombinace různých částí software dohromady do většího systému.

Autor navrhuje framework inspirovaný v modulárními neuronovými sítěmi. Tento framework reprezentuje různé subsystémy (kusy kódu) v jednotným způsobem. Spolu s jednotným komunikačním protokolem mezi těmito subsystémy (nazvanými Neurální Moduly), je umožněno libovolné kombinace těchto Modulů do větších systémů. Vnitřní komplexita každého Modulu se může pohybovat od jednoduchých matematických operací směrem ke složitým algoritmům z oblasti umělé inteligence.

Autor také představuje simulátor schopný simulace univerzálních modulárních systémů. Spolu se simulátorem tento výsledný framework umožňuje uživateli navrhovat a testovat různé komplexní modulární systémy/architektury. Důraz je zde kladen na využití tohoto frameworku pro navrhování architektur agentů v oblasti Alife.

Představený framework dále pak umožňuje použití optimalizačních algoritmů pro automatické navrhování nových architektur. Evoluční Algoritmus je zde použit pro změny vah mezi jednotlivými Neurálními Moduly tak, aby optimalizoval zadané kritérium v zadané úloze. Optimalizační algoritmus tak de-facto navrhuje nové architektury autonomních agentů speciálně pro zadaný úkol.

Přínosy tohoto navrhovaného přístupu jsou především následující. Poskytnutí platformy pro jednoduché opětovné použití stávajících algoritmů v nových systémech. Výsledné hybridní architektury kombinují tzv. přístup návrhu zdola nahoru (neuronové sítě) s opačným přístupem: shora-dolu, který je reprezentován samotnými Moduly.

Abstract

Research in these days is composed of small, isolated and often highly specialized subfields. One of main goals of the thesis is to contribute to simplification of reusing of outcomes of research of others. This is done by enabling the combination of various pieces of software together into a bigger system.

The thesis proposes a framework inspired in modular artificial neural networks. The framework represents various sub-systems (pieces of code) in an unified way. Together with unified communication protocol between these subsystems (called Neural Modules), the framework enables arbitrary combinations of these Modules into bigger, modular systems. A complexity of each Module can range from the simplest mathematical operation towards the complicated artificial intelligence algorithms.

Here, the simulator capable of general-purpose modular systems is proposed. Together with the simulator, the resulting framework enables user to design and test (by means of rapid prototyping) complex modular systems/architectures. Here, the focus is put on use of this framework for designing agent architectures in the domain of ALife.

Furthermore, the unified representation of Neural Modules in the framework enables to employ optimization algorithms for automatic designing of new architectures. Given a task and set of Neural Modules, the Evolutionary Algorithm is used to optimize connection weights in the network to de-facto design new architecture. The resulting architecture is then designed specially for the task.

The benefits of the proposed approach are mainly: Heading towards the simple reuse of current algorithms. Combining current specialized research in more complex architectures. Resulting hybrid architectures combine bottom-up design of ANNs and more classical top-down AI design. Neuro-evolutionary algorithm can be used to design entirely new hybrid architectures which fit specially for a given task.

Contents

List of Abbreviations **13**

List Of Symbols **17**

List of Figures **19**

List of Tables **23**

List of Algorithms **25**

1 Introduction **1**

1.1 Overview 1
1.2 Motivation 2
1.2.1 Brief Description of the Approach 3
1.3 Thesis Outline 4
1.4 Structure of the Text 4

2 State of the Art **7**

2.1 Artificial Neural Networks 8
2.1.1 Classification of Artificial Neural Networks 8
2.1.1.1 Common Topologies of Artificial Neural Networks 9
2.1.1.2 Classification of Neuron Models 10
2.1.1.3 General Methods of Artificial Neural Network Design . . 12
2.2 The Role of Modularity 13
2.2.1 Modular Neural Networks 13
2.2.1.1 Communication in Modular Hybrid Systems 16
2.2.2 Hierarchical Problem Decomposition 17
2.3 Evolutionary Algorithms 19

2.3.1 Memetic Evolution Algorithms 19
2.3.2 Neuro-Evolution 22
2.3.3 Memetic Neuro-Evolution 23
2.3.4 Modular Neuro-Evolution 23
2.4 Selected Agent Architectures 24
2.4.1 Selected Modular Architectures 26
2.4.2 Cognitive Architectures 27
2.5 Selected Design Methods of Modular Systems 31
2.5.1 Selected Ensemble-Like and Related Approaches 35
2.5.1.1 Group of Adaptive Models Evolution 35
2.5.1.2 Cartesian Genetic Programming 36
2.5.2 Ensemble Methods for Agent Architecture Design 37
2.6 Selected Modules for Ensemble-Based Systems 38
2.6.1 Modules for Temporal Sequence Learning 39
2.6.1.1 Time Delay Neural Networks 40
2.6.1.2 Short-Term Memory 40
2.6.1.3 Long-Term Memory 41
2.6.1.4 Categorization and Learning Module 42
2.6.1.5 Long Short-Term memory 43
2.6.2 Signal Generators 43
2.6.2.1 Echo State Network 44
2.6.2.2 Central Pattern Generator 45

3 Problem Analysis and Proposed Solution 49
3.1 Problem Analysis 50
3.1.1 Different Types of Communication 50
3.1.2 Theoretical Issues with Automatic Design of Modular Systems . . 51
3.1.3 Reusability of Sub-systems and Domain Dependency 52
3.1.4 Implementation and Practical Issues 52
3.2 Task Description 53
3.3 Proposed Solution 55
3.3.1 Unified Type of Communication Between Sub-systems 55
3.3.1.1 Reusability of Sub-systems and Domain Dependency . . 57
3.3.2 Design of Modular Architectures in the Framework 58
3.3.2.1 Weighting Between the Top-Down and Bottom-Up Design 58

3.3.2.2 Evaluating the Suitability of Sub-Systems in the Network 59
3.3.3 Automatic Design of Architectures Specific for the Task 60
3.3.3.1 Dimensionality Reduction? 61
3.3.3.2 Constraining the Set of Available Modules 62
3.3.3.3 Constraining the Space of Available Topologies 62
3.3.3.4 Predefined Classes of Neural Modules 64
3.3.3.5 On Constraining the Dimensionality of Inputs/Outputs . 65
3.3.3.6 Defining the Agent's Goals 67
3.3.3.7 Overall Principle of Automatic Design of Architectures . 68
3.4 Simulator Design 69
3.4.1 Simulation Engine 71
3.4.2 Library - Implementing and Sharing Pieces of Code 71
3.4.2.1 Modular and Reusable Design - Robotic Operating System 72
3.4.3 Simulator Engine with the ROS Integration 74
3.4.3.1 Autonomous Design of new Architectures in NengoROS 75
3.4.3.2 Documentation and Other Resources 76
3.4.4 Example of Hybrid System Simulated in the NengoROS 76

4 Theoretical Foundation and Design of Modules 79
4.1 Neuron Models 79
4.2 Logic Gates 80
4.2.1 Theoretical Foundation 80
4.2.2 Crisp Logic Gates 82
4.2.3 Fuzzy Logic Gates 83
4.3 Inner Sources of Agent's Motivation 84
4.3.1 Theoretical Foundation 85
4.3.2 Physiological Neural Module 86
4.3.2.1 Prosperity of the Physiological Neural Module 88
4.3.3 Suggested Use of Physiological Neural Modules 88
4.4 Reinforcement Learning 90
4.4.1 Theoretical Foundation 90
4.4.1.1 Learning 91
4.4.1.2 Action Selection Methods 93
4.4.2 State of the Art - Reinforcement Learning-Related 95
4.4.3 Stochastic Return Predictor Module 97

4.4.4 Reinforcement Learning Module . . . 101
4.4.5 Action Selection Mechanism Module . . . 102
4.4.6 Suggested Use of RL Modules . . . 103
4.4.6.1 Stochastic Predictor Module . . . 103
4.4.6.2 Reinforcement Learning Module with Separated ASM . . 104
4.5 Planning . . . 105
4.5.1 Theoretical Foundation . . . 105
4.5.1.1 Stanford Research Institute Problem Solver . . . 105
4.5.2 Design of the Planning Neural Module . . . 106
4.6 Sequence and Pattern Recognition . . . 108
4.6.1 Spatial Pattern Recognition . . . 108
4.6.1.1 Design of the Self-Organizing Map Neural Module . . . 109
4.6.2 Temporal Pattern Recognition . . . 111
4.7 Neural Modules for Simulating Agent's Environment . . . 112
4.7.1 Discrete Grid-World Simulator . . . 112
4.7.2 Simulator with Realistic Physics - ViVAE . . . 114

5 Experiments **117**
5.1 Hand-Designed Architectures - Testing Neural Modules . . . 117
5.1.1 Hand-Designed Agent Controllers - Navigation Task . . . 118
5.1.1.1 Controller Design . . . 118
5.1.1.2 Neural Modules . . . 119
5.1.1.3 Resulting Architecture . . . 120
5.1.1.4 Behavior of Resulting Architecture . . . 121
5.1.2 Q-Learning Based Agent Architecture . . . 122
5.1.2.1 Architecture Design . . . 122
5.1.2.2 Testing the Learning . . . 124
5.1.2.3 Influence of Physiology Decay on Agent's Behavior . . . 125
5.2 Evolutionary-Based Design of Architectures . . . 127
5.2.1 Evolutionary Algorithms Used . . . 128
5.2.2 EA-Designed Agent Controllers - Navigation Task . . . 129
5.2.2.1 Architecture Design . . . 129
5.2.2.2 Resulting Automatically Designed Architectures . . . 131
5.2.3 Artificial Neural Network of 3rd gen. vs. Hybrid Network . . . 132
5.2.3.1 Task Description . . . 133

5.2.3.2 Optimization of ANN-Based Model 134
5.2.3.3 Optimization of Hybrid Model 137
5.2.4 EA-designed Agents with Motivation-Driven RL 139
5.2.4.1 Architecture Design 139
5.2.4.2 Defining the Agent's Goals 140
5.2.4.3 Composed-Objective Fitness 141
5.2.4.4 Single-Objective Fitness 143

6 Conclusion **149**
6.1 Fulfillment of Thesis Goals 150
6.2 Main Findings of the Thesis 151
6.3 Known Limitations of the Research 152
6.4 Future Directions and Practical Use 152

Bibliography **153**
Author's Publications Related to the Thesis 169
Other Author's Publications 170
Citations of Author's Publications 170

A Additional Knowledge on Spiking Neural Networks **I**
A.1 Selected Models of Spiking Neuron I
A.1.1 Leaky Integrate-and-fire Model of Neuron I
A.1.2 Izhikevich's Simple Model of Neuron III
A.2 Neural Engineering Framework IV
A.2.0.1 Neural Encoding Process V
A.2.0.2 Neural Decoding Process VIII
A.3 Comparison of current ANN simulators X

List of Abbreviations

FDR	Fixed-sparsity Distributed Representations
HTM	Hierarchical Temporal Memory
CLA	Cortical Learning Algorithm
HTN	Hierarchical Task Network
EA	Evolutionary Algorithm
GA	Genetic Algorithm
RGA	Real-Valued Genetic Algorithm
GP	Generic Programming
CGP	Cartesian Generic Programming
NEAT	Neuro-evolution of Augmented Topologies
HyperNEAT	Hypercube Neuro-evolution of Augmented Topologies
GEP	Gene Expression Programming
RL	Reinforcement Learning
HRL	Hierarchical Reinforcement Learning
SRp	Stochastic Return Predictor
ANN	Artificial Neural Network
DS	Decision Space
HARM	Hierarchy, Abstraction, Reinforcements, Motivations Agent Architecture
STRIPS	Stanford Research Institute Problem Solver
MDP	Markov Decision Process
SMDP	Semi-Markov Decision Process
MAS	Multi-Agent System
MA	Memetic Algorithm
BDI	Belief-Desire-Intention autonomous agent architecture
AI	Artificial Intelligence
RL-TOPs	Reinforcement-Learning Teleo-Operators

TR	Teleo-Reactive planning system
TOP	Teleo-Operator
TD	Temporal Difference learning method
GPU	Graphical Processing Unit
GPGPU	General Purpose Graphical Processing Unit
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
API	Application Programming Interface
MPI	Message Passing Interface
IDL	Interface Definition Language
ROS	Robotic Operating System
SLAM	Simultaneous Localization and Mapping
CPPN	Cartesian Pattern Producing Network
CNS	Central Nervous System
LFP	Local Field Potential
FPGA	Field-Programmable Gate Array
GMDH	Group Method of Data Handling
MIA	Multi-layer Iterative Algorithm
FAKE	Fully Automated Knowledge Extraction
GAME	Group of Adaptive Models Evolution
KNN	K-Nearest Neighbor Algorithm
HOAP	Humanoid for Open Architecture Platform
MCS	Multiple Classifier System
TDNN	Time Delay Neural Network
LTM	Long-Term Memory
STM	Short-Term Memory
DBN	Deep Belief Network
NEF	Neural Engineering Framework
SPAUN	Semantic Pointer Architecture Unified Network
SARSA	State-Action-Reward-State-Action RL algorithm
PNM	Physiological Neural Module
PDDL	Planning Domain Definition Language
SOM	Self-Organizing Map
PCA	Principal Component Analysis
HMM	Hidden Markov Models

HNN	Hybrid Neural Network
CF	Composed Fitness
SOGA	Simple-Objective Genetic Algorithm
COGA	Composed-Objective Genetic Algorithm
CORGA	Composed-Objective Real-Valued Genetic Algorithm
MOEA	Multi-Objective Evolutionary Algorithm
SORGA	Simple-Objective Real-Valued Genetic Algorithm
HANNS	Hybrid Artificial Neural Network Systems
GUI	Graphical User Interface
S2F	Spike-to-Float
F2S	Float-to-Spike
BMU	Best Matching Unit

List of Symbols

t	Time
v	Membrane potential of neuron
u	Recovery variable for Izhikevich simple model of neuron
I	Current
I_R	Current on resistor
I_C	Current on capacitor
C	Capacity
R	Resistance
τ	Time constant
$t^{(f)}$	Spike - event characterized by a firing time of a neuron
Δ^{abs}	Refactory time of Leaky Integrate-and-Fire model of neuron
ϕ_i	Output weights of neurons in NEF
$J(x)$	Transformation from input to soma currents in NEF
$G(x)$	Transformation from soma currents to neural activity in NEF
$\alpha_i(x)$	Input transformation in NEF
$M(t)$	Amount of Motivation produced at the time step t
$I(t)$	Amount of Importance produced at the time step t
$P(t)$	Value of the Prosperity output at the time step t
$V(t)$	Value of the Physiological Variable at the time step t
MSD_t	Value of the Mean State Distance to optimal conditions until the time step t
Q	Utility matrix
A	Set of Actions
S	Set of States
r_t	Reward received at time step t
γ	Forgetting factor

λ	Decay parameter
α	Learning rate
δ	Change in the Utility matrix per one step
ϵ	Probability of randomization of the strategy

List of Figures

2.1 Jordan type of ANN for prediction of words in sentences. 14
2.2 Schematics of Unified Hybrid System 14
2.3 Transformational Hybrid System 16
2.4 Example of RL decision space decomposition 17
2.5 Example of Hierarchical Task Network Planning 18
2.6 Needle in the haystack and Baldwin effect 21
2.7 An example of simple encoding of ANN topology 22
2.8 Typology of software agents 25
2.9 Typical processing of information in a robotic system 25
2.10 Layered Architecture 26
2.11 Hybrid design for robotic navigation. 27
2.12 The scheme of connectionist cognitive architecture Leabra. 28
2.13 The scheme of ACTR cognitive architecture. 29
2.14 Schematics of the SPAUN architecture. 32
2.15 MIA-GMDH compared to GAME. 35
2.16 Meta-learning: network of supervised classification algorithms. 36
2.17 Example of CGP-designed logic circuit implementing one-bit adder. . . . 37
2.18 Description of the MALEVA framework. 38
2.19 Time Delay Neural Networks 40
2.20 Dual neuron for Short-Term Memory 41
2.21 Structure of model of Long-Term Memory 42
2.22 Categorizing and Learning Module 43
2.23 Long Short-Term Memory 44
2.24 The basic schema of an ESN. 45
2.25 Generator of sinusoidal signal. 46
2.26 Model of spinal cord for robotic salamander. 47

2.27 Central Pattern Generator-based control of walking. 48

3.1 Basic type of use of an agent architecture. 53
3.2 Principle of communication between Neural Modules. 56
3.3 Design of new architectures based on a particular task. 58
3.4 Comparison of top-down and bottom-up-designed hybrid systems. 59
3.5 Pre-selection of Neural Modules to be used in the architecture. 62
3.6 Example of feedforward hybrid architecture. 63
3.7 An overall principle of automatic design of architectures. 68
3.8 Number of ROS repositories. 72
3.9 Scheme of the current definition and implementation of Neural Module. . 73
3.10 An overall design of the NengoROS simulator. 75
3.11 Example of simple hybrid system in NengoROS simulator. 76
3.12 Course of simulation in NengoROS simulator. 77

4.1 General principle of connecting a logic gate into the HANNS network . . 81
4.2 An example of Neuro-Fuzzy system. 81
4.3 An example of crisp logic gate implementing the AND function. 83
4.4 An example of fuzzy logic gate implementing some Fuzzy OR function. . 84
4.5 Graphical representation of Physiological State Space. 87
4.6 Graphical Representation of Physiological Neural Module. 88
4.7 Example of course of value of Motivation produced by the PNM. 89
4.8 Example of simple grid environment which is difficult to explore. 94
4.9 Schematics of the Stochastic Return Predictor implementation. 98
4.10 Stochastic Return Predictor communication. 99
4.11 Stochastic RL Neural Module. 102
4.12 Basic Schematics of the ASM Module. 103
4.13 Graphical representation of the Planning Neural Module. 106
4.14 Graphical representation of SOM Neural Module. 109
4.15 Graphical representation of the Sequence Recognition Neural Module. . . 111
4.16 An example of GridWorld simulator. 112
4.17 Graphical representation of the GridWorld simulator. 113
4.18 An example of ViVAE simulator window. 115
4.19 Graphical representation of the ViVAE Simulator Server. 116

5.1 ViVAE environment with simple "maze". 118

5.2 Two simple Neural Modules for navigation task. 119
5.3 Simple hand-designed architecture composed of two Neural Modules. . . 120
5.4 Course of sensory and actuator data during the navigation task. 122
5.5 Setup of experiment testing SRP Module with one source of reward. . . . 123
5.6 Course of agent's learning during 100000 time steps of simulation. 124
5.7 An example of learned strategy after 40000 simulation steps. 125
5.8 Influence of Decay parameter of the PNM on learning. 126
5.9 Influence of Decay parameter of the PNM on learning - higher decay. . . 127
5.10 Principle of encoding of a hybrid agent architecture into genome. 130
5.11 Example of behavior of RGA-designed architecture for navigation. 132
5.12 Principle of EA-based optimization of connection weights. 134
5.13 Optimized connections in the (hybrid) model. 135
5.14 Example of hybrid modular system - recurrent ANN of third generation. 136
5.15 Course of evolution of the ANN-based system. 137
5.16 An example of performance of the ANN-based system. 138
5.17 Results of agent controlled by hybrid ANN. 138
5.18 Principle of encoding of hybrid agent architecture as a feedforward HNN. 139
5.19 Evolutionary design of (hybrid) ANN agent architecture - CORGA. . . . 143
5.20 Behavior of agent produced by the CORGA and COGA algorithms . . . 144
5.21 Evolutionary design of (hybrid) ANN agent architecture - SORGA. . . . 145
5.22 Analyzing the typical architecture found by the SOGA. 146
5.23 Behavior and knowledge learned in SORGA-designed architecture. 147
5.24 Visualization of the greedy policy learned by the SORGA-designed agent. 148

A.1 Scheme of Leaky integrate-and-fire neuron II
A.2 Three stereotypical neuron response functions from human cortical cells. V
A.3 A typical neuron tuning curve that codes for horizontal eye position. . . . VI
A.4 Graph showing generated tuning curves for neural ensemble with 50 nodes. VII
A.5 Example of representational transformation by tuning curves. IX
A.6 Example of experiment in Emergent simulator. XII
A.7 XOR implemented by network of Izhikevich's neurons. XIII
A.8 Example of simulation in Biological Neural Network Toolbox. XIV
A.9 Example of simulation in SpikeStream and NeMo. XV

List of Tables

5.1 Typical parameters of GA and RGA used. 128
5.2 Parameters of RGA used in the navigation task. 130
5.3 Parameters of RGA used for evolving the (Hybrid) SNN. 133
5.4 Parameters of GA and RGA used for RL-based architectures. 141
5.5 Comparison of typical EA-designed agent architectures. 142

A.1 Overview of Available ANN Simulators XI

List of Algorithms

1 Function of the Stochastic Return Predictor. 97
2 High-level operation of the required domain-independent planner. 107
3 Operation of the SOM Neural Module. 109
4 GridWorld simulator operation. 113
5 Generational model of Evolutionary Algorithm used in experiments. . . . 129

Chapter 1

Introduction

Times when one researcher was able to fully understand all available science fields are gone for a long time. In these days, there is so much information available, so that one is often not able to read everything even from his research field. This causes the situation where each research domain is divided into smaller and smaller pieces. Researchers are then able to read all information they should know, but this also often causes that people are loosing wider overview of situation. Good example of this situation can be seen in biology and medical research. Nowadays, there is huge amount of highly specialized and isolated articles. Therefore probably no one will ever be able to take a general overview of current knowledge. This means that it is very hard to steer the direction of research in some potentially more useful way. One goal of this thesis is to make at least small difference in the situation.

1.1 Overview

Aside of highly specialized research, there is also need for some systematic integration of new knowledge. This approach should be able to reuse current knowledge and combine results of research across selected research fields or sub-fields. For example, in neuroscience this goal tries to reach project called Blue Brain (Markram 2006). This project tries to use supercomputer to integrate knowledge across the neuroscience research into one huge model of brain. Aside of model of working brain (as they claim), this project

should provide new knowledge simply by gaining the overview of what is known in the field today. Projects of this type require collaboration of many people, my opinion is that projects like this are missing in many other research fields.

It is interesting to try to combine outcomes from particular research fields together. So rather than building one huge universal all-knowing model, the author would like to provide a ***tool that enables user to freely combine various pieces of research in some natural way***. The proposed method of composing pieces of knowledge should be also as universal as possible.

Establishing some *unified framework*, which defines how the current subsystems should be interconnected will open even new opportunities for us. Next, probably even more interesting step is to create a system that is capable of ***autonomous building new things*** from these currently existing standardized subsystems.

1.2 Motivation

The biggest problem about combining various pieces of code together lies in their compatibility. Which communication protocol should we choose so that it will be suitable for all kinds of possibly used sub-systems? Of course there is no ideal solution to this question.

Since it is known that an arbitrary function can be approximated by feed-forward neural network with only one hidden layer (Cybenko 1989; Hornik, Stinchcombe, and White 1989), it can be assumed that any behavior of arbitrary complexity can be build by means of neural networks with more complex topologies and/or more complex models of neurons. Therefore neural networks can be seen as theoretical way how to build an arbitrary system. Topologies of biological neural networks have highly structured and modular form. There are many attempts to exploit modularity for more efficient design of artificial neural networks (Auda and Kamel 1999). There are also successful methods of exploiting repetition and symmetry for evolution of these artificial networks (Stanley, D'Ambrosio, and Gauci 2009). Recently, it has been shown how this modularity emerged during the evolution (Clune, Mouret, and Lipson 2013) by taking into account cost of connections between neurons.

1.2.1 Brief Description of the Approach

These are the reasons why the author decided to use modification of framework for Artificial Neural Networks (ANNs) for representing these modular systems with heterogeneous nodes. Cornerstone of ANNs if neuron. Given part of ANN which can implement more complex behavior and consists of one or more neurons can be called module (Auda and Kamel 1999). Therefore an arbitrary subsystem is represented here as a **"Neural Module"**. In order to be able to connect sub-systems of various nature together in a seamless way, the original ***framework called Hybrid Artificial Neural Network Systems (HANNS) was created***. The HANNS defines a common representation of subsystems together common communication protocol. The thesis is not interested in designing systems similar to those that we can observe in the nature, but in designing systems that could be created by combining state-of-the-art knowledge from various sub-fields of AI together. This means that a given representation of subsystems is not necessarily biologically plausible.

This HANNS framework combines classical top-down approach (represented by subsystems encapsulated in so-called "Neural Modules") with benefits of ANNs, while suppressing their drawbacks. List of main advantages of these hybrid neural networks, which combine classical AI with neural networks are following:

- ability to combine arbitrary current subsystems, fast prototyping of new systems
- greatly improves capabilities of ANNs by mixing these black-boxes with subsystems of known inner structure, better overall understanding of resulting systems (compared to ANNs)
- improvement of currently known systems with well-known benefits of ANNs (associativity, noise-robustness etc..)
- provides unified method of connecting these systems together, therefore enable automatic creation of new modular architectures.

In this thesis, the HANNS framework - a system able to represent various pieces of software together in a seamless way - is presented in more detail. The functionality of the framework is presented on designing several types of agent architectures. Furthermore, the original approaches of composing agent architectures of the framework are described. Finally, it is shown how the ***framework can be used for automatic design of new***

agent architectures for a given task. That is: given set of Neural Modules together with their inputs and outputs, the Evolutionary Algorithm (EA) is used for optimizing connection weights between these Modules in order to provide an architecture with a desired behavior.

1.3 Thesis Outline

In the thesis, the focus will be put on selected parts of the project. From reasons mentioned above, this thesis is dedicated mainly to defining the suitable framework for unified representation of various modules. Then, the requirements for automatic design of new architectures will be defined. The thesis has following goals:

Goal 1 Creating a framework and tool that will enable fast integration of knowledge (concretely pieces of code) in bigger systems. This tool will be able to simulate and test the resulting systems.

Goal 2 Proving that the framework can be used (for design by hand) across the concrete research domains to solve problems by means of hybrid approaches.

Goal 3 Exploring possibilities of how this tool can be used for autonomous design of new modular systems composed of these pieces of knowledge.

1.4 Structure of the Text

The following Chapter 4, called State of the Art has several main parts. First, various types of Artificial Neural Networks (ANNs) are described, together with the evolutionary design of ANNs. Then, selected agent architectures are described. After that, selected related methods of designing modular systems are mentioned. Finally a small overview of existing "Modules" that are potentially suitable for use in the HANNS framework is shown.

Based on this knowledge, the Chapter 3 - Problem Analysis and Proposed Solution - describes the novel ideas proposed by the author. The Chapter describes the task to be

accomplished, then the HANNS framework is described in more detail. After that, the requirements for automatic design of new architectures in this framework are described. Finally, the simulator designed for simulating such a general-purpose modular systems is described.

The Chapter 4 - Theoretical Foundation and Design of Modules - describes implementations of selected systems as Neural Modules. For each type of Neural Modules, a brief own theory (and possibly state-of-the-art) sections are mentioned separately. Each Section describes one selected type of Neural Modules, that are later used in the experiments.

The Chapter 5 - Experiments - shows two types of experiments. First, the practical use of the HANNS framework together with the proposed NengoROS simulator are shown on simpler hand-designed examples. In the second part, the evolutionary approach is used for automatic design of new architectures for a given task.

The Chapter Conclusion then briefly concludes results of the entire thesis and outlines directions of future research.

Chapter 2

State of the Art

"In our opinion basing an approach to AI on a single kind of representation, symbolic or sub-symbolic, is going to falter. Each representation has limits to what it can express. Just as human intelligence is based on an interaction between declarative and procedural knowledge, so also artificial intelligence is going to need to incorporate both symbolic and sub-symbolic techniques if it is going to overcome these limitations." (Ross 2002).

Generally, this chapter describes selected knowledge related to automatic, or semi-automatic design methods of modular systems. The chapter should also provide information that this approach is not completely built from nothing. In some way similar, but very specialized one-purpose systems are already here. This thesis proposes more general approach, which can explore *combinations of subsystems from completely different research areas.*

The chapter starts with basic introduction of Artificial Neural Networks, then describes some interesting types of neural networks and circuits. Then the Evolutionary Algorithms and their use in design of ANN topologies are described. The chapter then ends with description of several architectures that are somehow related to this work.

2.1 Artificial Neural Networks

Differences between artificial neural networks and classical computation executed by computers are mentioned in various literature. Just to mention that Von Neumann type of computer implements from its principle deterministic computation based on exact data. The bottleneck of this architecture poses constraints to the maximum speed of computation. The fact that individual information have to be distinguished without errors causes consumption of considerable amount of energy. Also the second characteristic makes the architecture unfeasible for processing the real-world data.

Compared to this, brain is computational system designed by evolution with the following requirements:

- highly power-efficient
- highly robust against errors in hardware/sensory data
- works well with real world noisy data.

These requirements probably determined that the brain implements highly parallel type of computation, that the computation as well as memory is decentralized. We can see that these features are often direct opposite to von-Neumann architecture, so as to many computation models based on it.

There are several reasons why the ANNs are not used more widely, one of them is that neural networks often work as a black-boxes and we do not have sufficient methods to design networks of appropriate size.

My thesis tries to connect these two directly opposite approaches tightly together, so that the benefits of both approaches will be combined, while suppressing their drawbacks. First, this chapter will mention classification of ANNs according to various criteria. Then the role of modularity in these networks is discussed.

2.1.1 Classification of Artificial Neural Networks

Artificial Neural Networks (ANNs) can be divided according to several main criteria. These are particularly the following: Focus on **neuron model** vs **network of neurons**, classification by the **network topology**, types of **neuron model** used in the network

and finally, **the design approach** used for defining network topology and/or connection weights between neurons.

The following subsections will briefly divide Artificial Neural Networks (ANNs) by these criteria. Since the main focus is put on the network's behavior/computation, the models which focus on modeling single neuron will be omitted here. These models are usually very biologically accurate and therefore complex. Creating an useful ANN composed of these accurate models is often so computation expensive, that specialized HW or supercomputers (as in case of SyNAPSE project) have to be used. Nowadays, these networks are mainly used for more medical-oriented simulation.

2.1.1.1 Common Topologies of Artificial Neural Networks

Since the main types of topologies of ANNs are well known, let's note that those main are: ***feedforward, recurrent, bi-directional*** and ***Self Organising Maps (SOMs).*** Most often, the ANNs are used for transformation of high-dimensional input space into some resulting lower-dimensional space. Feedforward ANNs are capable of recognizing patterns in input data[1], while recurrent networks are able to represent also previous inputs, therefore are able to process time series data in some way. There are either fully recurrent (e.g. Hopfield network (Hopfield 1982)) or partially recurrent (e.g. Jordan (Jordan 1986) and Elman (Elman 1990)) networks. Very brief description of common network topologies can be found e.g. in (Krenker, Bešter, and Kos 2011; Wilamowsky 2003), or comprehensive introduction to ANNs in (Rojas 1996).

There are many types of combinations of these topologies, such as SOM with additional recurrent connections RecSOM (Voegtlin 2002). Here should be also mentioned topology called Deep Belief Network (DBN) (Bengio 2009). DBN models are generally described as multi-layer feedforward (hierarchical) topologies, where layers (sub-networks) are trained separately, one after another. Probably the most common type of DBNs uses Restricted Boltzmann Machines, but other types of neurons are used as well. DBNs are inspired in hierarchical data processing in biological brains (e.g. visual system) and are gaining big success lately, e.g. in visual recognition (Tang and Eliasmith 2010). These examples of more complex topologies (RecSOM and DBN) can be also classified as Modular Neural Networks (MNNs) as mentioned in the Chapter 2.2.1.

[1] Feedforward topologies without some preprocessing of time series, e.g. sliding window.

2.1.1.2 Classification of Neuron Models

According to the type particular computation units, Maas (Maass 1996) divides the neural networks into three main types: The ***first generation*** is based on McCulloch-Pitts neurons, also referred to as perceptrons. These networks gave rise to a variety of neural networks models as multilayer perceptrons or Hopfield nets. The characteristic feature of networks of this type can *provide only digital output*, but they are universal for digital computation and every boolean function can be computed by some multilayer perceptron with single hidden layer.

The ***second generation*** is based on the computational units that apply *continuous "activation function"* to the weighted sum of continuous inputs. The most common activation function is the sigmoid function: $\delta(y) = 1/(1 + e^{-y})$. Typical examples for networks from this second generation are feedforward and recurrent sigmoidal neural nets, as well as networks of radial basis function units. These networks are also able to compute (with the help of thresholding at the network output) arbitrary boolean functions. Furthermore, it has been shown that these neural nets can compute certain boolean functions with fewer gates than neural nets from the first generation. In addition, neural nets from the second generation are able to compute functions with analog input and output and are universal for analog computations in the sense that any continuous function with a compact domain and range can be approximated arbitrarily well by a network of this type with a single hidden layer. Another characteristic feature of this generation of neural network models is that they support learning algorithms that are based on gradient descent such as back-propagation. For a biological interpretation of neural nets from the second generation one views the output of a sigmoidal unit as a representation of the current firing rate of a biological neuron (Maass 1996). Several special types of neuron models, such as Radial Basis Function (RBF) (Broomhead and Lowe 1988), can be also classified into the second generation of neurons.

In 1982 it was shown that the monkey can recognize face in less than 100ms (later even in 20-30ms), while the firing rates of these neurons are usually bellow the 100Hz (Perrett, Rolls, and Caan 1982). It means that biological organisms use not only firing rates but also temporal combination of particular spikes. This gives rise to more biologically accurate type of neural networks: the ***third generation***, which uses model of spiking neuron (Maass 1996). Recent progress in computer technology enables us to simulate also this type of neural networks. The following Section (supplemented by the Appendix

A) will briefly describe them.

Networks of Spiking Neurons: As it was mentioned, neurons of third generation of neural networks communicate by means of discrete spikes. It is an interesting type of communication, because it can implement both, analog and discrete communication. It is known that in different parts of Central Nervous System (CNS), different types of *neural coding* are used.

Rate-based codes are used in places where represented values change slowly in time. Example of use of rate code can be seen in representation of muscle contractions. Practical example of this coding for representation angle of arm of humanoid robot can be seen in (Gamez, Fidjeland, and Lazdins 2012). Compared to this, temporal codes (where individual spikes play role) are used when the represented value changes rapidly, for example visual stimuli (Perrett, Rolls, and Caan 1982). It is believed that temporal codes are synchronized de-centrally by the Local Field Potential (LFP) - average activity of neurons in predefined area (Kraskov et al. 2007). The LFP periodically changes according to neural oscillation (brain waves). This oscillation is generated spontaneously by groups of neurons (Strogatz 1997) and can be observed on all levels of organization.

There is increasingly high number of applications of these spiking neural networks currently. This is due to fact that simulation of these networks is recently more and more feasible. These models of neurons can be simulated on classical personal computers or specialized hardware. The most cost-efficient way of simulating large-scale Spiking Neural Networks (SNNs) is by accelerating the computation of General Purpose GPUs (GPGPUs) (Yudanov 2010; Nageswaran and Donald 2009; Poggio, Knoblich, and Mutch 2010; Krichmar, Jayram M. Nageswaran, and Richert 2010; Fidjeland, Roesch, et al. 2009). One entire research field deals with developing specialized hardware for these networks, as for example SpiNNaker (Rast et al. 2010), Neurogrid (Boahen 2006), Spin Devices-based project sponsored by Intel (Sharad et al. 2012), European-funded FACETS[2] or DARPA-funded project called Systems of Neuromorphic Adaptive Plastic Scalable Electronics (SyNAPSE).

Spiking neurons are more expressive (powerful) than older models. It was shown that this third generation of neural networks is superset of the second generation. Each function/behavior that can be produced by ANNs of 2nd generation can be produced also with the 3rd generation with the same, or smaller amount of neurons.

[2]http://facets.kip.uni-heidelberg.de/

There are many models of neurons of third generation which are able to produce spikes in a similar way as their biological counterpart. An example application can be seen in modeling of visual cortex for image recognition (Yu et al. 2013; Eliasmith 2013). Since we often need to simulate many of these neurons and these models are computationally relatively expensive, there is trade-off between model accuracy and computational requirements. Two selected models of spiking neurons are mentioned in the Appendix A in section A.1.

2.1.1.3 General Methods of Artificial Neural Network Design

As described in our currently submitted paper, the ANNs have many desired properties, but the main problem still lays in insufficient methods of determining correct (optimal) topology of network with desired behavior. The ANNs can be divided also by means of design approaches. The main ones are the following three:

- **Learning Algorithms:** In this case, the network has predefined topology (e.g. feedforward network) and a *local learning rule*, which modifies connection weights between particular neurons. Here can be mentioned supervised learning in feedforward network by means of back-propagation algorithm, or unsupervised Hebbian or competitive learning.

- **Topology Optimization:** Compared to the previous case, the topology of ANN can be optimized by *globally operating optimization algorithm.* The topology is often partially predefined (e.g. to the feed-forward networks (Leung et al. 2003)) and the Evolutionary Algorithm (EA) is used to find correct weights. Currently, this approach is able to provide relatively complex systems, ranging from design of controllers generating coordinated quadruped gaits (Clune, Beckmann, et al. 2009), controlling bots in video-games (Stanley, Bryant, and Miikkulainen 2005). Here, the crucial parts are in predefining suitable topology constraints and in choosing suitable representation of ANN weights for the EA (Fekiac, Zelinka, and Burguillo 2011).

- **Neural Engineering:** This represents the *Top-Down* approach in designing neural systems. Instead of starting from an individual neuron, it works over populations of neurons. Each population has purpose of solving particular part of the problem. Here, the qualitative tools are often used to compute particular connections between

neurons and/or between populations of neurons. These methods are often used in larger-scale neural models (Garis et al. 2010; Eliasmith 2013). As examples can be mentioned Central Pattern Generators (CPGs) (Marder and Calabrese 1996; Zainer and Nagashima 2002) or Neural Engineering Framework (NEF) (Eliasmith and Anderson 2003). Naturally, this approach works with modular networks and therefore is suitable for designing hybrid neural systems.

2.2 The Role of Modularity

In simpler problems, it is often sufficient to use one homogeneous solution (e.g. simple ANN topology, as described in the Chapter 2.1.1.1) for solving the task with sufficient accuracy. However, many real-world problems (such as vision, language processing etc.) are too complex to be tackled with a single method. One of well known solutions to the course of dimensionality is approach so called *divide and conquer*. The task is divided into smaller sub-tasks and these are solved separately. This method brings some significant benefits compared to solving each problem from the scratch. The most obvious benefit is in often dramatic reduction of problem complexity. One of the main advantages is also (once discovered structure of the problem) in opportunity of re-using the discovered solutions to sub-problems. Various methods of decomposition of problems into smaller pieces was widely investigated in many different applications. Several selected types of exploiting the structure of problems are described in the following sections.

2.2.1 Modular Neural Networks

In the context of ANNs, each of sub-tasks can be solved by one ANN and the resulting solution can be composed of these sub-solutions. It is convenient to represent this approach as one big network, which is composed of interconnected sub-networks (or generally called ***modules***). This approach is called Modular Neural Networks (MNNs) (Boers and Kuiper 1992; Auda and Kamel 1999).

Similarly to a general case, MNNs have many advantages over big homogeneous networks. By decomposing one big network into smaller pieces, the human is able to better

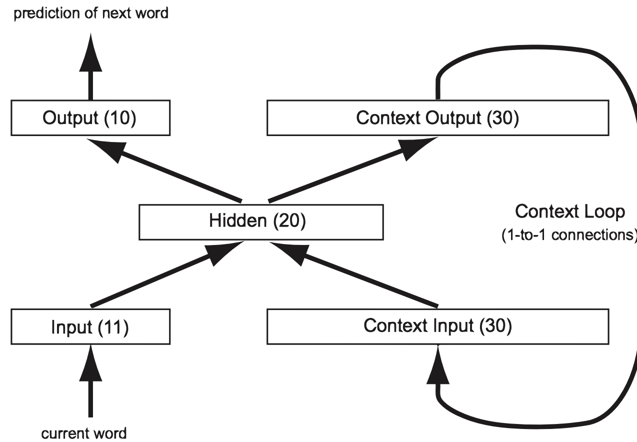


Figure 2.1: *Jordan type of ANN for prediction of words in sentences (Sugita and Butz 2008). Input and output modules communicate with outer world, while hidden module representing the state of the network depends also on the context of current input values.*

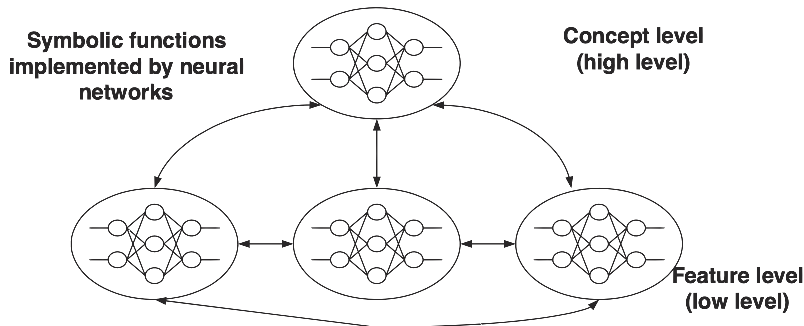


Figure 2.2: *Schematics of Unified Hybrid System composed of sub-networks implementing classical ANNs (Mcgarry, Wermter, and Macintyre 1999).*

understand the behavior of the entire system. Thus turning one big black-box into a set of smaller white, or gray boxes. The functionality of these modules is often inspired in biological brains and therefore may have various degree of biological plausibility. For example, a module with purpose of learning of sequences can be implemented either as a network of neurons (see the Chapter 2.6.1 for examples), or by means of some designed algorithm (for example Apriory algorithm, described e.g. in (Agrawal and Srikant 1994)). Despite the fact that the publication (Mcgarry, Wermter, and Macintyre 1999) deals with hybrid systems, this thesis will use its terminology to classify MNNs into two main types (of three described there):

- **Unified Hybrid Systems** are composed of modules which are implemented by

the same ANN model. Such a network can be then represented either as MNN, or as a monolithic ANN, defined by set of nodes and weights between them[3]. As an example of these networks can be mentioned Jordan network (depicted in the Fig.2.1, where each module represents one sub-network), Elman Network, RecSOM, or Deep Belief Network (DBN) (Bengio 2009) with the same type of sub-networks in each layer used. The main advantage of this type of networks is in decreasing overall number of connections between nodes. The form of particular modules can be found by any of the methods described in the Chapter 2.1.1.3. When a purpose of particular module is selected correctly, the learning process (or optimization of ANN topology by means of EA) is significantly faster, or the top-down design process can be significantly simpler. A graphical representation of such Unified Hybrid System can be seen in the Fig.2.2.

- **Modular Hybrid Systems**. So called *No Free Lunch Theorem* tells us that there is one algorithm (e.g. optimization technique), which is suitable for some set of tasks (Wolpert and Macready 1997). But there is another algorithm, which is superior in another set of problems. Theoretically, solving all problems by means of some kind of ANN is possible. But practically there are often well-know better ways for some problems. This gave rise of this type of MNNs, where different modules can implement different algorithms (not necessarily ANN-like ones). There was many systems of this kind developed in the past and many can be classified as Modular Hybrid Systems.

Least but not last, the advantage of modular systems is in re-usability of particular modules. These discovered (e.g. by means of optimization) or engineered sub-systems (modules) can be used to compose another modular system (than that they were developed for), which significantly reduces time required for designing new systems. As an example of extensive use of modularity in top-down design of large-scale ANNs can be mentioned Neural Engineering Framework (NEF) (Eliasmith and Anderson 2003), which is briefly described in appendix in the section A.2. Various examples of these biologically-inspired modules can be found in the Chapter 2.6.

[3]While considering a simplification of omitting prospective learning rules.

2.2.1.1 Communication in Modular Hybrid Systems

Since one of aims of this dissertation are Hybrid Systems, this chapter will describe Modular Hybrid Systems in more detail. In case that particular modules share the same information representation, the communication can be identical in the entire system. But in case that the information representation varies across modules, some kind of information transformation needs to be used in order to acquire successful communication in the system. For example, one module can be implemented by the ANN and another can process data in symbolic representation. The publication (Mcgarry, Wermter, and Macintyre 1999) describes a class called ***Transformational Hybrid Systems***. A graphical representation of such a transformational hybrid system can be seen in the Fig.2.3.

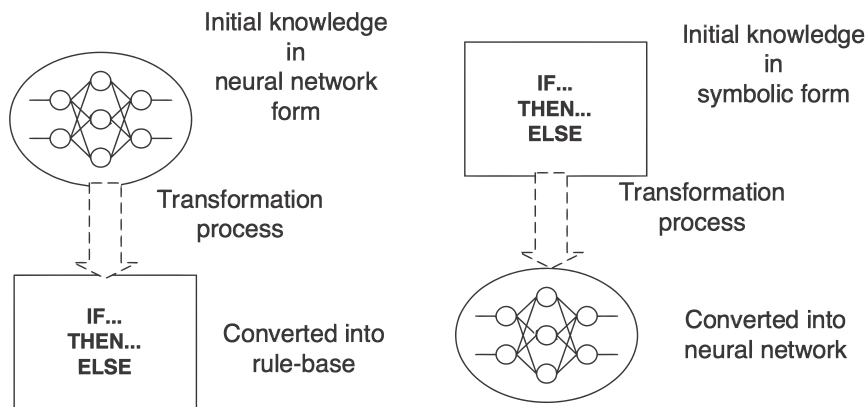


Figure 2.3: *Scheme of a Transformational Hybrid Systems. The system on the left transforms "neural" representation of information into the symbolic one. The system on the right in the opposite direction (Mcgarry, Wermter, and Macintyre 1999).*

The disadvantage of transformational systems is in the need of defining the appropriate transformation for a particular task. As one alternative can be mentioned the technique called "rule extraction", where some algorithm can be used for extracting rules from the ANN. Because of the need of the transformation process between particular modules, these systems are most often task-dependent. In contrast, the Transformational Hybrid Systems can be very powerful and therefore are widely used in various applications.

2.2.2 Hierarchical Problem Decomposition

Often, the problem can be decomposed into hierarchically organized structures. These hierarchies often contain uniform type of nodes. The nodes on the bottom of the hierarchy represent more primitive (basic) parts of the problem, while nodes placed higher in the hierarchy represent knowledge composed of lower sub-nodes. The most common problems while exploiting the structure of the problems are in requirement of a priory knowledge: how the problem can be decomposed into smaller pieces.

One example of hierarchical decomposition can be represented by **Hierarchical Reinforcement Learning** (HRL). Reinforcement Learning (RL) has often problems to represent and successfully learn in more-dimensional spaces. Where one RL module could not learn all possible states of the system, hierarchy of smaller RL modules can solve the task in a much more efficient way.

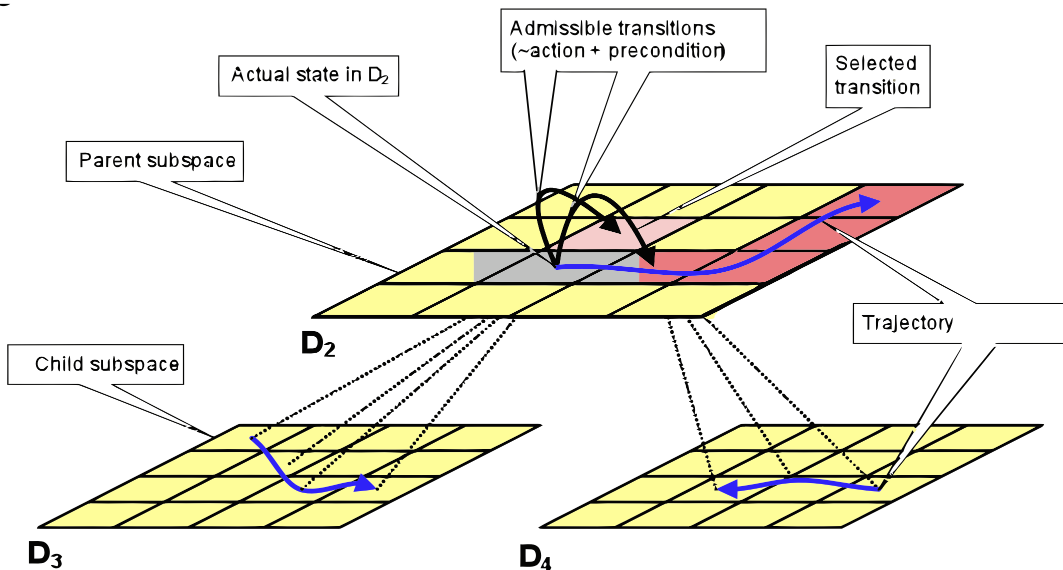


Figure 2.4: *Example of decomposition of decision space in the Hierarchical Reinforcement Learning (HRL) (Kadleček 2008). Each decision space contains either primitive actions or abstract actions. Each abstract action represents some strategy in the child decision space.*

In a system called Hierarchy, Abstraction, Reinforcements, Motivations Agent Architecture (HARM), author solved the problem of need for definition of the hierarchy a prioriby autonomous on-line generating of action hierarchy based on the interaction of agent with the environment (Kadleček 2008). Based on various types of reinforcements, the system

is capable of autonomous creation of hierarchy of RL modules. The example of two level hierarchy can be seen in the Fig.2.4. Here, two decision spaces contain primitive action and one decision space contains abstract actions (strategies in child decision spaces).

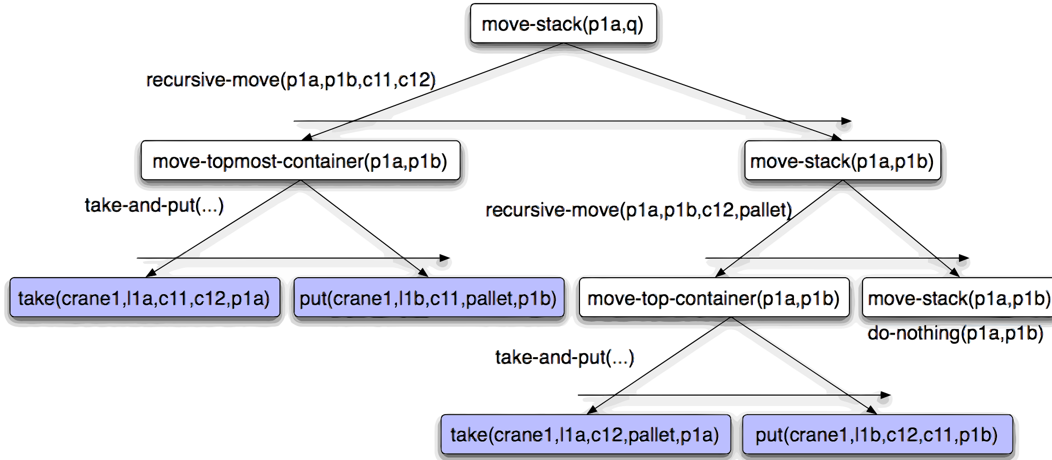


Figure 2.5: *Example of Hierarchical Task Network Planning on block-world problem. The plan is composed of primitive and non-primitive tasks. Each non-primitive task can be decomposed into less abstract tasks (Pellier n.d.).*

This can be compared to to many similar approaches, such as hierarchical problem decomposition in the domain of automated planning. Hierarchical Task Network (HTN) (Erol, Nau, and Hendler 1994) can also decompose the high number of possible states by defining the hierarchy of particular tasks. This hierarchy then contains primitive tasks (equivalent to actions in the Stanford Institute Problem Solver (STRIPS) (Fikes and Nilsson 1971)) and compound tasks, which are composed of primitive tasks and other compound tasks. There are also many available opportunities to try to exploited modularity in hierarchy.

Finally, these two examples (RL and planning) were general examples of *unified hybrid systems* in the terminology of MNNs. An example of *modular hybrid system* can be mentioned architecture presented in (Vítků 2011), where the HARM system is used to interpret abstract actions of the STRIPS planning engine, therefore the system contains RL modules together with planning module.

2.3 Evolutionary Algorithms

Basics of Evolutionary Algorithms (EAs) can be found in many publications, such as (Ashlock 2010). Very briefly: these are population-based metaheuristic optimization algorithms, which use mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. One sub-part of EAs: *"Genetic Algorithm (GA) is a method for moving from one population of "chromosomes" (e.g., strings of ones and zeros, or "bits") to a new population by using a kind of "natural selection" together with the genetics-inspired operators of crossover, mutation, and inversion. Each chromosome consists of "genes" (e.g., bits), each gene being an instance of a particular "allele" (e.g., 0 or 1)"* (Mitchell 1998). Many algorithms inspired by evolution are successful, such as evolutionary-based design of antenna for NASA (Lohn et al. 2004). Evolution can be successfully applied in the field of autonomous design of **physical structures**, e.g. robots (Durr, Mattiussi, and Floreano 2010). Also the design of **control systems** often require relatively complicated theory and complicated computation, so it is also suitable field for use of some automated design approach, such are EAs. In (Durr, Mattiussi, and Floreano 2010) there are some references for good examples in this field, the paper itself suggests the evolutionary design of neural controllers, because the input-output mapping made by neural networks can be seen as very similar to manually designed control system. This Section will describe only some selected types of Evolutionary techniques that are relevant to the Thesis.

2.3.1 Memetic Evolution Algorithms

There are three main types of evolution called Darwinian, Lamarckian and Baldwinian evolution. In the field of AI, these types are distinguished mainly by classical and a hybrid EAs, called Memetic Evolutionary Algorithms (MAs). "Memetic algorithms was the name given by Moscato (Moscato 1989) to a class of stochastic global search techniques that, broadly speaking, combine within the framework of evolutionary algorithms the benefits of problem-specific local search heuristics and multi-agent systems. MAs have been successfully applied to a wide range of domains that cover problems in combinatorial optimization, continuous optimization, dynamic optimization, multi-objective optimization etc"(Krasnogor 2012). Practically, the local search is used for smoothing the fitness landscape, which reduces the complexity of the EA's solution space (Conradie,

Miikkulainen, and Aldrich 2002).

For example, the use of local search in the MAs for enhancing newly generated individuals corresponds to learning performed by the living organisms during their life. These three types of evolution then define how is this enhancement used further. The three main types of evolution are described here:

Darwinian Evolution is a type of evolution where the success of organism (phenotype) is determined only by its genotype and thus the survival of organism depends only on its genetic constitution. The Darwinian evolution is used in **classical EAs**, where the genotype unambiguously describes the phenotype and its fitness.

Lamarckian Evolution The main difference against the Darwinian Evolution is in the fact that the characteristics acquired during the organism's lifetime (for example experiences) can be transferred to the offspring. In **Lamarckian MAs** this corresponds to the improvement of an individual by local search (or e.g. some type of learning), this improvement is then transferred to its offspring. That is: the genome now describes the individual with knowledge gained during his life.

Baldwinian Evolution is based on the theory originally published by James Mark Baldwin in (Baldwin 1896), called Baldwin Effect. Baldwin proposed that the genetic information encodes the ability to learn, organisms with better ability to learn have higher chance of survival, which causes the propagation of their genetic material. This approach is used n **Baldwinian MAs** where the individual is improved by the local search and evaluated, in the population is preserved the old, not improved, one, but its fitness is changed to the fitness of the improved individual (that is: the genotype represents an individual with some ability to learn, not the individual with learned knowledge).

Baldwin effect was studied for a long range of time in various research fields as is Evolutionary Developmental Biology (Newman 2002), Philosophy (Dennett 2003), Artificial Life (Levy 1992; Suzuki and Arita 2004) or Artificial Intelligence. But this effect (and type of evolution) plays the important role in the research field which is concerned with the evolution of Artificial Neural Networks called Neuro-Evolution. In the case of Neuroevolution, the MA can be composed of EA which optimizes the network topology (and/or weights) and some (possibly conventional) learning algorithm which operates on-line, so the main focus is put on the interactions between learning and evolution (Boers, Borst, and Sprinkhuizen-Kuyper 1995; Valdivieso et al. 2006; Nolfi 1999), and (Yao, Ieee, and

Liu 1996). Neuro-evolution is described in further details in the section 2.3.2.

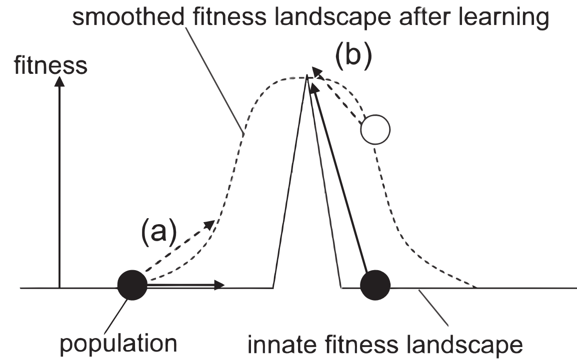


Figure 2.6: *An example of the smoothed fitness landscape after learning process. The slope of the fitness landscape is (a) increased or (b) decreased after learning process (or e.g. by the local search algorithm) (Suzuki and Arita 2007).*

The main advantage of hybridization of evolution and learning (in Baldwinian MAs) is in increasing the speed of evolution, this is caused mainly by the fact that the learning smooths the rugged fitness landscape. Hinton and Nowlan's pioneering work on the Baldwin effect (Hinton and Nowlan 1987) assumed evolution of the population on a "needle in the haystack" fitness landscape, by introducing the quantitative evolution of phenotypic plasticity (the ability of organism to change phenotype in response to changes in the environment) into a simple genetic algorithm, they showed that this effect of learning can guide the evolution of the population toward the fitness spike by increasing the slope of surface around it (Suzuki and Arita 2007), see Fig.2.6. One of the interesting papers describes how learning can guide neuro-evolution in hierarchical modular tasks (Wiles and Watson 2001).

From the practical point of view, the Baldwinian MAs store information that may be too specific for a particular task (e.g. recognizing only one type of flowers) in the genome. The individuals produced by this type of MA may store too specific information. In case that some amount generalization (or re-learning) is needed, the results may be far from the optimum. Compared to this, individuals produced by the Lamarckian MAs store the *ability to adapt* to a particular task, not its solution. Still, this type of MA is able to speed-up the evolution as depicted in the Fig.2.6.

2.3.2 Neuro-Evolution

One sub-part of evolutionary computation is called neuro-evolution. This notation stands for optimization of ANN structure (topology and/or weights) by means of EA. The crucial part of neuro-evolutionary design is in choosing the right encoding mechanism. The encoding should be able to represent the phenotype (ANN) uniquely in the genotype (genome - e.g. real-valued vector representing weights between neurons). System of conversion between genotype to phenotype should be expressive enough so that the EA is able to design ANN correctly. But encoding should also compress the representation of ANN structure, so that the searched space of ANN topologies is not too big. A simple example of encoding of ANN with binary weights can be seen in the Fig.2.7.

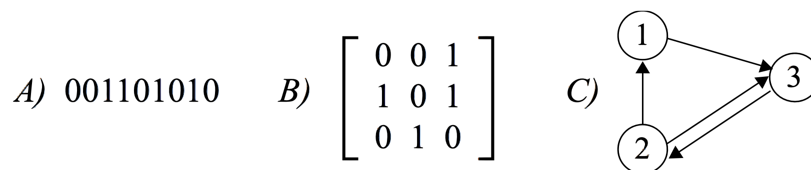


Figure 2.7: *An example of simple encoding of ANN topology into genome. A binary vector (A) (genotype) is transformed into binary adjacency matrix (B) of size $N \times N$, where N is the number of neurons in the network. The network topology is then defined as a directed graph with binary weights (C) (phenotype) (Fekiac, Zelinka, and Burguillo 2011).*

Encoding can be either direct or indirect. By means of compression, the latter one is able to evolve bigger topologies. There is many ways how to encode ANN into genome. For instance, the developmental encodings try to model development of brain during the animals life. Good overview of known basic methods how to encode ANN topology so that it will be feasible for EA is in (Fekiac, Zelinka, and Burguillo 2011).

One of the most promising, but also relatively complicated methods is called Hypercube-based Neuro-Evolution of Augmented Topologies (HyperNEAT) developed by Kenneth Stanley et al. (Stanley, D'Ambrosio, and Gauci 2009). This encoding can employ compression of ANN topologies e.g. by efficient representation of symmetry.

2.3.3 Memetic Neuro-Evolution

Memetic Neuro-evolution is a special kind of MEA which is specialized for designing ANNs. As mentioned in the Chapter 2.3.2, the neuro-evolution has two main goals: to optimize ANN topology and weights between particular neurons. Since the MEA is usually composed of two main optimization algorithms (most often there is one evolutionary-based and one local search algorithm), there are several possibilities how these two may be employed. The common types of mutation in these EAs are the following: mutation of the ANN topology is performed as random add/remove of connection between neurons. Compared to this, the mutation of connection weights is implemented as applying gaussian distribution to the current weight $w_{i,j} \in \langle 0, 1 \rangle$.

In the (Togelius, Gomez, and Schmidhuber 2008), two local algorithms ("hill-climbers") were used to design the ANN. The design of topology and weights were made "on different time scales" as follows: First, the new topology is proposed. Second, the connection weights are optimized by another local search. If the resulting ANN performs better than the previous "champion", the new solution is remembered and optimized further. Only one individual is stored in the population at a time, so the result is called *Memetic climber*.

The publication (Togelius, Schaul, et al. 2008) compares the Memetic climber to other MEAs on two different Reinforcement Learning (RL) tasks. Compared to the Memetic climber, MEAs presented in this publication use population of candidate solutions instead of only one. It was shown that these population-based MEAs outperform the Memetic climber and can solve problems that are unsolvable by non-memetic algorithms. Another example of MEA, called Symbiotic Memetic Neuro-Evolution (SMNE) can be seen in the (Conradie, Miikkulainen, and Aldrich 2002). In the publication, proposed algorithm is different from the previous ones: it combines Symbiotic EA and Particle Swarm Optimization (PSO) search.

2.3.4 Modular Neuro-Evolution

In order to reduce dimensionality of search space that has to be explored by the evolutionary algorithm (in order to create some network topology) many types of Modular Neural Networks were developed. More information can be found in (Boers and Kuiper

1992; Auda and Kamel 1999). An introduction to design and evolution of modular neural network architectures is in (Happel and Murre 1994). Exploiting the modularity of neural networks for evolving the ANN controllers is in (Durr, Mattiussi, and Floreano 2010). Some further examples how the evolution of neural networks can be employed are in (Durr, Mattiussi, and Floreano 2010; Ozawa, Tsutumi, and Baba 1999). Modular ANNs also give us the opportunity to reuse the currently discovered subsystem. For example, the designs of a modular systems which employ symmetry can be found in (Valsalam and Miikkulainen 2011).

2.4 Selected Agent Architectures

One of main aims of this thesis is in designing new architectures which control artificial agents in ALife domain. The typical use-case of these architectures is autonomous fulfilling of some given task under some given (or partially unknown) circumstances. Such an architecture may be controlling either virtual agent in a simulated environment or potentially real-world robot. The nature of these architectures is ranging from reactive to purely deliberative. Reactive agent reflexively reacts to stimuli from the environment. Compared to this, the deliberative agent maintains symbolic model of the world and decides based on symbolic reasoning (Wooldridge 1995). The symbolic model has to be build and/or maintained during the agent's life. This task requires *symbol grounding*) (Harnad 1990), which increases the complexity of the architecture significantly.

An agent architecture is usually equipped with sensory system (used for gathering information about its environment) and actuator system, which is used for interacting with the environment. This principle can be seen in the Fit.2.9. A typical agent architecture has to implement complex mapping from streams of sensory data to streams of actuator commands. In real-world robotics (but in virtual environments too) it is a task complicated enough, that some kind of system decomposition (e.g. by means of modularity) has to be employed. Furthermore, a successful robotic system has to be able to represent the task on various scales (scales of abstraction/precision, time scales etc). Modules in such a complex system then need to be ran on various levels simultaneously. Two well-known examples of reactive and deliberative agent architectures follow:

- **Layered Architecture**. Amongst those older reactive architectures there is well-

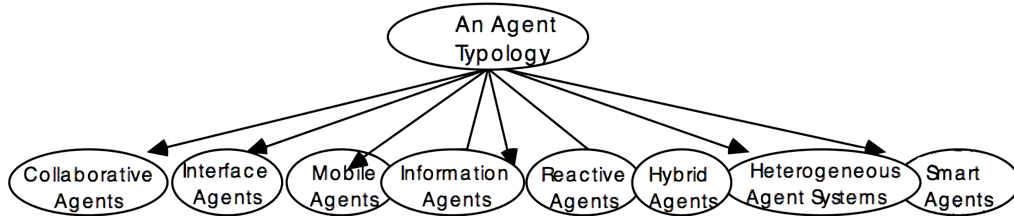


Figure 2.8: *Typology of software agents (Nwana 1996). Collaborative agents should perform tasks in collaboration with other agents, interface agents collaborate with their owners (human working on the same task). Mobile agents are software processes capable of traveling through the wide-area networks (WANs). Hybrid agents combine multiple different learning/decision making mechanisms in one bigger system. Compared to this, heterogeneous agent systems combine at least two different agents in one system.*

known Rodney Brooks's *Layered Architecture.* In the Fig.2.10 it can be seen that architecture consists of multiple layers. The higher level, the more abstract the activity is. The lower the layer is in the hierarchy, the more important the action (produced by the layer) is.

- **Belief Desire Intention Architecture**: One of the most known architectures is Belief-Desire Intention modem (BDI) (Sardina et al. 2006), which is typical representative of deliberative architectures. The BDI agent stores information about environment in form of beliefs, its objectives (goals) are stored as desires and its particular activity is driven by intentions (desire with a commitment for execution). This template of an architecture has many implementations specialized for various domains (e.g. multi vs single-agent environments).

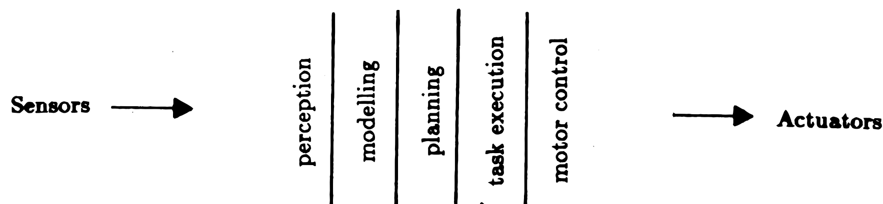


Figure 2.9: *Typical processing of information in a robotic system. First, the sensory data are pre-processed and some internal model of the environment is updated, then the plan of future actions is created. Finally, the task is executed by means of controlling the actuator system (Brooks 1986).*

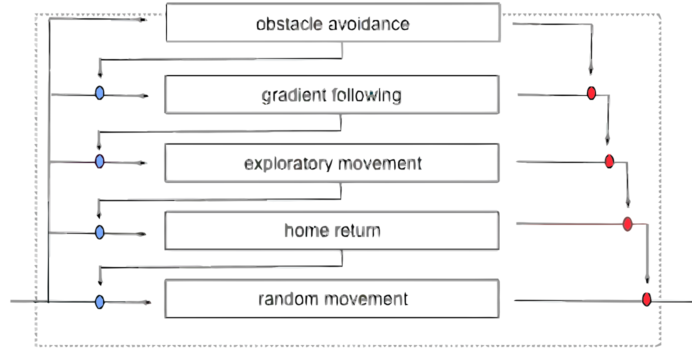


Figure 2.10: *Layered Architecture: each layer represents some level of abstraction of information abstraction. The lower in the picture the layer is, the more abstract activity it represents. Compared to this: the higher in the hierarchy, the more important action is (e.g. "return home" vs "not hitting the obstacle"). Behavior of a layer may substitute input data of the layer below it. Also, a more primitive architecture (higher in the hierarchy) can inhibit output data of more abstract layers (Briot, Meurisse, and Peschanski 2006).*

2.4.1 Selected Modular Architectures

Based on the reasons mentioned above, it is beneficial (and often necessary) to build architectures that are in some way modular[4]. Since this dissertation aims to build architectures similar to modular ANNs, this Chapter will be focused on interesting architectures of this type.

One good example of hybrid architecture, which employs spatio-temporal sequence learning for navigation of robotic system, was presented in (Nguyen, Starzyk, Tay, et al. 2010). The paper describes feed-forward hierarchical feature extraction by means of combination Long Term Memory modules (LTMs) and Short Term Memory modules (STMs)[5] in one hierarchy for navigation of robotic system. The architecture is inspired in the biological visual system. The K-iteration Fast Learning Neural Network (KFLANN) (Tay et al. 2007) was employed to establish scene STM clusters by global gist description. This implements the fast-learning behavior of scene tokens and maintains significant tolerance for disturbances in the scene. These initial experiences are stored in the STM, and then gradually consolidated and organized into LTM. Each sequence of navigating scenes is

[4]which includes also both Layered Architecture and BDI model

[5]These modules are described in the Chapter 2.6 in more detail.

stored in a LTM cell and is learnt via one-shot mechanism. During storage phase, the input sequences are stored in the corresponding LTM cells. During testing phase, the LTM cell will respond according to its degree of matching with the input sequence. The final decision's location is made by the Winner-Take-All (WTA) rule over all LTM cells. The architecture works with a streams of data, where no start nor end of sequence is marked.

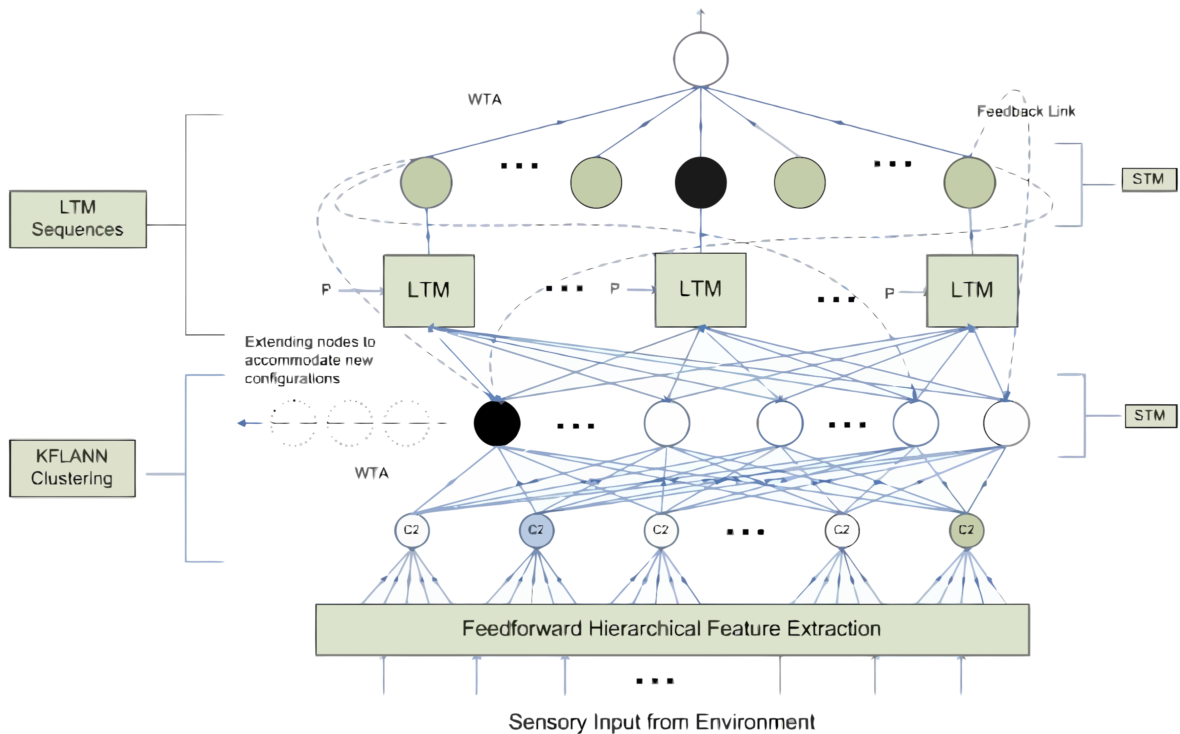


Figure 2.11: *Hybrid design for robotic navigation employing LTM and STM modules in one hierarchy. An example of hybrid system composed of heterogenous nodes (Nguyen, Starzyk, Tay, et al. 2010). The KFLANN algorithm implements STM subsystem. These short-term memories are then consolidated into LTM modules. The final recognition of scene sequences is chosen by the WTA algorithm according to the degree of matching of data in particular STM modules.*

2.4.2 Cognitive Architectures

One sub-field of agent architectures originates from cognitive science, which is often in aim of psychologist. These cognitive architectures are mainly biologically inspired and

try to model how the human mind works. Therefore here scientist try to model more complex structures than those mentioner earlier. These systems often vary in biological plausibility and we can find models ranging from purely connectionist-based to highly abstract. There are two well known architectures in this field, called *Leabra* and *ACT-R*, here will be only brief description of the main differences between them.

Leabra An example of biologically realistic cognitive architecture is system called Leabra (O'Reilly 1996). This relatively complex architecture employs several different learning algorithms together and tries to find good proportion between associative and error-driven types of learning. The structure of the architecture is depicted in the Fig.2.12. This architecture uses single neurons as basic building blocks, therefore it can be called connectionist approach. As it can be seen in the Fig.2.12, it is a modular ANN. More precisely, the architecture can be classified as a *Unified Hybrid System*, as described in the Chapter 2.2.1.

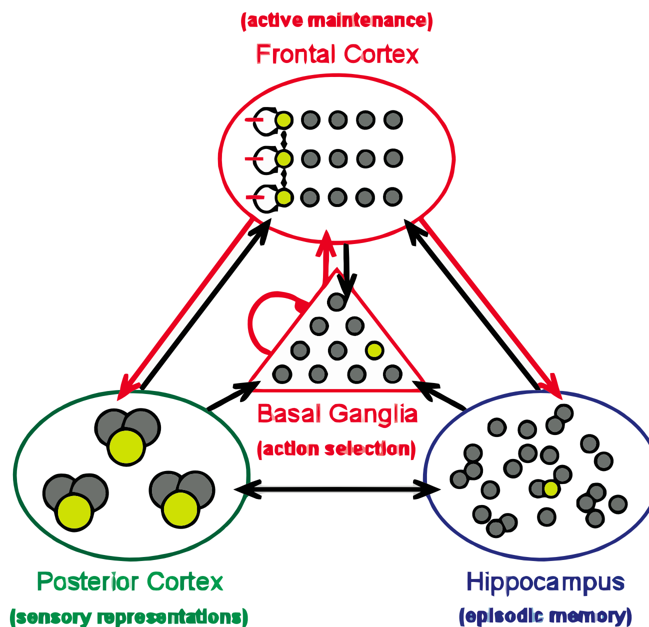


Figure 2.12: *The scheme of connectionist cognitive architecture Leabra. The system includes three major parts: the posterior cortex (for perceptual and semantic processing using slow, integrative learning); the hippocampus (for rapid encoding of novel information using fast, arbitrary learning); and the frontal cortex/basal ganglia complex (for active and flexible maintenance of goals and other context information, which serves to control or bias processing throughout the system) (O'Reilly 1996).*

ACT-R The opposite direction, holistic approach[6], represents architecture called Adaptive Control of Thought – Rational (ACT-R) (Liadal 2006). The architecture uses both symbolic and sub-symbolic systems. Compared to Leabra, the research is not focused on modeling architecture composed of single neurons. Rather the understanding overall function of parts of the brain as subsystems is investigated. Individual parts of the architecture are depicted in the Fig.2.13. It can be seen how individual sub-systems correspond to particular brain areas.

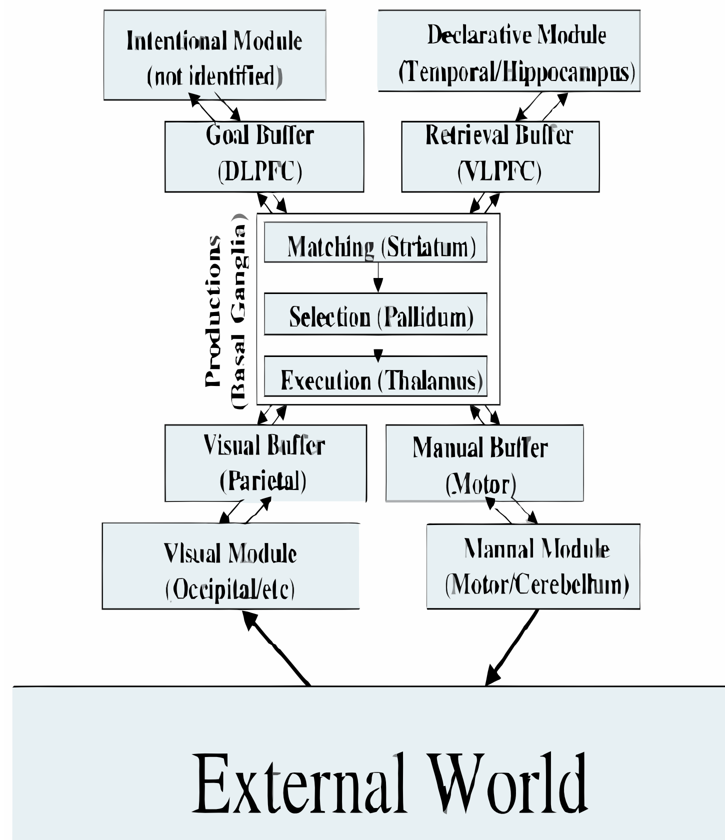


Figure 2.13: *The scheme of ACTR cognitive architecture (Liadal 2006). Compared to the schematics of Laebra, this architecture contains more single-purpose subsystems which serve to one particular purpose (e.g. the Goal Buffer). This is a typical example of the fact that it is easier to build a system with complex behavior by sacrifising some biological plausibility (such as non-neural implementation of the architecture in this case).*

SAL Each of architectures has its own pros and cons. The question whether the connectionist or holistic approach is better is still not responded completely. Rather,

[6]Which is an equivalent of a top-down designed system.

again, some kind of hybrid approach can be exploited. In the field of cognitive science this is called pluralism. Research teams from Leabra and ACT-R realized that, despite the different approach to designing the system, their architectures have very similar structure and individual modules from both architectures could be combined together. This gave a rise to explicitly pluralistic architecture called SAL. It was shown that this synthesis of ACT-R and Leabra can e.g. autonomously navigate agent in 3D environment, recognize and collect objects. As described in the paper: "SAL is an attempt to integrate and synthesise the Leabra theory of neural function, network behaviour and representation, and tripartite architecture with the ACT-R theory of symbolic and subsymbolic decision-making, representational activation and organisation, and modular architectural organisation. It also is worth pointing out that in the combined SAL architecture, most major machine learning techniques are represented, and grounded in forms that are motivated and informed by human psychology and biology" (Jilk et al. 2008).

SPAUN As mentioned in the abstract of the book (Jilk et al. 2008), *"both ACT-R and Leabra architectures are internally pluralistic, recognising that models at a single level of abstraction cannot capture the required richness of behaviour"*. The architecture called Semantic Pointer Architecture Unified Network (SPAUN) uses different approach. It is designed by the top-down (holistic) method, yet still is highly biologically plausible (both topology and behavior) and is implemented in ANN of 3rd generation. Recently, it was presented as the world's largest functional brain model (Eliasmith, Stewart, et al. 2012). Compared to other networks of biologically plausible neurons (such as (Izhikevich and Edelman 2008)), this network composed of 2.5 million neurons actually produces some required complex behavior. The basic schematics of the architecture is shown in the Fig.2.14, it receives commands on visual input in form of written digits (DBN-based hand-written digit recognition implemented in spiking neurons (Tang and Eliasmith 2010)) and draws the answers by means of arm with simulated muscles (biologically plausible hierarchical motor control). It employs multiple learning algorithms (Bekolay, Kolbeck, and Eliasmith 2013) and uses human-scale knowledge representation (Crawford, Gingerich, and Eliasmith 2013). It is capable of switching between several complex tasks, such as parsing sequentially presented commands (Stewart and Eliasmith 2013), instruction following (Choo and Eliasmith 2013), question answering. For implementing top-down-defined behavior, the architecture employs Neural Engineering Framework

(NEF) (Eliasmith and Anderson 2003), which serves as a *"neural compiler"*. For abstraction of information in the ANNs, the SPAUN uses a principle called *Semantic Pointer*, which compresses information from higher-dimensional space (lower-level representation) to the lower-dimensional space (higher-level representation). the Semantic Pointer also points the other way for the purpose of decompressing the information from higher-level representation, thus it can serve e.g. for symbol grounding. The compression of the knowledge and symbolic operations are based on circular convolution and implemented by means of Vector Symbolic Architecture (Stewart, Bekolay, and Eliasmith 2011).

It is an interesting approach for building large-scale neural-based and biologically-plausible systems, which combines both worlds: ability to build complex architectures by means of top-down design and highly noise-robust neural-based implementation. Detailed description of the architecture can be found in (Eliasmith 2013).

To conclude this chapter: there are many more cognitive architectures that could be mentioned here. The architectures may be built either as Unified Hybrid Systems (such as the Leabra and SPAUN) or Transformational Hybrid Systems (that is pluralistic, such as the SAL). But the inherent common property of such architectures is their modularity. This is caused by the fact that the complexity of behavior required from these systems is so big to be handled by a non-modular architecture. In the end, the human brain is also a modular system, so the modularity appears to be the correct approach here.

2.5 Selected Design Methods of Modular Systems

The thesis deals with some generally applicable approach for automatic (or semi-automatic) design of hybrid modular systems. The task can be decomposed into two main parts, these are: ***set of subsystems*** that can be used and ***a suitable way of combining these subsystems into a bigger system***. Selected state-of-the-art on both of these topic will be covered in this Chapter. First, the focus will be put on ensemble-like methods of designing more modular structures from some set of basic components. The second main part of this chapter will mention several selected modules (or more generally things that can be represented as a module), that are potentially useful for designing agent architectures. The rest of the knowledge will be covered in one of the

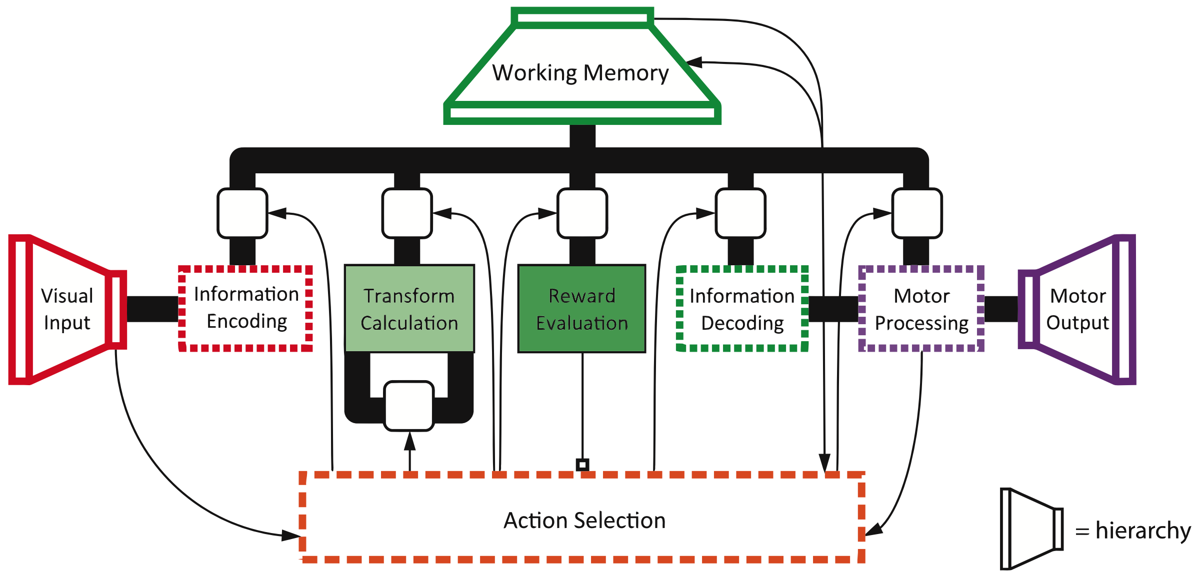


Figure 2.14: *Scheme of the SPAUN architecture. The architecture processes sequentially-presented commands in form of hand-written digits and answers by drawing its answers by means of physically modeled arm. Both the visual processing and motor control are implemented as hierarchical systems using Semantic Pointers. "Thick black lines indicate communication between elements of the cortex; thin lines indicate communication between the action-selection mechanism (basal ganglia) and the cortex. Boxes with rounded edges indicate that the action- selection mechanism can use activity changes to manipulate the flow of information into a subsystem. The open-square end of the line connecting reward evaluation and action selection denotes that this connection modulates connection weights" (Eliasmith, Stewart, et al. 2012).*

following Chapters if needed.

As mentioned before, it is often beneficial to build systems in a modular way. Such systems can be more easily designed, controlled and debugged[7]. Moreover, it is also useful to design hybrid systems. One algorithm (type of solution) usually performs well on one set of problems, but on some problems it may tend to stuck in local optima, or may not work at all. For example, heterogeneous systems that employ multiple algorithms in parallel and combine their outputs together may perform generally better than any of these algorithms alone. Another benefit is that multiple different algorithms can suggest

[7]These are called Unified Hybrid Systems in the text.

multiple good solutions[8] at a time, which increases robustness of a system and enables it to compose further new solutions. But since there is more options how to combine them (in parallel, serial, recurrent...), there arises a question: ***How to combine multiple subsystems together***? It turns out that there already exists relatively widely used theory concerning with such a question, which is called *Ensemble Methods*.

Generally, an Ensemble Method is an approach of combining results of multiple algorithms in order to obtain results superior to those that can be provided by any of the algorithms alone. Currently, these methods are commonly used in the field of Machine Learning for mostly supervised[9] learning (Opitz and Maclin 1999). Generally these hybridized machine learning systems are referred as *Multiple Classifier Systems* (MCSs), but similar key-words could be: "committee of machines" or "mixture of experts". It was shown that the bigger diversity in particular models (contained in the MCS), the better the overall performance of the ensemble (Brown et al. 2005). Also, it was shown that it may be beneficial to use relatively simple (or randomly-generated) ensembles in MCSs (Gashler, Giraud-Carrier, and Martinez 2008).

First, some basic techniques used in Ensemble Learning will be used. Then, selected Ensemble Methods (with application that is potentially useful in this thesis) will be described.

Bucket of Models is an ensemble methods which also holds multiple learners "in parallel", where *each problem (task) is solved by the model which showed the best performance on the problem.* For each problem, the training data are divided into the training and testing subset, all models are compared on this dataset and the one with the best performance is selected. From now on, the particular problem is solved by the chosen model. It was shown that the Bucket of Models has better performance while averaging on multiple problems (Džeroski and Ženko 2004).

Bootstrap Aggregating - Bagging this method uses multiple models in one layer *"used in parallel"*. The topology of Bagging could be liken to the feedforward ANN with one hidden layer. Each model receives randomly selected subset of input (training) data and suggests the solution. The overall decision is based on voting of all models, while all have equal weight. Different methods of averaging of results models can be found. This method is useful for preventing the ensemble to over-fit

[8]solution proposed by an algorithm can be called hypothesis in terms of Machine Learning
[9]but also semi-supervised or unsupervised methods can be found

the training data. A typical example of Bagging are random forests, where each tree in the forrest receives different input data.

Boosting could be named as more systematic Bagging. The method works with a set of "weak learners" (multiple inaccurate rules-of-thumb (Freund and Schapire 1997))[10] and tries to combine their hypotheses to create a "strong learner" by sequential systematic adding of new learners. There are two challenges. First: how to divide training data for particular weak learners? And second: how to combine weak hypotheses into the resulting output? While adding new learner to the ensemble, connection weights to this learner of (currently) misclassified data samples are increased. This forces new learner "to focus learning" on those currently misclassified data, and therefore to improve the overall performance of entire ensemble. Compared to Bagging, the main difference is that models are not trained (added) independently here. Also, compared to Bagging (which primarily reduces variance[11] error), the Boosting reduces bias[12] error of the ensemble's prediction. The most known algorithm implementing boosting is called Adaboost (Freund and Schapire 1997), which implements the Adaptive Resampling and Combining.

Stacking , which is also called stacked-generalization and uses an learning algorithm which combines learned predictions from other learning algorithms. This can be likened to feedforward multi-layer ANN topology. Such a technique is very successfully used for example in Deep Belief Networks (DBNs). DBN typically features a multilayer architecture, where layer is sequentially trained on data produced by the previous (already trained) layer.

One of my goals is to compose various sub-systems that are useful for autonomous agents for building more complex modular architectures. Examples of these systems are for example: planning, decision-making, action selection mechanisms etc. The Ensemble Theory can be potentially used for this particular task. The following sub-chapters will mention some particular (potentially ensemble-like) methods used for the analogous problems.

[10] Weak learner produces a hypothesis which is only slightly better than random guessing.

[11] The variance is how much the predictions for a given point vary between different realizations of the model.

[12] Bias measures how far off in general the models' predictions are from the correct value.

2.5.1 Selected Ensemble-Like and Related Approaches

This thesis should deal with autonomous building of complex modular systems with unified Input-Output (I/O) interface. The design of connections between these MIMO (Multiple-Input Multiple-Output) systems is computationally difficult problem. This chapter shows some knowledge related to design of these networks. Also, some examples of modular neural (hybrid) architectures are briefly described here.

2.5.1.1 Group of Adaptive Models Evolution

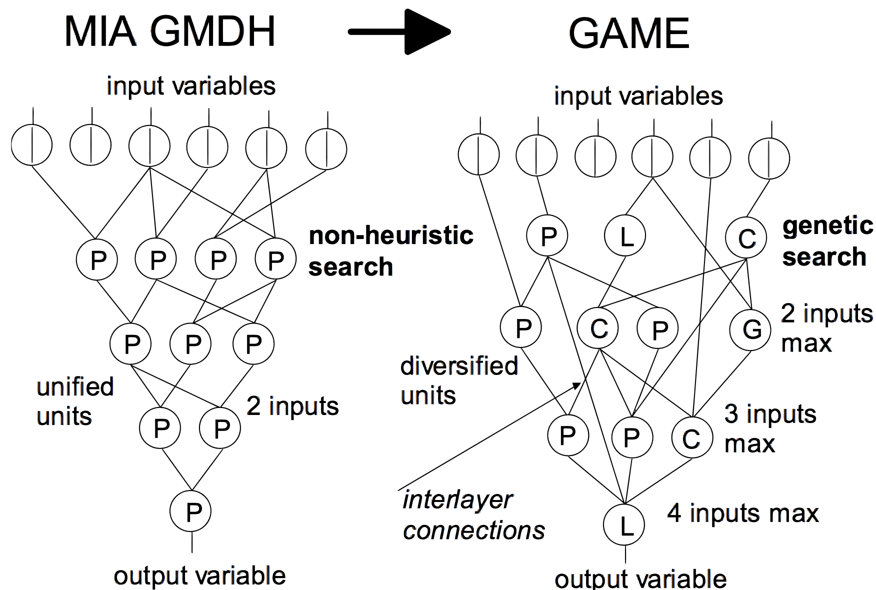


Figure 2.15: *The comparison: original MIA GMDH network and the GAME network (Kordík 2006). GAME supports interlayer connections and heterogenous nodes with more than two inputs. Various types of neurons as well as sub-networks implementing back-propagation learning are used.*

An interesting alternative to evolutionary design of ANNs is described in Kordik's dissertation thesis (Kordík 2006). This publication describes method for automated Ensemble Learning, called Fully Automated Knowledge Extraction using Group of Adaptive Models Evolution (FAKE GAME). It is based on modified Multi-Layer Algorithm for Group Method Data Handling (MIA-GMDH). This modification, called GAME automatically designs feedforward networks of heterogenous nodes used for ensemble learning. GAME builds feedforward networks of nodes and uses EA for consecutive adding of new layers

into the network. Connections of nodes in new layers are not restricted to be connected only to previous layer, but can be wired to the output of any node currently contained in the network. Comparison between MIA-GMDH and GAME produced networks is depicted in the 2.15.

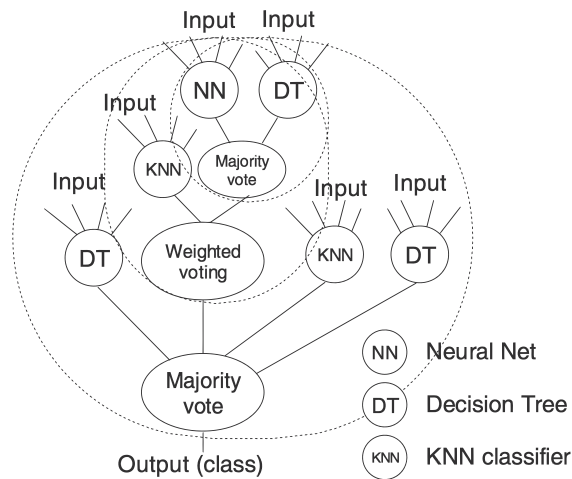


Figure 2.16: *Meta-learning: network of supervised classification algorithms (Kordík and Černý 2012). Network consists of ANN, Decision Tree (DT) and K-Nearest Neighbor (KNN) algorithm.*

An example of modular system composed of several supervised learning algorithms used for classification was presented in (Kordík and Černý 2012). The network of classifiers depicted in the fig.2.16 solves classification task for benchmark problem.

My thesis has similar goals in several ways with FAKE GAME. My hybrid networks will support MIMO subsystems, compared to these Multiple-Input Single-Output (MISO) units. Compared to this, result of my thesis will be able to design more general architectures. Also, aims of this thesis will be in generating systems that exhibit complex (also internal) *behaviour*.

2.5.1.2 Cartesian Genetic Programming

One of very interesting modifications of Evolutionary Computation (EC) is called Cartesian Genetic Programming (CGP). In (Sekanina 2010; Fišer et al. 2010) authors used CGP for autonomous design of logic circuits, which is able to meet given requirements for these circuits better than a human designer. The Figure 2.17 shows an example of one-bit adder implemented by CGP-designed network of logic gates.

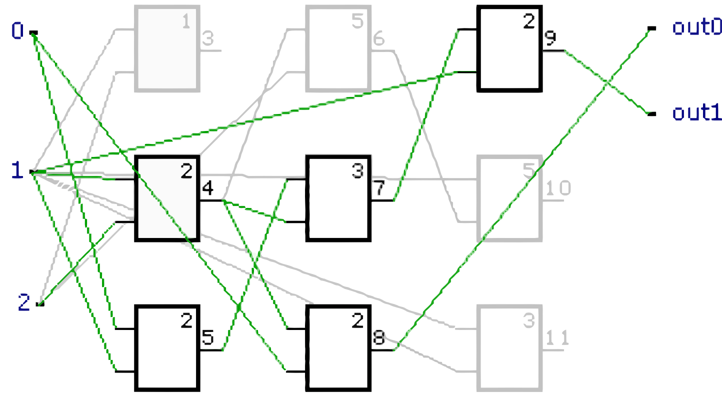


Figure 2.17: *Example of CGP-designed logic circuit implementing one-bit adder (Vašíček and Sekanina 2004). Adder is composed of four XOR gates (marked with no.2) and one AND gate (marked as 3).*

Khan et al. shown encoding shown how CGP can be modified to represent neural networks (Khan, Julian, and Halliday 2009). The publication presents The Cartesian Genetic Programming Computational Neuron (CGPCN) a method of automatic design of ANNs based on CGP and its performance on playing checkers.

Potentially interesting combination of EA and LSTM can be seen in (Schmidhuber, Wierstra, and Gomez 2005), where EA is used to design LSTM for data series prediction.

2.5.2 Ensemble Methods for Agent Architecture Design

This section will contain mainly some applications of ensemble-based design in the context of agent architectures. Here, a system will be called an agent architecture if it contains some "sensory system" and some "actuator system", which enables it to take actions - to somehow change the "state of the world". The paper (Briot, Meurisse, and Peschanski 2006) presents a direction of research similar to the one used in this thesis. Agent architectures are decomposed into its common components (e.g. obstacle avoidance, following gradient, escaping). These standardized components can be then composed into bigger architectures in the framework called **MALEVA**. Composing more simpler components together produces an architecture with more complex behavior. In the MALEVA framework, the components, which represents simpler behavior are also encapsulated as software components. A graphical description of two connected subsystems in the MAL-

EVA framework can be seen in the Fig.2.18.

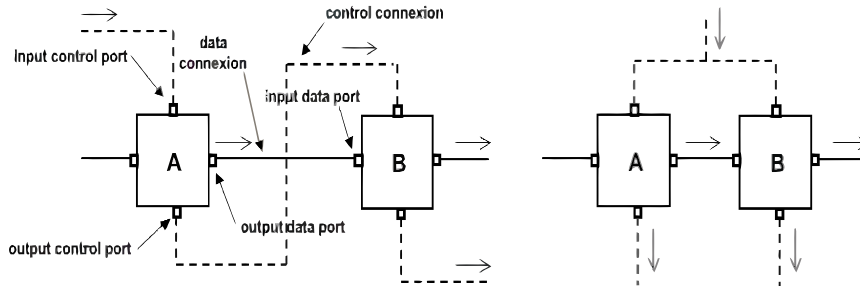


Figure 2.18: *Description of the MALEVA framework. The framework features data ports and control ports. Data ports are used to pass data, while the control ports trigger a computation in a particular component. In the scheme on the left, the A component executes the computation, sends data and then sends the control signal. Compared to this, on the scheme on the right, the data are processed concurrently (Briot, Meurisse, and Peschanski 2006).*

There are many examples of ensemble-like architecture designs which employ the Reinforcement Learning (RL). The RL tends to suffer from course of dimensionality, which is probably the reason for emerging many modular (e.g. hierarchical (Kadlecek and Nahodil 2008; Vítků 2011)) architectures, or those which directly refer to this kind of architectures as ensemble-methods (Wiering and Hasselt 2008). It was also shown that it is possible to combine multiple learner and/or planning subsystems together into one complex architecture (Zhang et al. 2012).

2.6 Selected Modules for Ensemble-Based Systems

This section will describe some selected *neural network "ensembles"*. Here, by word ensembles is meant some self-consistent component (often called ***"module"*** in the text), that can be used for example for: learning, predicting, signal generating etc. Such a module can be often used either stand-alone, or in a *network of multiple modules.* The latter approach is called ensemble method, which is described in the previous section.

2.6.1 Modules for Temporal Sequence Learning

The description of typical network structures as is feedforward, recurrent / Hopfield network will not be described here. Rather, the focus will be put on some less-known network structures which could be reused in modular systems together with classical feedforward networks with back-propagation-based learning. Good overview of most used ANN architectures and corresponding learning rules is in (Wilamowsky 2003).

It is assumed that there are two crucial characteristic features of human brain which make it so special in real-world "applications":

Pattern Recognition Probably the most important feature of neural networks is their ability to learn and recognize patterns. Their ability of associative learning and robust recognition of similar objects/situations still has not been fully replicated in artificial systems.

Sequence Recognition *The ability to understand one's environment, essential for intelligence, is not static. The order in which events occur can be even more important than the events themselves, and an intelligent system, whether it be a frog, a robot, or a human, must be able to detect this ordering and to reproduce this ordering on some cue (Wang and Arbib 1990).*

In a simplified model of brain, each of these qualities could be implemented by one module, or sub-network. Then we can **combine these modules** in e.g. hierarchical structure. The resulting system should recognize sequences of patterns, patterns of sequences of patterns and so on..

This thesis focuses on investigation of possibilities of interconnection of various modules together and benefits of particular structures. This is the reason why rather small sub-networks with capabilities that are potentially useful for designing artificial agents will be described here.

The following sub-sections will describe some types of neural circuits that are able to recognize sequences. Temporal sequence is composed of components, which are alternatively called spatial patterns or symbols.

2.6.1.1 Time Delay Neural Networks

Time Delay Neural Networks (TDNNS) can recognize and reproduce time sequences and can be used for temporal association, that is to produce particular output sequence in response to a specific input sequence. The main idea is in use of floating window over the input temporal sequences. This modification of feedforward network is the simplest approach to learn sequences. One benefit is that the conventional back-propagation algorithms can be used for learning. The main downside is in limited length of delay line, where for larger delays more neurons are needed, see Fig.2.19.

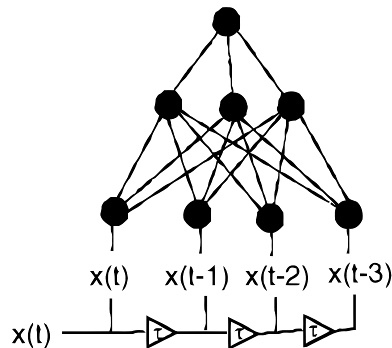


Figure 2.19: *Time Delay Neural Network with uniform delay. Triangle represents unit delay of input signal. This sampled signal is fed into a typical feedforward network structure. Network can learn e.g. to predict the following input.*

There are several improvements of this approach, as is allowing of non-uniform sampling, according to the equation 2.1. This improvement introduces ω_i, which denotes waiting time before input neuron i. Equation 2.1 describes how the input values are processed, n is a number of input neurons. In this approach, the memory is not limited only by the n previous samples.

$$\tilde{x}_i(t) = x(t - \omega_i) \tag{2.1}$$

2.6.1.2 Short-Term Memory

Short-Term Memory (STM) is based on use of so called dual-neurons (Wang and Arbib 1990). In each dual neuron there is one input, one output and a recurrent connection between neurons which temporally stores information received on the input, strength of

this information decays with time, see Fig.2.20. This system is able not only to remember stored information, but also is able to determine *how old* the information is.

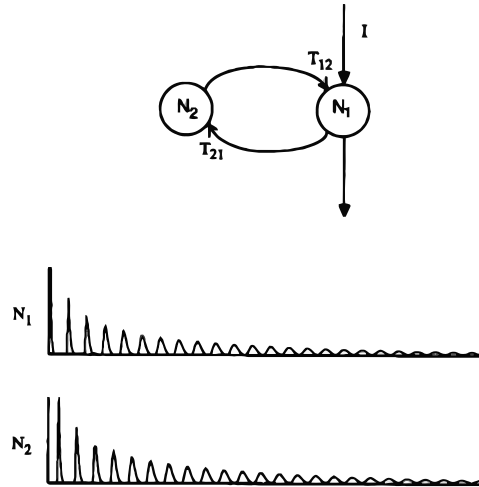


Figure 2.20: *Dual neuron and it's response which maintains a signal for a certain memory span.*

2.6.1.3 Long-Term Memory

Long Term Memory (LTM) is similar type of learning as a STM, but compared to STM the LTM is able to learn longer sequences and to store them for a longer period of time. LTM is designed for spatio-temporal learning and recognition and is inspired by the longterm memory model of the human cortex. The LTM presented in (Nguyen, Starzyk, Wooi-Boon, et al. 2012) is able to process real-valued and multidimensional sequences.

The presented network works similarly to classical algorithms for sequence recognition. Input sequence is passed to the inputs of structure and one output indicates whether the sequence matches the stored one - the memory.

The LTM network uses sparsely-connected nodes organized in four layers, see Fig.2.21. Layers are named as: input layer, the primary layer, the intermediate layer, and the secondary layer. The primary layer consists of primary neurons, depicted as R. The content of a training sequence is stored as the synaptic weights between input and primary layers. The role of the primary layer is to compute the degree of similarity between an input vector and components of the stored sequence (Nguyen, Starzyk, Wooi-Boon, et al. 2012). The two upper layers then are used to indicate how well the input vector matches the stored sequence. The matching is given by the sum of all correct values on inputs.

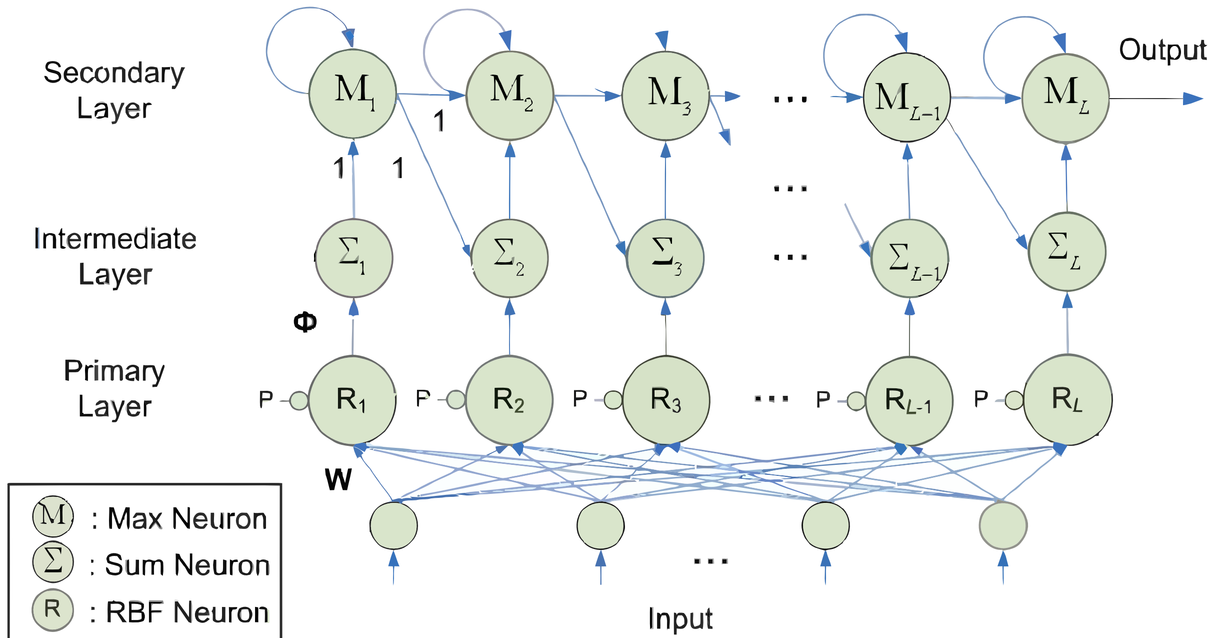


Figure 2.21: *Structure of model of Long-Term Memory for recognition of temporal sequences. One LTM module has L inputs and one output describing how well the input sequence matches to the stored one. Sequence is stored in connection weights between input and primary layer of neurons.*

2.6.1.4 Categorization and Learning Module

Other potentially useful approach for composing modular neural systems is called Categorizing and Learning Module (CALM) developed by J. Murre and published in (Murre, Phaf, and Wolters 1989). It is a stand-alone module capable of sequential learning, which can be embedded into more complicated system, e.g. into a hierarchy. Its structure is inspired in cortical mini-column and is composed of neurons with real-valued continuous activation function. Its structure is depicted in the Fig.2.22.

Connections inside of CALM are static, only inter-module weights are modified by a learning algorithm. V-nodes and R-nodes form matched pairs - each R-node excites only one V-node. Each V-node produce much stronger inhibitory signal back to non-corresponding R-nodes. Therefore, if the V-node wins the competition, all other R-nodes are inhibited and only one pair of nodes remains active. A-node (arousal node) integrates excitation and inhibition from other nodes in the module. If only one R-V pair is active, A node does not produce signal. In other situations A-node injects noise back into the network and stimulates competition between particular node pairs.

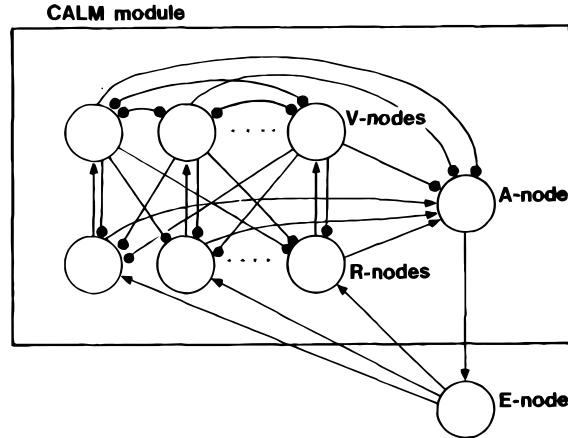


Figure 2.22: *Categorizing and Learning Module. Solid circles indicate inhibitory, arrows excitatory connections (Murre, Phaf, and Wolters 1989). R denotes Representation nodes, V denotes Veto (inhibitory) nodes.*

Later, the modification of CALM so that it can learn sequences, was shown and named CALM2 (Koutník and Šnorek 2004). Further, the CALM capable learning sequences was improved by A. G. Tijsseling, who added ability to adapt the particular module size to the complexity of given problem (Tijsseling 2005).

2.6.1.5 Long Short-Term memory

Long Short-Term Memory combines abilities of both LTM and STM, is composed of more complicated and less biologically accurate modules with some possibilities of their control. LSTM is a good example of system that is on a boundary between artificial neural networks and top-down designed module, which has inputs,outputs and rigid inner structure.

2.6.2 Signal Generators

In contrast to temporal sequence recognition, ANNs are also successfully used for generating the signal. Recurrent ANNs (RNNs) can be used for generating various types of signals, from chaotic to periodic ones. Signal generators often belong to group called reservoir computation, where an input signal is fed into a (randomly generated) dynamical system (reservoir). The dynamics of the reservoir map the input to a higher

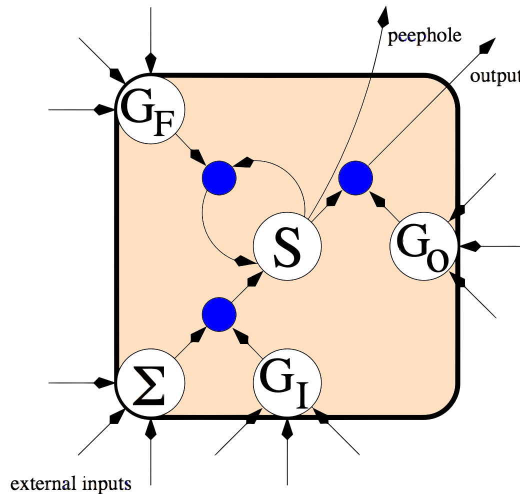


Figure 2.23: *Long Short-Term Memory Diagram. The cell has an internal state S together with a forget gate (G_F) that determines how much the state is attenuated at each time step. The input gate (G_I) controls access to the cell by the external inputs that are summed into the $\sum$ unit, and the output gate (G_O) controls when and how much the cell fires. Small dark nodes represent the multiplication function (Schmidhuber, Wierstra, and Gomez 2005).*

dimension.After that, a simple readout mechanism is trained to read the state of the reservoir and transforms it to the desired output signal. This section will briefly describe two interesting types of neural-based signal generators.

2.6.2.1 Echo State Network

The first type of signal generators is called Echo State Networks (ESN) - a good example of reservoir computation (Jaeger and Haas 2004). These networks can be used as signal generators tunable by input signal. ESNs are composed of neural ensembles containing randomly and recurrently interconnected neurons. There is predefined set of inputs neurons, but no particular outputs are defined a priori. Due to recurrent connections between neurons, ESN exhibits chaotic behavior as a response to input signals. The desired output signal (as a response to input values) is obtained by tuning weights of connections from hidden neurons to output ones. Figure 2.24 depicts the principle of function of ESN.

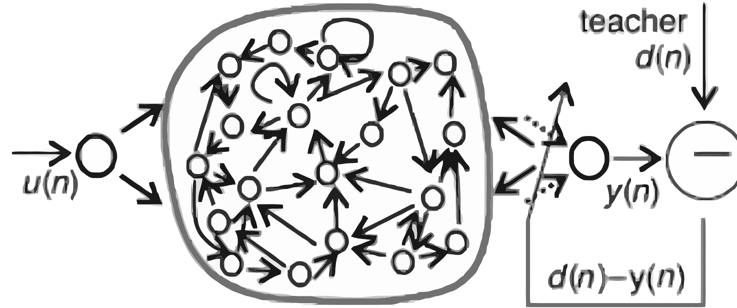


Figure 2.24: *The basic schema of an ESN. Weights between hidden and output neurons are optimized so that the desired output signal is found (Jaeger and Haas 2004).*

2.6.2.2 Central Pattern Generator

Central Pattern Generator (CPG) has the similar structure as ESN networks do. CPGs also contain recurrently connected neurons, but instead of randomly connected ensembles of neurons, CPG ensembles are (in case of artificial system) manually designed to produce desired periodical signal.

This approach is inspired in wiring of neurons in animal's spinal cord. In most animals, many muscles are not controlled directly by the brain. Brain sends higher-level commands to spinal cord, which is able to produce predefined patterns of behavior and send these patterns to particular muscles. This predetermines use of CPGs mainly for controlling the gait in legged robots. As mentioned in (Pei et al. 2012): *There are three common motion control methods: model-based method, behavior-based method and biocybernetics method. Model-based method is difficult to establish dynamic model and has poor real-time capability and environmental adaptability; behavior-based method is mainly used for insect intelligence bionic; biocybernetics method realizes the robot's motion using rhythmic movement of animal.* The CPGs can be conveniently used to implement the third method mentioned.

Oscillatory behavior between neurons can be generated in two ways: through the interaction between neurons (network based) or by means of interactions among currents in individual neurons. The basic structure that can produce rhythmic behavior is called Half-Center Oscillator (HCO), which consists of two wired neurons. The scheme of HCO which generates sinusoid signal is in the fig 2.25.

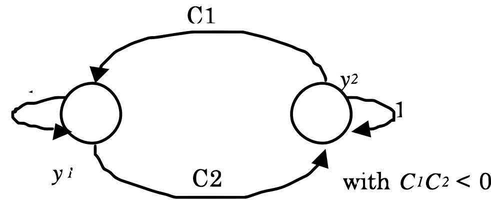


Figure 2.25: *Generator of sinusoidal signal (Zainer and Nagashima 2002). Each neuron has recurrent loop of weight 1. One connection C is positive, the other negative. As the signal goes through positive connection, time delay of neuron and negative connection, the sinusoid signal is produced on outputs of both neurons.*

Another example of simple CPG which generates sinusoidal signal is in the Fig.2.25, this principle is similar to dual neurons. Other various sub-networks which are able to produce rhythmic activity are mentioned in (Matsuoka 1985), example of quadratic polynomial generator polynomial composed of two neurons is in (Zainer and Nagashima 2002).

Interesting application of CPG is for example in (Ijspeert et al. 2007). Here, the researchers modeled wiring of neurons in a spinal cord of salamander. This animal is not capable of arbitrary movement, neurons in his spinal cord are wired in the manner which automatically produces predefined behavior patterns. The scheme of spinal cord of salamander is depicted in the Fig.2.26.

Another interesting application of CPGs ability to produce periodical signals is in (Zainer and Nagashima 2002). They developed language for designing ANN topologies which exploit CPGs. This special language was then used for composition of ANN which controls the walking of humanoid robot called Humanoid for Open Architecture Platform (HOAP). Central pattern generators were used here for generating periodical signals for robot joints which produce smooth walking. The scheme depicting a part of ANN used for generating the motor commands is depicted in the Fig.2.27. Neural network is composed of neurons, dead neurons (neurons without time delay - marked with small circle), thresholds (triangle) and switches (triangle with switch input). Compared to model-based approaches, this controller was assembled with only several lines of code.

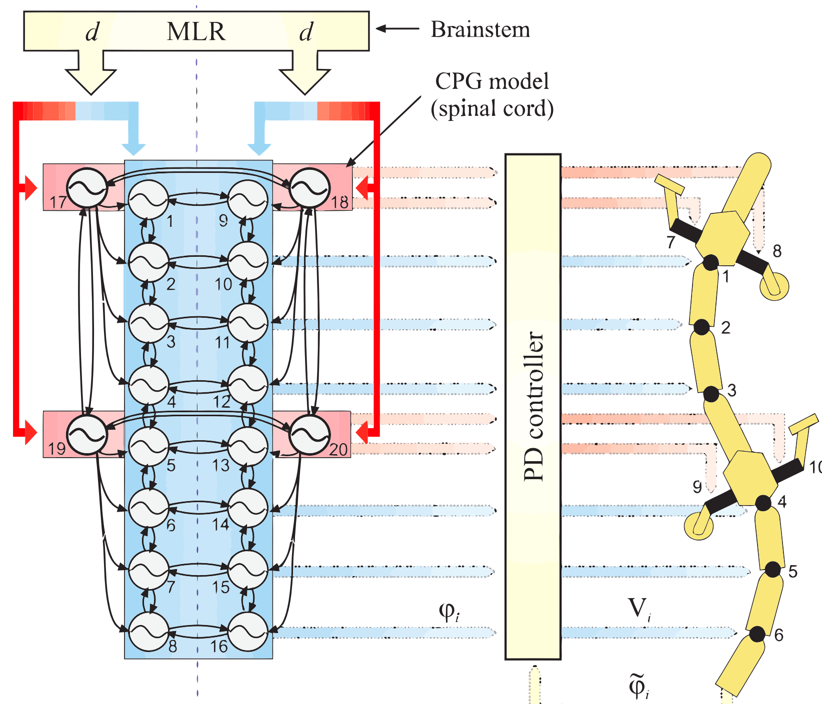


Figure 2.26: *Model of spinal cord for robotic salamander. Salamander is not capable of arbitrary movement, neurons in his spinal cord are wired in the manner which automatically produces predefined behavior patterns. Model is composed of series of CPGs which produce given signal to control the salamander movement (Ijspeert et al. 2007).*

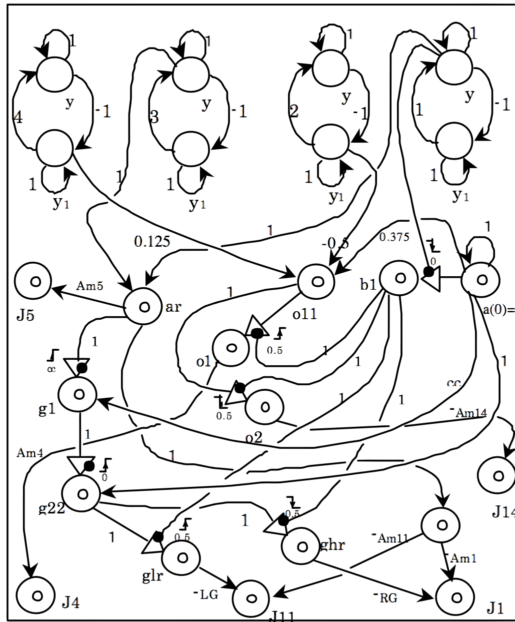


Figure 2.27: *Part of the Central Pattern Generator-based network which controls walking of humanoid robot HOAP (Zainer and Nagashima 2002). Nodes marked J_i represent actuators on joints of robot. We can see that on the top of the network, there are four generators of sinusoid signal, each with different properties.*

Chapter 3

Problem Analysis and Proposed Solution

The main disadvantage of *ANNs* is obvious. Despite their robustness, these often black-box-based solutions are hard to repair or modify. There are ways how to engineer big modular ANNs, but these large-scale networks have often too big computational requirements (e.g. as (Eliasmith, Stewart, et al. 2012)).

Compared to this, more *classical AI* (that is: top-down approach or explicitly engineered approach general) is able to provide solutions to very complex problems, including deliberative behavior originated from symbolic operations (e.g. playing chess, design of controller etc..). The problem here is of course that these solutions are often able to solve very constrained part of the problem, but are not robust enough to work when something goes wrong. Also, in most cases: the more advanced and complicated top-down-designed system, the more domain specific is.

Main goals of this thesis are to bridge over these common design problems by proposing the framework that aims to fulfill the following main goals:

- to provide unified approach for designing the architectures
- to build reusable (and as domain independent as possible) parts of the architectures
- to simplify design and prototyping of new architectures
- to enable novel uses of current algorithms (e.g. new combinations with other algorithms/sub-systems)

- to combine benefits of both design approaches: top-down and bottom-up
- to provide the possibility of automatic design of architectures for given task.

The following Section will describe main challenges during designing of such a framework. The second Section will describe the type of problems that will be the proposed approach tested on. The last Section will describe the proposed solution to the problems stated earlier.

3.1 Problem Analysis

In the ideal case, such a framework should be able to combine various sub-systems together into one bigger hybrid architecture (for example path-planning, vision systems, voice recognition, neural networks, robotic HW etc.). There are many levels of challenges on the path to the solution, these are mainly: theoretical, issues with compatibility, implementation specific, practical etc. Those main will be described in this Section. The Section 3.3 will address all of these challenges.

3.1.1 Different Types of Communication

The requirement of connecting various systems has one main obvious problem. The first question is: *how to obtain method of connect sub-systems together, that is general enough?* First, in the ideal case, it would be very suitable to have an unified representation of information in the entire system. It turns out that this requirement is very strong and hard to fulfill. There are two main problems here: some sub-systems may employ continuous time representation, some other may require discrete time steps.

The second problem is in the form of representation of the information, that needs to be passed between sub-systems. There are three main types of information that were considered, passing: explicit symbolic representation, real-valued numbers and spiking communication. For instance, it is essential that almost all planning systems use the symbolic representation. Compared to this, the output of a typical ANN is in form of vectors real-valued numbers. Moreover, SNNs may produce series of spikes, rather than

anything else.

The encoding of information in the human brain is not known so far. There are several types of encoding of information in SNNs that are used currently (Ponulak and Kasinski 2011). In the simplest case (not fast-changing information) the rate-code can be used, where the average firing rate corresponds to the real-valued number represented by the neuron (or the population of neurons).

More common problem is in converting the real-valued data into symbolic representation. This is known as a *symbol grounding problem*, where the mapping between continuous values and the symbolic representation needs to be defined. Currently, tare several different ways how to tackle this problem. As the most intuitive one: the mapping between symbolic and sub-symbolic[1] can be defined manually, by use of domain knowledge. This mapping can be also learned (mined) from data, ideally online during the operation of the system. Several examples of coupling the ANN with system employing symbolic representation can be found in (Mcgarry, Wermter, and Macintyre 1999). Probably more promising solution is to represent symbols together with the symbolic operations directly the real-valued data, as shown in (Eliasmith and Anderson 2003). Systems employing this approach then do not need an explicit conversion from sub-symbolic to symbolic representations, but this approach is not widely used so far.

3.1.2 Theoretical Issues with Automatic Design of Modular Systems

Completely another challenge then lies in the number of possibilities of combinations of the sub-systems together, into one bigger system. Typically, a representation of complex system composed of multiple sub-systems would not be a problem, *but searching in space of possible systems* is a big issue.

For example on the ANNs, the neurons have one output and one input (because all weighted input values are accumulated together). Searching for correct weighted connections in classical ANNs of general (that is fully connected topology) is a big challenge, since the number of connection weights grows exponentially with number of neurons in the network. This problem is tackled by two main approaches. First is in constraining

[1]Sub-symbolic representation is here referred to as real-valued one.

the connections in the network, for example to only layered feed-forward topologies. The second is in modularization of ANN into smaller sub-networks (Auda and Kamel 1999).

But the goal is to represent the following: *sub-systems of different types* and *sub-systems with multiple different inputs* and possibly also *sub-systems with multiple different outputs*. This means that one resulting system should contain multiple types of Multiple-Input Multiple-Output (MIMO) systems combined together. As mentioned in (Kordík 2006), the searched space of such a problem grows mind-blowingly fast. Also, note that the Kordík's approach works only with Multiple-Input Single-Output (MISO) sub-systems.

3.1.3 Reusability of Sub-systems and Domain Dependency

Another challenge is in requirement of seamless representation of different sub-systems. For example, it is not suitable to represent one neuron model in the same way as the planning system. Typically, many of more complicated sub-systems will require either domain configuration or "definition of the goals" - what to do (that is: what to learn, or search for) in a bigger system. In some research-fields (e.g. vision) it was shown that domain independent (or/and biologically inspired) solutions exhibit comparable performance to those domain-specific. Despite this fact, the many of domain-independent algorithms have to be at least tuned for given task. That is, setting at least a few parameters may be necessary for each sub-system.

3.1.4 Implementation and Practical Issues

Lats, but not least, there are practical issues of such a framework. It should be capable of integrating of multiple sub-systems together. During designing and implementation of some complicated system, is often suitable to use some existing solution/implementation. This means that various sub-systems may be implemented in different programming languages. Moreover, some system may require to be run on a specific machine with a specific HW (such as for example GPU-accelerated SNNs (Fidjeland, Roesch, et al. 2009)). Often, a single computer may not be powerful enough to run entire simulation, so some kind of de-centralized solution may be required.

Recapitulation This Section described main challenges on a way of defining a frame-

work, that is capable of representing an combining as general sub-systems as possible together into bigger modular systems. There is many problems, starting from the representation of communication, problems with domain dependency and configuration of particular sub-systems. Also, the practical issues need to be taken into account during designing the framework.

3.2 Task Description

The proposed framework should be held as general as possible, and therefore should be applicable also on tasks outside this scope, but still there are some inherited and common properties of tasks the author intend to use the framework. This Section will describe main properties of domain, where the approach will be tested.

The thesis focuses on the agent/cognitive architectures in general. Common property of such systems is their *operation in the closed loop with the world*, or some simulated environment. Typically, it is expected that such a system learns either from experience by interaction with the environment and/or by means of unsupervised learning. So there is no requirement for supervised learning from input-output examples. Rather, these systems will work from incoming continuous streams of data and produce some actions on outputs.

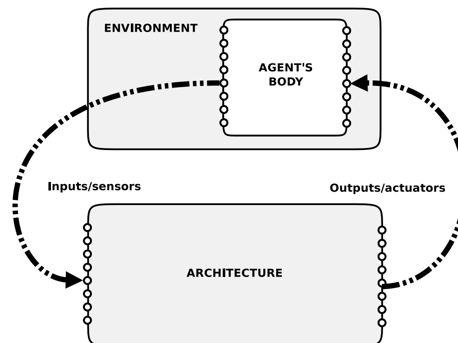


Figure 3.1: *Basic type of use of an agent architecture. The system has inputs and outputs connected in a closed-loop with the agent's body which is situated in an environment (either simulated or real).*

Typical setup of an architecture can be seen in the Fig.3.1. The overall system is therefore designed for the known inputs and outputs, which can be used for constraining the space

of searched architectures. The agent architecture should be able to successfully perform some meaningful task in the environment. The particular task is defined during the experiment setup, but in general, an agent architecture should be able to:

Sense the external world. Which includes mainly sensory data pre-processing, such as for example data fusion, dimensionality reduction etc...

Represent the knowledge about world, which includes mainly modeling of the world. For example learned symbols, relations between symbols etc.

Represent/obtain goals and be able to meet them. Examples of some potential subsystems with the following functionality:

- generating internal/external goals,
- reasoning/planning in the model of the world,
- policy (plan, strategy) creation.

Policy execution module, which implements some action selection (what will be placed to actuators) and possibly some policy persistence (as mentioned in the Section 4.3.1), representing agent's intentions.

Here will be a bit better description of typical sources of agent's goals in the domain of ALife. Each agent should have some goals, goals can be either binary (fulfilled/not fulfilled) or can define some optimal conditions. Then, the current conditions of agent are continuously optimized in order to get close to those optimal conditions. Both types of goals can be defined *from the outside* (as some explicitly defined mission passed to the agent's sensors) or *from the inside*, defined as some agent's need.

Recapitulation This chapter described a typical use of agent architecture in the ALife domain. This characterizes the target use of the architectures that the author intends to generalize. The proposed framework will be shown on examples with similar properties as described in this Section, but the proposed framework should not be constrained only to this specification.

3.3 Proposed Solution

This Section will describe the original solution proposed by the author. First, the motivation for this framework will be written, then the framework itself will be described in more detail.

The main and original idea of this thesis is to attempt ***to unify representation of current*** (computer science-oriented) ***knowledge in a compatible way***. Currently, the knowledge about new algorithms is currently passed through the description of the algorithm in a scientific paper. Usually, the reader needs to understand the information in the paper correctly, re-implement the algorithm, debug it and test it on the data. This is highly inefficient way. A better solution is to share directly also implementations of new algorithms. Even better solution is ***to provide common platform*** for sharing new algorithms. In case that algorithms have unified communication interface, these can be directly used for another research, for example composing new modular systems.

Therefore one of the main contributions of this thesis lies in the definition of the framework that enables combination of sub-systems of different nature together in bigger modular architectures. These architectures can be hand-designed from existing sub-systems. Furthermore, the framework enables automatic design of new modular architectures specifically for a given task, which could potentially lead to automatic discovering of entirely new modular systems.

As can be seen in the following Sections, the framework is inspired in Modular Neural Networks (MNNs), therefore *each sub-system is called* ***Neural Module*** here. Particularly, the Neural Module represents an enclosed sub-system in the framework that is able to operate as a stand-alone unit, it receives data, implements a given computation and sends new data further on outputs. Since the framework is mentioned to employ heterogeneous sub-systems, it was called ***Hybrid Artificial Neural Network Systems (HANNS)***. The following text will describe the proposed framework while addressing problems stated in the Problem Analysis (Section 3.1).

3.3.1 Unified Type of Communication Between Sub-systems

First, the proposed framework deals with different representations of information for different Neural Modules. Typical hybrid systems need to explicitly deal with conversion

of information, usually between symbolic and sub-symbolic domain. In order to create a general approach enough, the *HANNS does not allow explicit transformation between information representation*. Instead, one communication protocol was selected to be used across the entire system. This allows arbitrary combination of different Neural Modules together.

The framework is inspired in the ANNs, so that the *communication is realized through (vectors of) real-valued numbers* passed between particular Neural Modules. Since there is no (currently discovered) optimal common type of communication, this common representation was chosen because the both other types of communication described in the Section 3.1.1 can be relatively easily converted to this one.

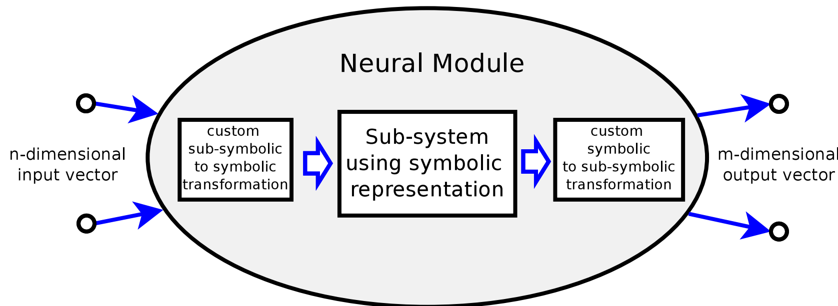


Figure 3.2: *Principle of communication between Neural Modules and translation of information. Each Neural Module (algorithm) is allowed to use an arbitrary type of inner information representation. The only requirement is that the data passed to the Module and sent by the Module have predefined format - vectors of real-valued numbers. The example shows a sub-system using symbolic representation.*

Note that the *HANNS framework does not define how the information is translated* from one type of communication to another. Instead, it is a responsibility of Neural Module to implement own transformation of input and output information. The disadvantage is that this poses some constraints on use of such a module, there are the following possibilities in general:

The transformations are predefined. Some of these definitions may be domain specific (for example some symbolic representations), but there are cases that this does not hold. As example of domain independent transformation can be mentioned the rate-code in SNNs.

The transformations are learned from data. For example, the planning sub-

system can learn symbolic representation from received data online, during the simulation. In this case, the inner representation of information in the sub-system changes during the simulation, so that the sub-system may not operate optimally, or with human-readable data. For example, a Neural Module using symbolic representation and implementing planning engine was tested as a part of one Bachelor Thesis (Skála 2013).

In some cases, such a requirement of one communication interface can be too strict, but it enables the framework to combine the sub-systems in almost arbitrary ways. Furthermore, a more practical side of the framework is described in the Section 3.4.2.

Note that the communication in the HANNS is event-driven: if the Neural Module receives data, it processes them and the result of computation sends on own outputs.

3.3.1.1 Reusability of Sub-systems and Domain Dependency

The framework suggests that as domain independent as possible Neural Modules are used. But even this cannot be done some cases. Aside of *data input/outputs*, a Neural Module has also *configuration inputs*. These inputs have the same form as the data inputs, but their usage is optional. The configuration inputs define values of configuration parameters of the sub-system (algorithm) encapsulated in the Neural Module (such as learning rate, forgetting rate etc). If the configuration input is not connected, the default value of parameter is used.

Since the configuration inputs are represented seamlessly as the data inputs, the configuration of Neural Module can be changed also during the simulation. This can be done by connecting it to some source of signal (real-valued data). More on reusability of Neural Modules will be described in the Section 3.4.2.

Also, many algorithms can operate with data of different dimensionality (e.g. k-means algorithm etc) and are able to provide data of various dimensionality. It is not possible to change the number of inputs/outputs during the simulation of the system. Therefore these properties need to be configured before the architecture is simulated. The framework can also employ simple linear the constraints on the input/output dimensionality of Neural Modules. These can be written in form of a simple equations. For example, for the neuron model, it would be: $no_{in} = 1$; $no_{out} = 1$. Or for example for the Principal-Component Analysis algorithm (PCA) it would be: $out_{dims} \leq in_{dims}$. This enables

3.3.2 Design of Modular Architectures in the Framework

This section will describe main possibilities of designing new architectures in the HANNS framework, these are mainly: creating hand-designed architectures and automatic design of new architectures specifically for a given task.

Figure 3.3: *Principle of design of new architectures based on a particular task. The Designer is either the user (human designing the architecture) or some kind of optimization algorithm. Based on a given task, set of Neural Modules is picked from the library and placed in the architecture. Then, the connection weights between Neural Modules and architecture inputs/outputs are optimized in order to gain the desired behavior of the architecture.*

3.3.2.1 Weighting Between the Top-Down and Bottom-Up Design

One of the main benefits of the framework is in the ability to weight between these two quite opposite design approaches. The network-like part - connections between particular Neural Modules are represented as in case of classical ANNs. This means that the ***HANNS uses weighted connections*** to pass the information between Neural Modules in the network.

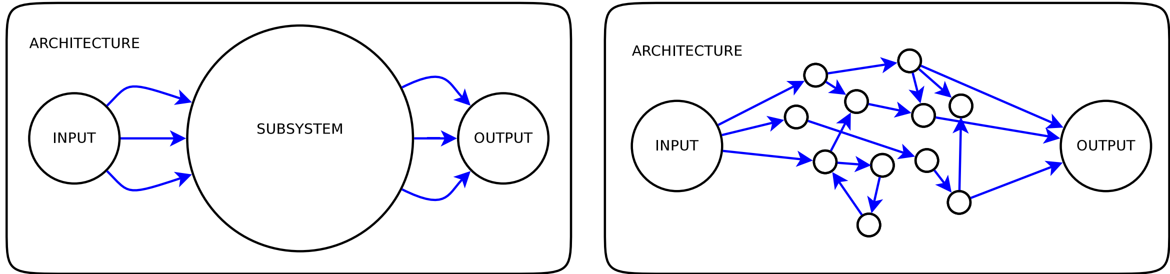
This means that it is possible to exploit both kinds of design at once. Here, the *reusable Neural Modules represent top-down* engineered approach (e.g. implementation of RL, k-means algorithm etc). While the *intelligent behavior emerging from interaction* of provided components represent *bottom-up* approach.

The main benefit of this approach is in the fact that the *amount of explicit engineering* in resulting generated architectures *can be arbitrarily chosen.* Because top-down approach is represented by particular subsystems, used can define how big part of solution wants to design by hand. Again, there can occur two extreme cases of designed system:

- Subsystem designed by hand solves the entire problem, the used only connects inputs and outputs. An example of such almost entirely top-down-designed system

is depicted in the Fig.3.4(a).

- In the opposite extreme, the resulting architecture is composed from the smallest elementary subsystems (probably neurons) and the used is supposed to connect everything by hand, that is: to design entire system "from the scratch". Example of such a completely bottom-up designed architecture is in the Fig.3.4(b).



(a) *Scheme of a completely engineered system, the user provided complete solution by the subsystem. Architect just connected inputs and outputs.*

(b) *Scheme of a system generated from the elementary parts, no explicit design of structure was involved.*

Figure 3.4: *Comparison of top-down and bottom-up-designed hybrid systems.*

It is important to note that systems in the figures 3.4(a) and 3.4(b) can be identical (that is, the engineered system has the same inner structure as the automatically-designed one). The only difference is that the engineered system is encapsulated in a Neural Module and therefore has a predefined un-changeable functionality. But in the second case, the same part of the system can be changed/optimized for a particular task.

3.3.2.2 Evaluating the Suitability of Sub-Systems in the Network

The modularization of the Hybrid Neural Network has also the benefit that the user can have better insight into the architecture's inner processes. Also, the behavior of entire architecture can be better altered/optimized. But it is difficult to measure how well the agent performs in the environment. The proposed solution should also provide little better insight into the information: how the particular Neural Modules are used in the system.

One of the main aims of this framework is to study new, alternative use-cases of known algorithms/subsystems. During the automatic design of these Hybrid Artificial Neural Network Systems, the following use cases of Neural Module can occur:

- Neural Module is connected in a ***completely wrong way***: corresponding algorithm is used inefficiently or it does not work at all.
- Neural Module is connected in an ***unexpected, new way***: corresponding algorithm is employed, but in a way not anticipated by the user, potentially new use of the algorithm.
- Neural Module is connected in an ***expected way***: the algorithm is employed as expected during the Neural Module design.

The second mentioned case does not necessarily mean that the algorithm is not advantageous in the architecture. However, the behavior of an algorithm (and potentially its purpose in the architecture) can be often hard to analyze by hand.

In order to identify incorrectly used parts of resulting architecture, it would be convenient to distinguish between these three use-cases automatically. Furthermore it would be useful to be able to evaluate the algorithms performance in a given situation. However, this is possible only by means of heuristics. The author introduces the function called ***Prosperity***. This newly introduced Prosperity output of the Neural Module defines ***subjective heuristics*** defining ***"how well the algorithm performs"*** in a given architecture during the simulation. This enables user (and potentially EA) to distinguish between good and bad parts of a particular architecture. It is up to designer of particular Neural Module how to define its Prosperity function. The function should produce values in the interval $\langle 0, 1\rangle$. For example, the subjective Prosperity of k-means algorithm could be computed as average distance of data sample to the nearest center of the cluster, that is: *how well is the data represented by the algorithm* during the simulation.

These values of Prosperity can be then employed for inspecting the suitability of use of particular Modules in the system. Furthermore, these can be used for evaluating the quality of entire system during automatic design of architectures (see the text below).

3.3.3 Automatic Design of Architectures Specific for the Task

Often, there are several research fields that try to provide the solutions to the same problems. This means: many of research fields overlap at least partially, and therefore provide own solutions to the same problems. Also, none of research fields is (and will not be) able to provide solution to all of the problems. Rather, probably the best current

method to designing new solutions to problems is the following. Based on particular task, we *estimate how the solution could look like, pick knowledge from several fields of science and combine them* together. One of the main aims of the HANNS framework is to provide methods for at least partial automatization of such a process.

This Chapter will describe the possibilities of automatic design of architectures that are suitable just for a given task. As described so far, the HANNS framework represents all sub-systems (small solutions from different fields of research) as different Neural Modules - standalone blocks with MIMO connections of common type.

The Fig.3.3 depicts the principle of designing of new architectures. Based on a particular task, a set of Neural Modules is picked from the library and the connection weights between these modules (and input/output connections of the architecture (see the Fig.3.1)) are optimized in order to gain the desired behavior. The following Sections will describe how the proposed solution tackles the problems stated in the previous Sections.

3.3.3.1 Dimensionality Reduction?

There are two sides of automatized design in the proposed HANNS framework. First, by encapsulating a particular sub-systems (which could be implemented for example by small ANN) greatly decreases the complexity of the overall topology that required to obtain desired behavior. This provides the possibility to employ the same topology optimization algorithm to create systems with superior overall complexity, than it would be possible by means of only ANN-based solution.

On the other hand, as described in the Section 3.1.2, the size of searched space of all possible architectures still grows exponentially with:

- number of Neural Modules available,
- number of input connections for a Module
- number of output connections for a Module
- number of input/output connections of the architecture.

The author proposes the following constraints on possible architectures, in order to obtain the system that is able to design new architectures in reasonable time.

3.3.3.2 Constraining the Set of Available Modules

First, the automatic design does not include selection of Neural Modules. The set of Neural Modules used as a starting point for the topology optimization is defined at the beginning. On one side, this is a bit simplification of the approach. But typically, some of the problems can be solved by some set of Neural Modules. This, at least partial domain knowledge can be used to very efficient pruning of space of all possible topologies. The user therefore does not need to know everything about the task, but often *some knowledge about underlining principles of the problem is known*. For example it can be known that some dimensionality reduction of input data can be required. Therefore the user picks for example Neural Modules implementing some clustering algorithms. The principle is shown in the Fig.3.5.

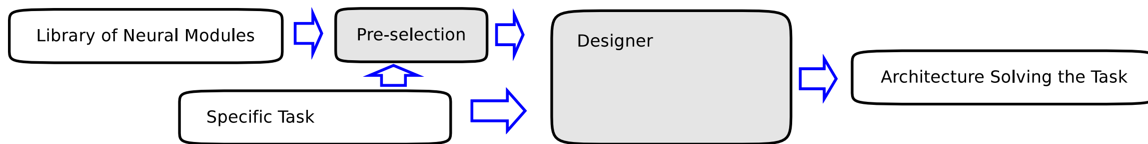


Figure 3.5: *Pre-selection of Neural Modules to be used in the architecture. Currently, the pre-selection is defined by the user using approximate domain knowledge.*

By this way, the user can significantly reduce the size of space of architectures searched by the optimization algorithm, while also "suggesting" the approximate type of the architecture. There are two main possibilities here. First, to filter Neural Modules by their purpose, the purpose can be represented e.g. by Modules' keywords (such as for example "clustering", "learning", "policy generation", "action selection", "random behavior"...).

Here, the second bigger constrain is used. The user defines the set of Neural Module's instances, that will be used in the architecture. That is: how many and what Modules will be used during the optimization.

3.3.3.3 Constraining the Space of Available Topologies

Now we have predefined set of Neural Modules in the architecture. Still, the number of all possible connection weights between these Modules is too big.

It is possible that there are some useful architectures that need to be designed as fully connected networks (for example some hybrid version of Echo State Networks). But it is more likely that useful architectures will have rather feedforward nature. Therefore the constraint is also put on the architecture topology.

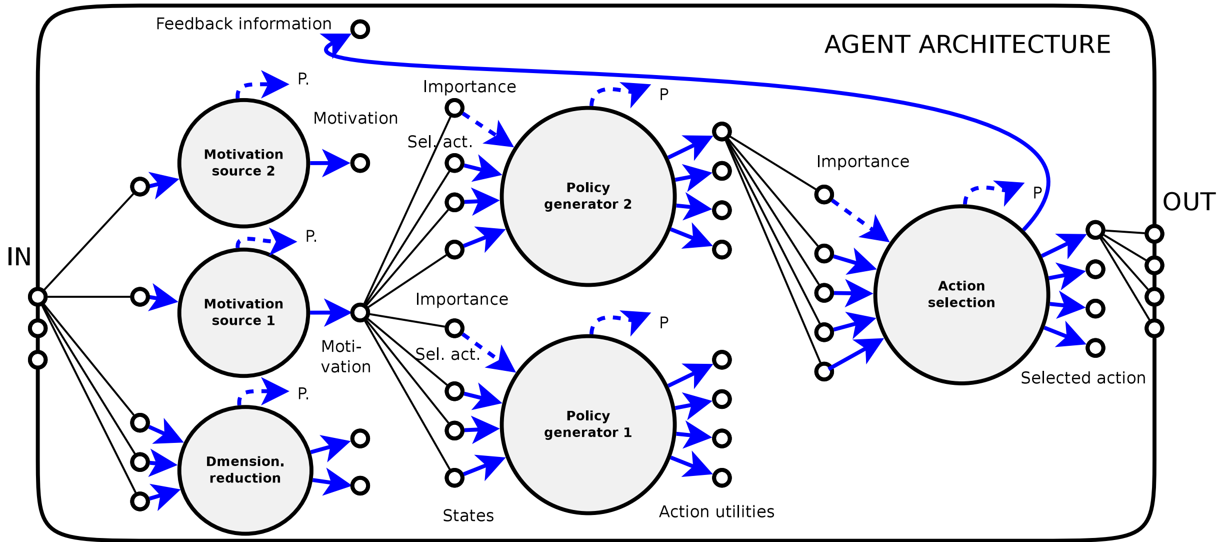


Figure 3.6: *Example of an architecture that is represented as a feedforward network of Neural Modules. On the left there are sensory data, on the right there are outputs to actuators (see the Fig.3.1). The thin lines represent wighted connections between Modules (interlayer connections are shown only for one output). In this example, the dimensionality of sensory data is reduced from 3D to 2D, then the information can passed to two Policy generators. The final action to be executed by the agent is chosen by the Action selection Module. The Module sends information about the decision to the previous layer.*

Currently, the HANNS framework ***represents architectures as hybrid networks with feedforward topology***. This principle can be seen in the Fig.3.6. Similarly to classical ANNs, the HANNS *places Neural Modules in layers*. The Neural Modules between particular neighboring layers are fully connected. This means that the optimization algorithm searches either only in the space of all possible weighted connections between layers.

Unfortunately (as in the human brain) in many systems, the ***feedback connections*** are also important. For instance, the policy generating and learning Neural Module needs feedback information about whether the action it proposed has been actually taken by the agent. Therefore the Neural Modules are *allowed to register feedback connections*

into the previous layer. On the example in the Fig.3.6, it can be seen that the *Action selection* Module sends the feedback information about the action that was selected. This information is fed to the hidden layer containing two (possibly independent) *Policy generators*.

3.3.3.4 Predefined Classes of Neural Modules

In addition to the (mostly) feedforward topology, the agent architectures usually use some types of sub-systems on some places. This is the reason why the author of the thesis defines several ***classes of Neural Modules***. Currently, the proposed framework uses ***three-layer feedforward architectures***, as can be seen in the Fig.3.6. The classes belong to one of these layers in the topology. The classes defined are the following:

Data pre-processing, which implement mainly reduction of dimensionality of input data. Typically, the decision/learning systems "want to" learn something lower-dimensional than raw input data (for example vision). Simple examples of data pre-processing Neural Modules can be Principle Component Analysis (PCA), K-Means algorithm or the Self-Organizing Map (SOM). As more complicated can be mentioned for example Deep-belief Network (DBN). These Modules belong to the first layer in the network.

Sources of Motivation, are Neural Modules that have purpose of generating motivation signal. The motivation signal is fed into other Modules. Each Module decides at each time step "what to do", based on information telling: ***how is my contribution necessary just now?*** Neural Modules may have the ***Importance input***, which tells exactly this information. If the value of Importance is high, the module tends to provide correct information with a high amplitude. In the Fig.3.6, the Policy generators send action utilities to the network. Each of generators scales values of the produced utilities based on the current value of its Importance. The Importance input can be connected to the source of Motivation. The source of Motivation can be either inner (e.g. some physiological Module) or outer (Motivation can be controlled from the outside - by means of architecture inputs). Generally, in the HANNS framework, the Motivation is a way how (and how much of) the signal is spread in the network.

Policy generation, this class belongs to the center layer. It may include Policy genera-

tion Modules (as depicted in the Fig.3.6), or some learning algorithms. An example of policy generator can be mentioned Reinforcement Learning (RL), or Planning Module. As an example of learning algorithm can be mentioned again some clustering algorithm. In this second case, the recognized pattern corresponds to some action that could be taken by the agent.

Action selection. In case that multiple Neural Modules from the class "policy generation" suggest some actions to be taken by the agent, there has to be some node that makes the final decision: which action to take. Therefore the Action Selection class of Neural Modules select (typically) one action, based on action utilities collected from the previous layer. As the simplest example of Action selection Module, the Greedy strategy can be mentioned: an action with the maximum utility is selected to be executed. The Action selection then typically produces output encoded with $1ofN$ code, where only one action is chosen.

The framework is not limited to these classes of Neural Modules. For example, an additional layer could be added (after the "Data pre-processing" one) with the ability of further data processing (for example learning sequences of patterns as shown useful e.g. in (Hawkins, Ahmad, and Dubinsky 2011)). But the three-layer architectures are sufficient in most cases.

The framework is not limited to these classes of Neural Modules. For example, an additional layer could be added (after the "Data pre-processing" one) with the ability of further data processing (for example learning sequences of patterns as shown useful e.g. in (Hawkins, Ahmad, and Dubinsky 2011)). But the three-layer architectures are sufficient in most cases.

3.3.3.5 On Constraining the Dimensionality of Inputs/Outputs

Finally, there is only one detail to be defined. The domain configuration of Neural Modules often includes defining dimensionality of input/output data. This dimensionality influences how many input and output connections a particular Neural Module will have. This means that this must be ***decided before the simulation/optimization***. Intuitively, the choice of input/output dimensionality of one Module should be made so that its connections are "compatible" with Modules placed in the "neighboring" layers. That is: the output dimension of one Module should be similar to input dimensions of the

Module in the following layer, etc. Here, the author suggests two approaches how to choose this property of Neural Modules, where only first one is shown in experiments.

First possibility is to define the input/output dimensionality by hand, based on Neural Modules that are placed in the neighboring layers. Second possibility is then to determine the dimensionality automatically. Here will be a brief description of a proposed solution for such an automatic configuration of Modules.

Usually, ensemble methods combine nodes of MISO type connected into unknown number of layers (the layers are added until some accuracy on data is obtained) (Kordík 2006). This framework is different, because it uses MIMO Modules organized in the three layer feedforward topology. In this topology, the dimensionality of inputs and outputs (of the architecture) is given by the task. That is by properties of sensory and actuator data. By this, one can determine the input dimensions of Modules in the first layer and the output dimensions of the Modules in the last layer. It would be convenient to spread this information through the entire architecture.

Each Neural Module can be equipped by own set of constraints on dimensions of input/output connections. This can be represented by set of linear inequalities. An example of such constraint of Neural Module that requires to have input dimension twice smaller than the output dimension, and the minimum dimension of input is 3, can be written as follows:

$$\begin{aligned} dim_{in} &= 2 \times dim_{out}, \\ dim_{in} &\geq 3. \end{aligned} \tag{3.1}$$

Suppose that each Neural Modules have own set of such constraints. It is then possible automatically determine some reasonable compatibility of input/output dimensions of all Neural Modules in the architecture. All these constraints of neighboring Modules can be composed together for one architecture. Based on these constraints, and the dimensions of inputs/outputs of the architecture, the Integer-Linear Programming (ILP) can be then used to minimize differences between input/output dimensions of neighboring Modules.

3.3.3.6 Defining the Agent's Goals

During the autonomous design of an agent architecture, there has to be some criterium evaluating ***how well the agent performs***. Since the HANNS framework aims to be general as possible, it would be suitable to define some general metric, which can evaluate this. Clearly, it is very difficult to ***generalize what the architecture should do***. That is: to find the domain independent evaluation of agent's desired abilities.

Particularly, the user should be able to clearly state what objectives the architecture should follow. Then, these objectives will be taken into account during the automatic optimization of the architecture. The author proposes the (first two) following original main possibilities of evaluating the agent's performance, which for evaluating the agent's behavior employ the ***Prosperity values*** defined in the Section 3.3.2.2. These are the following:

All of the Neural Modules should be used as efficiently as possible. For the purpose of evaluating the effectiveness of the Module usage, each of the Modules publishes own value of the Prosperity function. The overall performance of the agent is then computed as a composition o Prosperity values of particular Modules.

Architecture fulfills predefined inner needs. In this case, only Prosperities of Modules from the class *Sources of Motivation*. In this class of modules, the Prosperity value is often defined as inverse of average Motivation produced by the Module. This means that the ***Prosperity of Motivation Source defines how well the agent performs during the corresponding task*** - how well is able to fulfill the particular need. In this case, the user defines Source of Motivation and hardwires it in the architecture so that it produces desired need. Then, the quality of architecture is given by composition of Prosperities of these Sources of Motivation.

Architecture fulfills predefined task, this last approach s mentioned for completeness. It is used widely for instance in neuro-evolution, where the performance of the system is measured externally by using some domain knowledge (e.g. explicit measuring average agent's speed in the map).

Note that such a definition of the desired quality measure then leads the optimization algorithm during design of architecture. The architecture is then evolved in order to fulfill given objectives in a given environment.

3.3.3.7 Overall Principle of Automatic Design of Architectures

In the previous Chapters, particular parts of the approach were described. Here, the overall principle of automatic design of new agent architectures will be described.

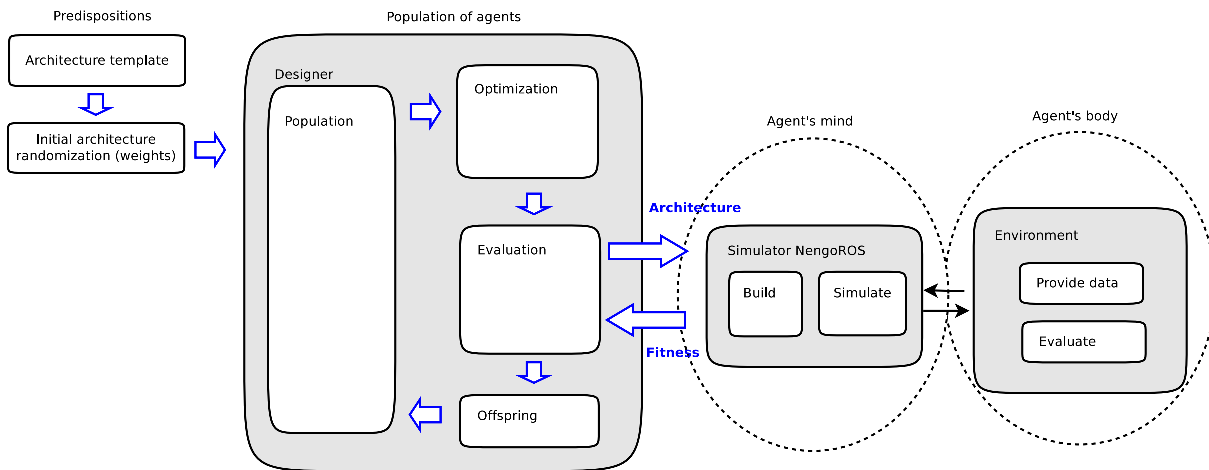


Figure 3.7: *An overall principle of automatic design of new architectures specifically for a given task. This example uses generational model of Evolutionary Algorithm (EA). First, the constraints described above are applied, this produces an Architecture template. Based on this template, the initial population of agents is defined. The EA is used to optimize connection weights between Neural Modules in the architecture. The fitness obtained from the simulation is used to guide the evolution.*

After taking account all details mentioned above, the design process of agent architectures is similar to neuro-evolution. The Fig.3.7 depicts this principle. First, the template of architecture is created by using the constraints described above. The template contains a set of Neural Modules together with their placement in the feedforward topology. The Nodes' inputs/outputs are fully connected between layers. An initial population containing architectures with random weights is defined. Then, the generational model of EA is used to optimize these connection weights.

While taking the architecture template into account, the ***architecture can be represented as a vector of numbers***. These numbers then represent connection weights between Neural Modules (as shown in the section Experiments better). The weights can be either binary, or of continuous value from a selected interval (typically $gene_i \in \langle -1, 1 \rangle$). The use of binary or real-valued weights then determine the EA used. In case of binary

weights ***simple Genetic Algorithm (GA) can be employed***. In case of real-valued weights, the GA needs to be modified (in the simplest case), this is called the Real-valued GA (RGA). Both of algorithms that were used are described in the Section 5.2.1.

The evaluation of individual is done as follows (see the Fig.3.7). Based on the template and genome, the architecture is built in the simulator. Then, the architecture is simulated for a given number of time-steps. The architecture (representing the agent's mind) typically controls agent's body in some (simulated) environment. The required agent's behavior is defined by one of methods proposed in the previous Section, the quality of agent's behavior is then set as the fitness value for the optimization algorithm. The design of the simulator is described in the following Section in more detail.

Recapitulation This Section described the proposed framework of Hybrid Artificial Neural Network Systems, which serves mainly (but not only) for autonomous designing of new agent architecture specifically for a given task. This is accomplished by unified representation of different sub-systems and by defining unified type of communication between them. The Section then described some following issues and their solutions. The framework deals with the course of dimensionality by constraining the architecture topologies to only those, which are potentially most useful for agent architectures. Furthermore, the framework defines several classes of Neural Modules, together with their suggested place in the architecture. Several ways how to define agent's required behavior and how to evaluate it's performance were proposed. Finally, the principle of autonomous design of new architectures was described as optimization of connection weights in pre-defined architecture template.

3.4 Simulator Design

Usually, researchers write entire code (together with simple simulators) for testing algorithms by themselves. One advantage is that such a piece of software is well optimized for their purposes, there are several disadvantages though. There are some time limitations constraining how long the development can be done, so single person is unable to create sophisticated tool enough. The other disadvantage is that such a piece of code most likely will not be reused by any other person, because it is too domain specific. This results in

the situation where everyone is writing the same SW again and again.

The author decided to build the simulator for purposes of prototyping and testing Hybrid Artificial Neural Network Systems (HANNS). The proposed simulator is able to simulate modular systems in general. As described in the Section 3.4.2, there are many problems with designing useful software (SW). Therefore, rather than being a one-purpose tool, the proposed simulator is composed of two main parts:

Simulation engine, which handles one part of communication between Neural Modules and uses as an interface (graphical, command-line and scripting interface) with the user.

Library of Neural Modules. The simulator is designed in such a way, that it is able to use more general-purpose pieces of SW. By adding a small scripted interface (which defines properties of the resulting Neural Module), these pieces of SW can be transformed into Neural Modules, that can be directly used by the simulator. Such a solution has two main benefits. Firstly, the current ***existing SW*** (and the HW too) ***can be re-used in the simulator easily***. Secondly, ***newly implemented SW can be used also without this simulator***, for an arbitrary other tasks in other systems/simulators.

This Section will very briefly describe the simulator, which was designed with the following requirements. Since the HANNS is based on ANNs, the approach was chosen to use current simulator of ANNs and extend its functionality for purposes of this thesis.

- Simulation on level of networks (not single neurons)
- Support for 2nd and 3rd generation of ANNs
- Open-source (need for high degree of customization of simulator)
- Platform independent (ideally Java implementation)
- Support for modular networks
- Simulation acceleration on Graphical Processing Units (GPUs) or by means of Message Passing Interface (MPI)

Other things that were taken into account were: availability of GUI, speed of network design and supported learning algorithms. The Appendix A.3 compares state-of-the-art ANN simulators in more detail.

3.4.1 Simulation Engine

Finally, the simulator Nengo was chosen to be used as a simulation engine of ANN part and as a front-end for the user. The Nengo is a simulator of large-scale spiking ANNs, which is being developed on university of Waterloo. The main purpose of Nengo is to run biologically plausible networks of spiking neurons, which are designed by means of Neural Engineering Framework (NEF) (Eliasmith and Anderson 2003). It is written almost exclusively in Java and is open-souce[2].

The Nengo is used to provide the simulation engine and user front end. It supports GUI and a Jython scripting interface for defining the models. It supports nice real-time plotting of data too. The Nengo simulator modified to provide suitable platform for simulating the hybrid systems composed of re-usable parts. The main contribution of the author is in adding the ability to directly employ Robotic Operating System in the simulation, which is described below. First, the re-usability of particular Neural Modules will be addressed in the following Section. Then, the overall structure of the simulator will be briefly described.

3.4.2 Library - Implementing and Sharing Pieces of Code

Since the requirement for communication by means of real-valued numbers may be too strict for sharing the code, the optional approach was used. Each Neural Module separates the sub-system from information transformations. That is: each sub-system (e.g. algorithm that was shared) can be used also as a stand-alone node without need of converting data into these vectors of numbers.

Another main requirement for reuse of sub-systems is in the programming language and operating system neutrality. A particular sub-systems can be therefore implemented in variety of programming languages and can be freely reused in applications other than the HANNS framework. This was accomplished by use of Robotic Operating System (ROS).

[2]The original unmodified simulator can be downloaded from: `http://nengo.ca/`.

3.4.2.1 Modular and Reusable Design - Robotic Operating System

The ROS is almost completely decentralized node-based system with purpose to simplify and standardize research in the domain of robotics (Quigley et al. 2009). A system built using ROS consists of a number of processes - ***ROS nodes***, running potentially on a number of different hosts, connected at runtime in a peer-to-peer topology. Nodes communicate by passing predefined ROS messages through the TCP/IP protocol - network.

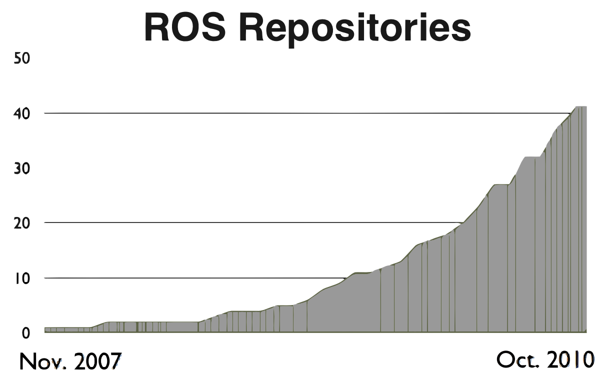


Figure 3.8: *Number of ROS repositories in first three years. Almost exponential growth.*

First, this node-based structure of ROS systems is clearly very suitable for use in hybrid neural systems. Second, the ROS became the standard in robotic research very quickly. The ROS provides a library of reusable ROS nodes (e.g. path-planning, vision, support for various types of HW...) which can be used and modified freely. As can be seen in the Fig.3.8, the number of packages[3] grown almost exponentially. The system is multi-lingual and currently supports the following programming languages: C++, Python, Octave, LISP and several experimental versions, including Java.

The HANNS framework therefore uses ROS nodes as sub-systems. Each ROS node (or group of nodes) can be encapsulated into the Neural Module. This is done by the *modem.* Modem is used as a bridge between ROS infrastructure and the ANN simulator, it is a ROS node which includes API supported by the simulator. The modem holds Encoder and Decoder. These are used for conversion of ROS messages into vector of real-valued data (which is compatible with the unified communication used by HANNS (see the Section 3.3.1)) and back. The schematics of complete Neural Module, which consists of one ROS node, is depicted in the Fig.3.9. Data on inputs of the Neural Module are

[3]Package may hold multiple ROS nodes.

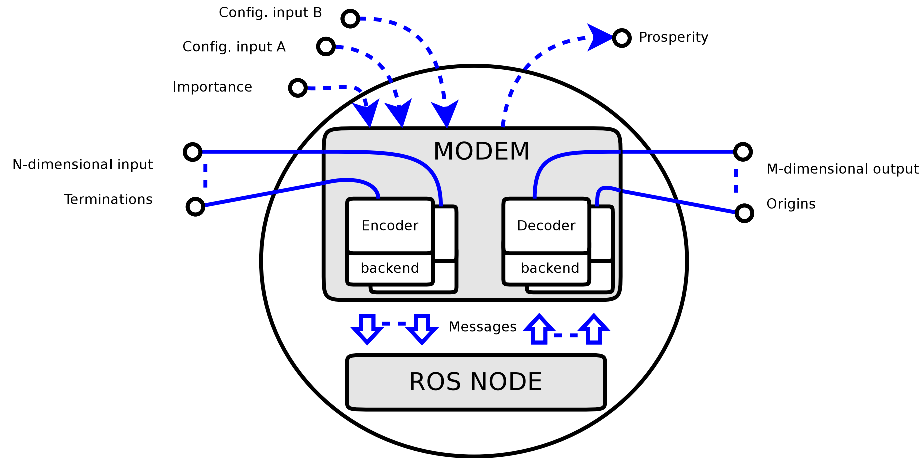


Figure 3.9: *Scheme of the current definition and implementation of Neural Module. A standard ROS node interacts with own modem. Modem contains decoder/coder for each incoming/outgoing type of message. Each coder/decoder knows data topic name and data type of own message and directly sends/listens for new ROS messages. ROS backend is then used to directly convert data between ROS message and real-values received from/sent to the simulator.*

translated into ROS messages, while incoming ROS messages are converted into arrays of real-valued numbers and passed to the rest of the hybrid system. By defining own Encoder/Decoder, the user can define custom communication transformation (e.g. own transformation between symbolic and sub-symbolic representation).

This approach (in using ROS nodes) has two following benefits:

The code can be freely shared and reused. Since the newly implemented SW is not constrained to use only in the HANNS framework, it still serves the purpose of free sharing the code. That is, such an implementation of new algorithm in the field of AI can be freely shared with other researchers, which can use the algorithm directly in their future work (using either the HANNS framework or only the standardized ROS infrastructure).

The Neural Modules can benefit from current ROS nodes. A useful ROS nodes (that are already available in the huge library) can be directly employed in the HANNS framework simply by adding own modem and defining the suitable backend.

So in order to implement own subsystem (Neural Module), user just has to implement

own ROS node[4] which communicates by means of ROS messages. Then, a simple Encoder/Decoder is defined. From now on, the simulator handles translation between ROS messages and a "language of ANNs" (see fig.3.9). This approach provide availability to reuse potentially any currently implemented ROS node without modification. All the user has to do is to enumerate which message streams should be available in the NengoROS simulator and the way how to convert them. Even this process could be automatized in the future, so NengoROS should be able to directly use current library of available ROS nodes in the simulation.

3.4.3 Simulator Engine with the ROS Integration

By extending the Nengo simulator engine with capabilities of using the Robotic Operating System (ROS), the simulator called NengoROS was created. It uses front-end and simulation engine of Nengo, but it adds the ability to work with the ROS directly. The NengoROS is able to start/stop and communicate with arbitrary ROS nodes. This is done by representing (a group of) these nodes as Neural Modules. If such a Neural Module is added into the simulation, the NengoROS initializes the ROS-related components and launches both ROS notes: the modem and the corresponding ROS node. Everything happens on background, so the Neural Modules are represented in an exactly same way as other components in the Nengo simulator.

Commonly, the ROS is downloaded as a compiled package onto the machine running the Ubuntu Operating System. But in case of the NengoROS simulator, the author of thesis used an experimental[5] implementation of ROS in Java - called rosjava[6]. This enables to maintain the platform independency of the entire simulator. The graphical representation of the overall design of NengoROS simulator can be seen in the Fig.3.10. Connecting of Neural Module is done in the simulator engine, while all Neural Modules are running externally. The communication is accomplished by sending messages with data, potentially over the network. This way, almost arbitrary SW, or potentially HW can be employed in the simulation.

The structure of the HANNS network is defined in the Jython script. If the NengoROS

[4]In a favorite programming language, e.g. by following some of many ROS tutorials

[5]Experimental at least in time of developing the SW.

[6]More information about rosjava can be found online at `http://wiki.ros.org/rosjava`.

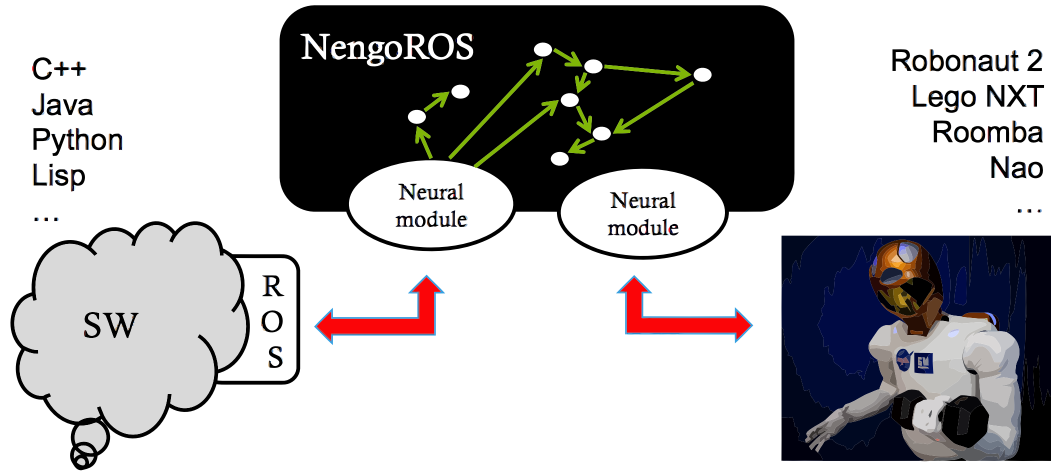


Figure 3.10: *An overall design of the NengoROS simulator. The black windows the simulator engine, while the peripherals are represented as white Neural Modules. This is the way how the simulator integrates various pieces of SW together. Note that each part of the simulation can be implemented in different programming language and can be ran on different machine.*

detects use of Neural Module during instantiation of the module, the ROS infrastructure is initialized (which includes for example automatic launching of the java "roscore"[7]). After launching the core, ROS nodes that required in all Neural Modules used in the architecture are started.

At each simulation step, the NengoROS passes information between Origins (outputs of ensembles) and Terminations (inputs of another ensembles). After receiving new values on its Termination (see the Fig.3.9), the modem encodes the information, sends it as a ROS message for computation and waits for the response. After receiving the message, the response is decoded by the Decoder and its value is placed on the Module's Origin.

3.4.3.1 Autonomous Design of new Architectures in NengoROS

During the autonomous design of agent architecture, the NengoROS simulator is used as follows. The architecture template is loaded from the Jython script. An externally running EA provides genomes to be evaluated. For each genome, the connection weights in the current model are altered and the simulation is restarted. After running the simulation for given number of steps, the fitness value is obtained and next evaluation of

[7]More information about the roscore can be found online at: `http://wiki.ros.org/roscore`.

next genome can be started.

3.4.3.2 Documentation and Other Resources

Despite the fact that the SW is undergoing constant modifications and improvements, it can be downloaded from github[8]. More information about the simulator design, together with tutorials can be found online too[9]. Further java documentation for the simulator and several selected Neural Modules was also published online[10].

3.4.4 Example of Hybrid System Simulated in the NengoROS

The Fig.3.11 shows an example of simple hybrid system composed of one generator of random signal, one population of neurons and one Neural Module. Neural ensemble here implements identity - tries to represent input signal.

Figure 3.11: *Example of simple hybrid system in NengoROS simulator. Neural Module (in the center) has inputs and outputs which correspond to the messages subscribed/published by the ROS node. The ROS node searches for the minimum and maximum value in the input vector of size 4. Neural Module is treated as an ordinary neuron (with more inputs/outputs) by the simulator. Note that multi-dimensional connections are shown as one line here.*

The ROS node used here receives message with vector of 4 float values, rounds the minimum and maximum value to the nearest integer and sends back the message containing these two values. In this case, the user described only how input/output messages look like and how to convert them into vectors of floats. Nengoros then created neural module

[8]NengoROS available online at `https://github.com/jvitku/nengoros`

[9]Information about the simulator design and tutorials available at `http://nengoros.wordpress.com`

[10]Javadoc documentation for selected parts of the project available online at `http://jvitku.github.io/nengoros/`.

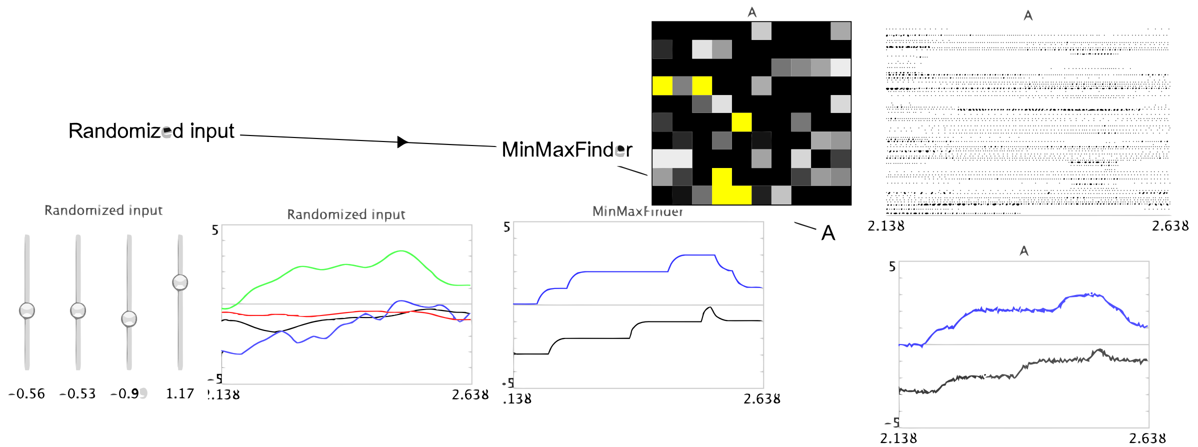


Figure 3.12: *Course of simulation in NengoROS simulator (scheme of the system is in the fig.3.11). Signal generator sends a vector of four values into the neural module. Module publishes rounded values of minimum and maximum values. These two values are then represented by neural ensemble on the right. On the top we can see actual membrane potential and spike raster for each neuron.*

called MinMaxFinder, *which is handled as an ordinary neuron* with more independent inputs and outputs. The screenshot from the NengoROS GUI can be seen in the Fig.3.11. The white big dot represents the Neural Module, which runs externally as a ROS node. An example of real-time plots provided by the original Nengo GUI. It can be seen ho also the output values of the Neural Modules is visible here.

Recapitulation This chapter shown basic description of the resulting NengoROS simulator. It combines the large-scale simulator of SNNs with a modular and multi-purpose ROS nodes. Entire core of the system is implemented in Java and therefore can be ran on variety of platforms. Despite the Java implementation, the simulation can contain ROS nodes that are implemented also in many other programming languages. The basic principle of use of this simulator for designing new architectures was mentioned. Much more information about the simulator can be found online. The simulator uses externally running ROS nodes, that can be freely reused in other systems and also obtained from other systems without requirement of their modifications. Also, not only parts of the agent architectures are used in the simulation in this way. The virtual environments for agents are used in the simulation in the same way as ordinary Neural Modules. So it is sufficient to employ the simulated world with the ROS interface and use it in the NengoROS.

Chapter 4

Theoretical Foundation and Design of Modules

The purpose of this Chapter is to describe Neural Modules (subsystems for agent architectures) that were designed for the HANNS framework. The Modules were implemented, tested and then used for automatic design of agent architectures. First, a basic theoretical foundation is described for each type of modules. Then, specific implementation modifications of the algorithms used are mentioned. These include mainly dealing with requirements posed by the HANNS framework, that is mainly representation of input/output data. Also, not all algorithms used here are strictly domain independent. Therefore second part of each section describes possibilities of tuning the algorithm parameters and its domain configurability. There are some basic test-experiments presented for most of the Neural Modules.

4.1 Neuron Models

Set of components of the HANNS framework contains modules of different complexity. From the simple ones (as for instance neurons, logical gates) towards the more complex ones (such as network of neurons implementing selected function or a planning system). Description and comparison of different neuron models is placed in the Chapter 2.1.1.2. Therefore the basics of neuron models will not be mentioned here again. The original

Nengo simulator provides implementation of various spiking neuron models, such as for example Leaky Integrate and Fire (LIF), Izhikevich (Izhikevich 2003) simple model of neuron etc. For mode information about these models, see the Appendix A.1.

The Neural Engineering Framework (NEF) originally uses networks of these spiking neurons for implementing arbitrary predefined functions. This is done by so-called encoders (equivalent of input weights to the network) and decoders (equivalent of output weights). For completeness, the basic principle is described in the Appendix A.2. In some of the experiments presented, the encoders and decoders provided by the NEF are used for implementing simple spiking neural networks under the HANNS framework. In order to maintain the defined currently supported non-spiking communication in the HANNS framework, each spiking sub-network (Neural Ensemble in the NEF terminology), each output of the spiking neuron (spike raster) is integrated and thus interpreted as a real-valued output (rate code (Abbott and Sejnowski 1999; Ponulak and Kasinski 2011)) (see e.q. the Fig.A.2). Note that all neuron models used are implemented directly in the simulator, therefore none of them is implemented as a stand-alone ROS node.

4.2 Logic Gates

Many tasks can be solved by using simple logical operations. The solution of the task then can be represented in form rules which look like $IF(x) \rightarrow THEN(y); ELSE(z)$. By combining multiple similar rules, relatively complex decision making systems, such as expert systems. For more information and examples, see e.g. (Russell and Norvig 2003).

4.2.1 Theoretical Foundation

In such a network-based framework as the HANNS is, it is intuitive to represent (implement) these logical operations as *networks of logic gates*. By finding the right connection weights in given set of logical gates, various types of problems can be solved. One example of similar approach is called Cartesian Genetic Programming (CGP) (see the Chapter 2.5.1.2). In order to incorporate such logical gates into the HANNS framework (where inputs/output values should be in the interval $\langle 0, 1 \rangle$), some kind of pre-processing

and (potentially) post-processing is needed. This is depicted in the Fig.4.1. Particular requirements will be described in the following subsections for different kind of gates separately.

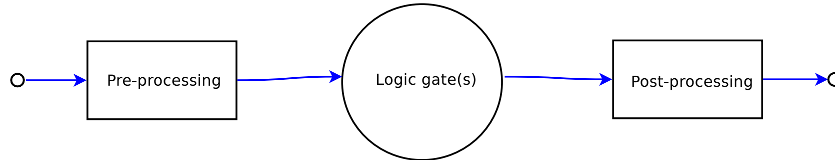


Figure 4.1: *General principle of connecting a logic gate into the HANNS network. Each logic gate expects inputs with a given constraints (pre-processing). Also, the result of computation can be interpreted in several ways (post-processing). In case of fuzzy logic, the pre-processing is called fuzzyfication (converting the input data to fuzzy membership functions). The block called "Logic gate(s)" stands for both knowledge base and fuzzy inference (engine). Finally, the post-processing part is referred to as de-fuzzyfication of data.*

Two main types of logic gates were implemented and tested. The ***crisp logic*** gates are used for implementing crisp logic. Furthermore, to enable the framework to be used for building hybrid ***neuro-fuzzy systems***, fuzzy logic gates (Zadeh 1994) were implemented.

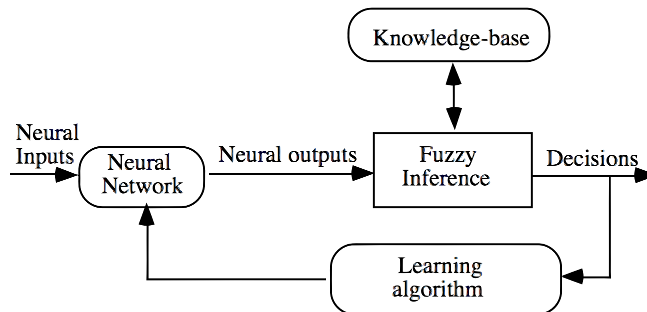


Figure 4.2: *An example of Neuro-Fuzzy system. Neural network is used for pre-processing the data to the fuzzy inference engine. Here, the ANN is used for supervised extraction of fuzzy-rules from data (pre-processing module in the Fig.4.1). In case of unsupervised extraction, the data are clustered according to some similarity measure and therefore the feedback loop is not necessary (Fuller 2001).*

There are various types of neuro-fuzzy systems. These systems combine benefits of ANNs

and Fuzzy-logic together. *"For example, while neural networks are good at recognizing patterns, they are not good at explaining how they reach their decisions. Fuzzy logic systems, which can reason with imprecise information, are good at explaining their decisions but they cannot automatically acquire the rules they use to make those decisions. These limitations have been a central driving force behind the creation of intelligent hybrid systems where two or more techniques are combined in a manner that overcomes the limitations of individual techniques (Fuller 2001)."* In these hybrid systems, the ANNs are often used for data pre-processing for the fuzzy inference system. In case that the pre-processing involves adaptation to data, the fuzzy membership functions do not have to be defined manually, which is a big advantage compared to classical Fuzzy logic. The adaptation can be either supervised (e.g. back-propagation) or unsupervised (clustering techniques). The basic principle of supervised data pre-processing can be seen in the Fig.4.2, where the ANN is used for defining the fuzzy membership functions, which are used by the fuzzy inference system. Note that there are many various Neuro-fuzzy systems (good overview can be seen in (Vieira, Morgado Dias, and Mota 2004)), but the following text will focus on own definition of hybrid logic systems.

As seen in the Fig.4.1, logic gates need some kind of pre-processing and potentially post-processing. In the HANNS framework, both of the operations are considered to be separated from particular gates. This enables the designer to define own kind of modules with this purpose (as for example SOM network, direct sensory data etc. . .). Again, the post-processing can be done by any module. Since the output values produced by logic gates are already in the recommended interval of $\langle 0, 1\rangle$, arbitrary modules can be connected to its outputs (e.g. de-fuzzyfication, input of the planner, direct actuator data etc. . .). This means that the only task of HANNS logic gates is to make sure that input data are of required format. Currently, this is done very intuitively as described in the following sub-sections. A general logic gate in the HANNS framework takes input data, implements logical operation and produces output value.

4.2.2 Crisp Logic Gates

A crisp logic gate takes binary input(s) and produces one binary-valued output. This means that all values should be from $\{0, 1\}$. Basic schematics of the crisp logic gate can be seen in the Fig.4.3. It can be seen that unknown input values (from other gates, sensors

etc...) are thresholded by the threshold $\theta = 0.5$. All values that are not higher than θ are considered to be logic 0, everything other is evaluated as logic 1. After thresholding all input values, the gate implements its computation and passes the result $z \in \{0, 1\}$ to its output. Currently, the following logic gates were implemented and used: AND, NAND, NOT, OR and XOR.

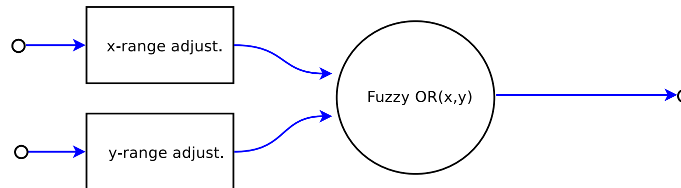


Figure 4.3: *An example of crisp logic gate implementing the AND function. Real-valued input data are thresholded by the value of 0.5. If the input is bigger than threshold, the input is evaluated as logic 1, the logic 0 is considered otherwise.*

4.2.3 Fuzzy Logic Gates

A rough thresholding of input values required by crisp logic gates is not suitable for many real-world tasks. It is probably that many sub-systems will provide real-valued data. For example, such an output can represent membership of data to some class (e.g. output of a sub-system that implements a clustering algorithm). Here, the concept of Fuzzy Logic (Zadeh 1994) can be used conveniently here. More information about Fuzzy Logic can be found in many resources, for example in (Fullér 1995). A Fuzzy-Logic rule accepts value of membership function ($x \in \langle 0, 1 \rangle$) and outputs real valued result of the operation. Note that the de-fuzzyfication (as is marked as post-processing block in the Fig.4.1) is not defined nor implemented explicitly here. Rather, the de-fuzzyfication can be done implicitly in other modules in the network. For example, a sub-network of fuzzy-logic gates can output multiple information telling *"how hot"*, *"how cold"*, *"how rainy"* it is, and these information are processed by some decision-making subsystem.

A fuzzy logic gate takes values of membership functions (values from the interval $\langle 0, 1 \rangle$) and produced output of the operation. The following Łukasiewicz operations were implemented for testing:

- Negation: $F\neg(x) = 1 - x$

- Weak Conjunction: $F(x, y) = min\{x, y\}$
- Weak Disjunction: $F(x, y) = max\{x, y\}$

An example of Fuzzy-Logic module implementing fuzzy-OR operation can be seen in the Fig.4.4. Since it is not necessary condition, that the values passed between modules have to be from the interval $\langle 0, 1\rangle$, the pre-processing of input membership function values is used for each fuzzy-gate as follows: all values outside the interval are placed to the nearer boundary of the interval.

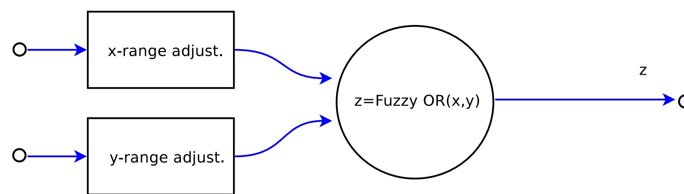


Figure 4.4: *An example of fuzzy logic gate implementing some Fuzzy OR function. Real-valued input data are cropped to be in the interval of $\langle 0, 1\rangle$, the resulting value is sent to the fuzzy logic gate.*

This chapter shown some examples of logic-gates. All gates that were considered so far have either one, or two inputs and one common output. Some of logic-gates can be extended to have arbitrary number of inputs. This would enable one gate to compute e.g. logical AND across multiple variables (thus reducing the number of modules required in the network), but would also increase the number of possible modules. And the bigger number of modules, the bigger space needs to be searched by the design algorithm.

4.3 Inner Sources of Agent's Motivation

In case that there is only one objective to be followed by the agent, a simple goal-directed behavior can be used to solve the task. But often, there are more than one goal to be fulfilled. In this case, an architecture of a rational agent architecture has to be able to:

- Intelligently choose which goal to follow in a given situation.
- Provide some persistence of goal selection. If one goal is chosen, the agent should stick to following this chosen goal for some time. In case that the architecture switches between antagonistic strategies to often, no goal will be reached.

The following chapter briefly mentions some methods how these requirements can be fulfilled.

4.3.1 Theoretical Foundation

This chapter will briefly describe selected two different methods how to switch between various policies of an agent. Despite the fact that both of the mechanisms fulfill requirements specified above, both use fundamentally different approach.

A good example how to switch between different behaviors is the Belief-Desire-Intention (**BDI**) (Sardina et al. 2006) architecture. Even the highest level description of the BDI architecture deals with this kind of "focusing" on selected tasks. The main structure of the BDI agent is s follows in this reminder:

Belief - set of information that agent considers to be true.

Desire - represents what the agent could try to accomplish. A **Goal** is then a Desire which has been adopted for active pursuit by the agent. A set of active goals should be consistent.

Intention - represent what the agent has chosen to do at the moment. It is a Desire which has agent started to perform in some way. Typical way how to fulfill some intention is to execute a corresponding **plan** - sequence of actions.

Such a BDI architecture *is committed* to own choices of plans. Yet, the architecture is still *responsive* - is able to adapt to changes of the situation and to continue to achieve the same goal (re-planning). If the goal is reached (or set of Intention is changed), the agent commits to new Intention, creates new plan and starts to follow this new policy.

Another good example how to create an agent that is committed to changes can be found in (Kadleček 2008). Here the **Hierarchical Reinforcement Learning** (HRL) is used to learn new knowledge and to produce behavior. Simple said, multiple RL nodes compete for control over the agent. The architecture needs to be able to pass the control to the right node in the right moment. There are two following ideas behind convenient system for switching policies (passes control to different RL nodes). For each primitive action (available to the agent) is computed its value (how beneficial it is to execute this action at this moment) and each of RL nodes "votes" how good a given action is from its point

of view. Each RL node takes into account two criteria:

- **How far** the policy (represented by the RL node) is **from the goal** state.
- **How important is the goal** (represented by the RL node) **in a given situation** for the agent.

These two objectives are combined into (so called) "utility values" for all actions available. For each primitive action, the se values are summed together. Then the agent can simply take the action with the highest utility value without considering which policy (policies) the action belongs to. How far the policy is from the goal is described later in the Chapter 4.4. I will now describe **how the current importance of the goal is represented**. Kadleček defines so-called "physiological state-space" (Kadleček 2008), which holds all inner physiological variables of the agent in one state-space. The state-space has purgatory and limbo area. Each variable has own predefined dynamics (each variable typically goes toward the Purgatory area). If the reward is received (e.g. food is obtained), the corresponding physiological variable (e.g. hunger) is moved towards the Limbo area. This temporarily satisfies the physiology and causes the motivation to execute the policy to decrease. This system enables the architecture to switch between various behaviors (policies) dynamically during the agent's life.

4.3.2 Physiological Neural Module

This chapter describes motivation source inspired by the Physiological State Space. In order to **provide the source of motivation**, but maintain the modularity of HANNS, I propose the Physiological Neural Module. Compared to the Physiological State Space, the Physiological Neural Module (PNM) contains only one state variable. In the current implementation, the state variable V has linearly decaying dynamics, as follows:

$$V_{t+1} = V_t - decay, \tag{4.1}$$

the speed of decay can be set during initialization of PNM, or online during the simulation. Then, the amount of Motivation produced by the PNM is determined by applying the

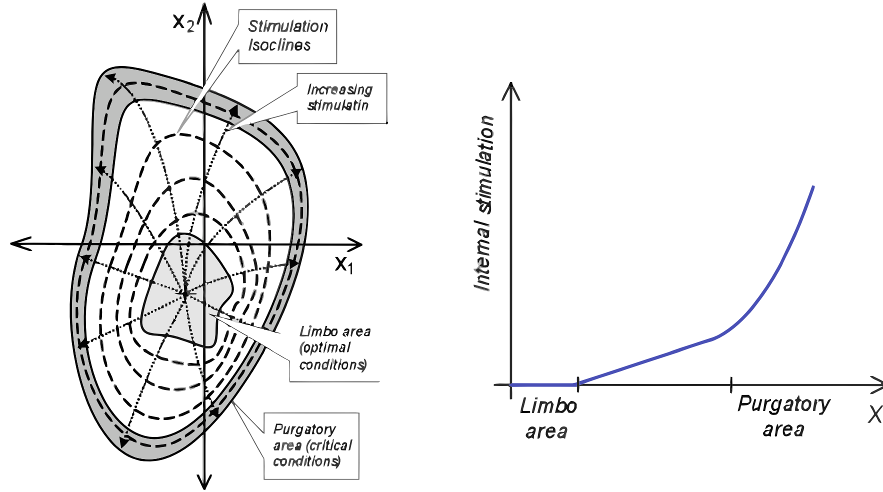


Figure 4.5: *Graphical representation of Physiological State Space. The sate space contains two state variables X_1, X_2 (representing e.g. hunger and thirst). The graph on the right shows mapping of state variable to amount of stimulation produced. This stimulation defines how important a given policy (e.g. "obtain water") is. From the mapping it can be seen that the Limbo area does not produce any stimulation. While Purgatory area produces exponentially increasing stimulation (Kadleček 2008).*

sigmoid to the inverse value of V. The resulting amount of Motivation M at time t is:

$$M_t = \frac{1}{1 + e^{min+(max-min)\times(1-Vt)}} \quad , \tag{4.2}$$

where min and max parameters are chosen so that value of the variable $V_t = 0$ approximately corresponds to the motivation of $M_t = 1$. In case that the reward is received, the value of V_{t+1} is set to 1 and therefore the motivation decreases towards 0. This event then makes room for another behaviors and/or exploration of new knowledge. Note that here, the state variable V has limbo area near the value of 1, this means that the value represents rather the physiological state (such as "amount of food in agent's body"), rather than need (e.g. "hunger"). This process can be seen in the Fig.4.7. For purposes of use in feedforward agent architectures, the Neural Module features also the Reinforcement pass-through output, which mirrors value of reward received on input. This enables to pass the information about reward received further to next layers.

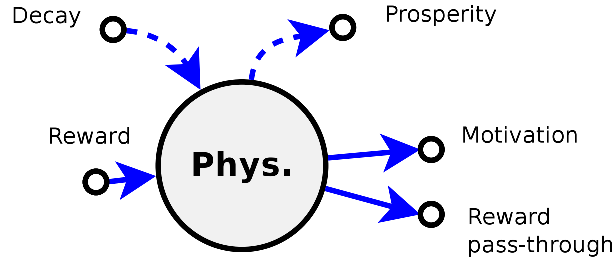


Figure 4.6: *Graphical Representation of Physiological Neural Module. The amount of Motivation produced by the module increased nonlinearly. In case of receiving the reward, the motivation falls towards the zero. The Module contains also pass-through Reward output which just mirrors the value of Reward received on the Reward input.*

4.3.2.1 Prosperity of the Physiological Neural Module

If the agent behaves efficiently enough, the mean Motivation produced by this module is low, because the agent's physiological variable is near the optimal conditions. The Mean State Distance to optimal conditions (MSD) defines how far is the agent's physiology from the optimal values as follows:

$$MSD_t = \frac{\sum_i d_i}{i} \ \forall i \in 0..t, \tag{4.3}$$

where d_i is distance of state variable V_i from the optimal conditions of $V = 1$. The MSD_t is computed online for each simulation step. Since the Prosperity is indirectly proportional to this, its value is computed as:

$$P_t = 1 - MSD_t. \tag{4.4}$$

Simply said, the Prosperity of PMS tells how well is the architecture able to maintain the average Motivation low.

4.3.3 Suggested Use of Physiological Neural Modules

In case of using only one decision-making system in the architecture, such a PNM can be used for example for dynamical weighting between exploration and exploitation, as

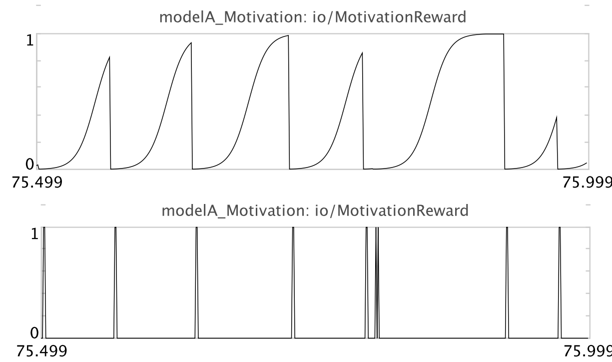


Figure 4.7: *Example of course of value of Motivation produced by the Physiological Neural Module. On the X-axis, there is a time while Y-axes represent binary event of receiving the Reward (bottom) and amount of Motivation produced by the Module (top). It can be seen that the amount of Motivation increases until the reward is received, then the process repeats. After receiving the reward, the Motivation to follow the corresponding policy is small enough to switch to some other behavior.*

described in one of experiments in the Chapter 5. Aside of biologically plausible approach for agent control, the benefit of such an approach is in fact that it prevents the system for getting stuck in some local optima (after reaching the goal/reward, the exploration can be preferred again). Note that this *Motivation output* is meant be connected into the *Importance input* of several Neural Modules (see for example Chapter 4.4.3). The Importance input tells the Module *how important is its service at the moment* and the Module can adapt its strategy (e.g. to prefer exploitation of knowledge) according to this input.

In more complex case, multiple decision/learning systems can be employed in one architecture. Here, multiple of these PNMs can be used to implement multiple independent sources of motivation. These Motivation channels can then be connected to other Neural Modules to dynamically weight between different behaviors. These behaviors can be implemented by different Neural Modules (e.g. Planning, Reinforcement Learning, Random Walk, etc..) and can serve different purposes (e.g. find food, escape from danger, explore..). Each of Physiological Neural Modules can use different inner dynamics for increasing the Motivation produced. Moreover to this, dynamics of each PNM can be changed during the simulation, e.g. by output of some other Neural Module. This enables to get even more complex patterns of behavior switching.

4.4 Reinforcement Learning

This section describes selected Neural Modules, which implement Reinforcement Learning (RL) algorithms. First the main parts of used RL theory are described. Second, the specific Neural Modules are presented together with changes against theory and possibilities of their configuration.

Generally, there are three types of learning/adaptation: supervised, unsupervised and reinforcement learning. In case of Reinforcement Learning, the agent learns solely by interaction with the environment. This is one of fundamental principles of research in the ALife domain. The agent takes actions (interacts with the environment) and observes consequences of these interactions (new states, rewards, punishments..). By means of learning the *"cause and effect"*, the algorithm is able to gradually gain new knowledge through the agent's *"lifetime"*. The RL is originally inspired in behavioral psychology, but since it is very general approach, it is used across more research fields, such as for example: ALife domain, control theory, game theory and Genetic Algorithms (GAs). The following text will describe in the context of ALife and agent architectures.

4.4.1 Theoretical Foundation

There are several basic types of RL algorithms. The following text will focus on the algorithm that is used in my Neural Modules and the rest will be only mentioned for the completeness.

The RL task usually consists of the following parts: set of state variables S together with their values, set of actions A that is agent allowed to execute, transition rules between states and set of rewards R that the environment produces. The task is to learn appropriate strategy π which maximizes cumulative reward that is received from the environment. Many RL algorithms require that the environment is of type called Markov Decision Process (MDP). That is: probability of each (stochastic) state transition $P_a(s_t, s'_{t+1})$ depends only on the current state s_t and action a_t at current time step t. The strategy π that is learned then stores "for all states s information about: which action a should to be executed in order to maximize the reward".

There are several approaches how to obtain such a strategy. For instance, the ***Brute Force*** tests and evaluates all possible policies and samples their outcomes (rewards). The

Monte-Carlo methods use two steps: policy evaluation and improvement. The policy is represented by the function $Q^{\pi}(s,a)$, which maps selected action for each state. The policy evaluation operates across multiple randomly initialized episodes of experiments. The final values of the function are averaged across multiple episodes. In the policy improvement step, the best policy (the value $max_a Q(s,a)$) is selected in each step. Both, the brute force and Monte-Carlo based methods work well only in simpler tasks. The following text will describe algorithms that are used in my RL Neural Modules, that is: those that are called ***Temporal Difference*** (TD) methods. The TD methods are based on the recursive Bellman Equation (thus are inspired in dynamic programming). I use the incremental version of TD methods, which work as follows: action a_t is selected in the state s_t, the reward r_{t+1} and new state s_{t+1} are observed and the $Q(s_t, a_t)$ value is update according by comparing of expected and real outcome of the action. More information can be found e.g. in (Sutton and Barto 1998).

4.4.1.1 Learning

The Q-Learning algorithm is named according to its Q matrix, which maps state-action pairs to utility value. At each environment state, the $Q(s,a)$ matrix stores utility values for all agent's actions A in all admissible environment states S as follows:

$$Q : A \times S \rightarrow \mathbb{R}, \tag{4.5}$$

The utility value represents discounted future reward/punishment that will be received by the agent in case it will follow given action a in a given state s. Online learning is governed by obtaining new $Q(s,a)$ values into the matrix. During the initialization, all values are set to either zero or small random values. Change of the value in the matrix is represented by the value of δ, which is computed according to the following equation:

$$\delta = r_{t+1} + \gamma Q(s_{t+1}, a'_{t+1}) - Q(s_t, a_t). \tag{4.6}$$

At a current state s_t, the action a_t is selected (according to some Action Selection Method) and executed. The algorithm remembers the (s_t, a_t) pair. The execution of action a_t causes transition into the new state s_{t+1} and may cause receiving non-zero reward r_{t+1}.

Based on these information and new action a'_{t+1} in the new state, the value $Q(s_t, a_t)$ is updated as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta. \tag{4.7}$$

The following parameters used: $\gamma \in \langle 0; 1)$ is a forgetting factor and $\alpha \in (0; 1\rangle$ is a learning rate. I used two following versions of TD RL algorithms, where the difference is in the choice of the a'_{t+1} action in the eq.(4.6), these are the following:

Q-Learning - this algorithm is more optimistic one and selects a'_{t+1} as *the best possible action in new state* s_{t+1} based on current knowledge. This means the following equation:

$$a'_{t+1} = a^*_{t+1} = \max_a Q(s_{t+1}, a_{t+1}), \tag{4.8}$$

where the a^*_{t+1} is the best possible action in the state and therefore follows optimal policy π^*. In other words: the Q-Learning assumes that the optimal policy π^* is being followed, which may not be true, or even beneficial.

SARSA - is the abbreviation of State-Action-Reward-State-Action and determines the a'_{t+1} in the following way: it *observes which action* in the s_{t+1} *is actually executed*. That is:

$$a'_{t+1} = asm(Q(s_{t+1})), \tag{4.9}$$

where the utilities of all actions in the current state are passed to the Action Selection Mechanism (ASM), which selects and executes the action a'_{t+1}. The drawback of the SARSA algorithm is that the information about action/state pairs has to be stored a bit longer. But the benefit is that it is able to gain more information from the exploration[1].

According to the equation (4.7), the TD algorithms update only one value at a time, which can be time-consuming. The learning speed can be enhanced by modification called ***Eligibility Trace***, which enables the algorithm to update values of multiple past state-action pairs at one step. Such modification of these algorithms is called Q-Lambda (or $Q(\lambda)$) and SARSA-Lambda (or $SARSA - \lambda$) algorithm. By introducing the error function $e(s, a)$, which is the fundamental for the eligibility traces-based approaches, we

[1] As can be seen in a good tutorial available online at: `http://goo.gl/ORQrFY`

can rewrite the equation as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta e(s, a), \tag{4.10}$$

where the parameter error is defined for each state-action pair as follows:

$$e_t(s, a) = \begin{cases} \gamma\lambda e_{t-1}(s, a) & \text{if } (s, a) \neq (s_t, a_t) \\ \gamma\lambda e_{t-1}(s, a) + 1 & \text{if } (s, a) = (s_t, a_t) \end{cases} \tag{4.11}$$

The equation says that for all state-action pairs, there is a value of error function, which decays in time. If the state-action pair is used (visited), the error function is increased by 1. Such a modified equation (4.10) means that *all state-action pairs are updated* each step.

The decay parameter $\lambda \in \langle 0, 1 \rangle$ defines magnitude of update of previous states. In case that $\lambda = 0$, pure one-step Temporal Difference (TD) learning is used. In case of $\lambda = 1$, the Monte-Carlo learning is obtained. Correct estimation of the λ can improve the speed of learning, but also can cause oscillations in learning.

4.4.1.2 Action Selection Methods

The previous Chapter described only passive observing what is happening and learning what has been observer. The other problem is how to choose actions (a_t) to be executed, this is typically task for the Action Selection Method (ASM). The selection is based on the action utility: *the higher the utility, the higher the expected future outcome.* Therefore a straightforward solution is to use the ***Greedy ASM***, which takes the action with the highest utility:

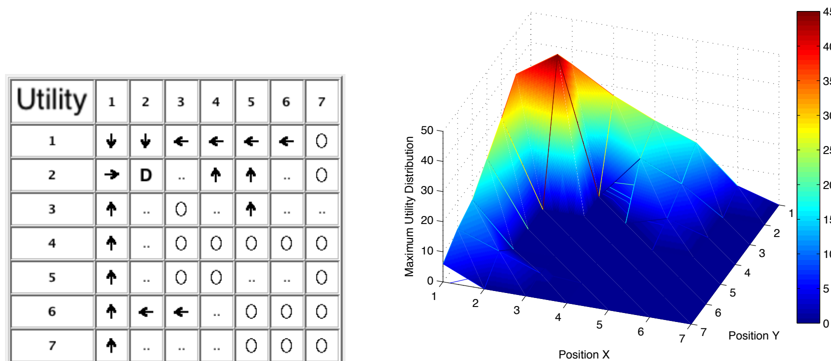
$$a_t = a_t^* = \max_a Q(s_t, a_t). \tag{4.12}$$

This algorithm exploits the knowledge well (always follows the π^*), but does not allow any exploration - gaining new knowledge. This drawback can be tackled by use of the ***Epsilon-Greedy*** ($\epsilon - Greedy$) AMS. The method introduces parameter ϵ, which affects

the amount of randomization. Random action is selected with the probability of ϵ, while the Greedy strategy is followed with the probability of:

$$P(a_t = a_t^*) = 1 - \epsilon. \tag{4.13}$$

This means that the parameter ϵ directly weights between exploitation of knowledge and exploration of state space.



Utility	1	2	3	4	5	6	7
1	↓	↓	←	←	←	←	O
2	→	D	..	↑	↑	..	O
3	↑	..	O	..	↑	..	..
4	↑	..	O	O	O	O	O
5	↑	..	O	O	..	..	O
6	↑	←	←	..	O	O	O
7	↑	..	..	..	O	O	O

Figure 4.8: *Example of simple grid environment which is difficult to explore in non-episodic expriments (Vítků 2011). The world consists of 2D grid with obstacles (marked as dots in the left picture) and one source of reward (marked as D). The agent is able to move in 4 directions. The value of utility for the best action (in a given state) is depicted in the graph as a function of agent's position. The table on the left depicts the best action learned for the sate. The mark "0" indicates that all actions have utility of 0. It can be seen that the agent explores only near neighborhood of the reward and is unable to go far way in the maze. This problem is often tackled with episodic-experiments with randomization of initial conditions (agent's position).*

Typical use of TD algorithms is in episodic experiments, where the initial state of environment is selected randomly. This helps to uniform exploration of (and learning in) the entire state-space. In non-episodic experiment, the value of ϵ has to be carefully tuned, it it is too high, the knowledge is not exploited enough. On the other hand, if the ϵ is too low, the agent will tend to stay only around the nearest the attractor. An example of this problem can be seen in the Fig.4.8.

In order to add domain-independence, the designed module has *to be able to operate in non-episodic experiments*. In non-episodic experiments (particularly those simpler with

one attractor), there is necessary to find a better way how to balance between knowledge exploitation and exploration/learning. In the real-world, there are better and worse situations for learning often. For example, young animals learn by play primarily in near-optimal conditions. It is convenient to simulate agents' needs for something. If a ***need*** for some resource is high, the motivation causes *agent to focus* on a particular behavior which leads to satisfying this need. The concept of motivation is then used for dynamic adjusting the *exploration* vs. *exploitation* tradeoff. The approach is called ***Motivation-Driven ASM***.

This is the reason why I introduce an input to Neural Module called **"Importance"** which generally defines the ***current need for "services" provided by the module***. In case of the $Q(\lambda)$ module, the importance input represents *motivation for behavior represented by this node*. The amount of randomization in the ASM should be indirectly proportional to the importance input. We use the $\epsilon - Greedy$ ASM where the randomization is defined as:

$$\epsilon_t = \begin{cases} 1 - Importance_t & \text{if } 1 - Importance_t > \epsilon_{min} \\ \epsilon_{min} & \text{if } 1 - Importance_t \leq \epsilon_{min} \end{cases} \tag{4.14}$$

By increasing the importance of the $Q(\lambda)$ module (increasing the motivation for executing a behavior represented by this node), the probability of taking the greedy action a^* increases. This means that the importance enables the agent to learn by exploration in free time and to exploit the information if needed.

4.4.2 State of the Art - Reinforcement Learning-Related

This sub-chapter will describe relevant publications to this topic, that is: modularization of one system into multiple RL "modules". Since a RL algorithm often needs to store the quality of given policy in some memory, there are constraints on tasks that can be solved by the RL. In case of Q-Learning and SARSA algorithms, the $Q(s, a)$ matrix grows with number of actions, state variables and their number very fast. This limits the use of RL algorithms (which use the memory in form of the look-up table) considerably. Currently, there are two approaches how to deal with the course of dimensionality. First, the values in the memory can be fitted for example by some mathematical function, ANN, or Fuzzy logic (Busoniu et al. 2010). The second approach is in decomposing the decision space

into smaller sub-spaces, which is called *modular, or hierarchical RL* or can be also referred to as an *ensemble RL*.

This second approach is interesting in the context of this thesis, so it will be covered in more detail. In case of modular RL, the algorithm is encapsulated as standalone subsystem, which is similar to the Neural Module. This particular example (use of RL in HANNS) can be likened to Ensemble Algorithms in Reinforcement Learning (Wiering and Hasselt 2008), or to the Aggregated Multiple Reinforcement Learning System (AMRLS) (Jiang 2007). Compared to these, a single ensemble is represented as a Multiple-Input Multiple-Output sub-system, which communicates compatibly with 2nd gen.

The goal of the implementation of these RL Modules in the HANNS framework is their domain independence. Therefore the user can instantiate and use multiple RL ensembles (Neural Modules) and connect them in various ways. Another benefit of modularization of RL to multiple sub-tasks is in the ability of the resulting system to learn/execute multiple policies in parallel. In fact, there are the following possibilities of use multiple RL modules in one architecture:

- Each of RL modules can learn the same policy, but receive different input data.
- If the reward signal of different RL Neural Modules is connected to different sources, these can either: compete to learn/exploit antagonistic strategies (Kadlecek and Nahodil 2008) (e.g. drink vs. eat),
- or to fuse cooperation/competition in case of partially shared goals (Jiang and Kamel 2006; Araabi, Mastoureshgh, and Ahmadabadi 2007) (e.g. food is near the water source),
- or to learn/exploit compatible sub-goals of one (hierarchically) composed strategy (Dietterich 2000; Bakker and Schmidhuber 2004) (e.g. open the door and go for a drink). Such a combination of Modules can then result in a system similar to Modular Hierarchical Reinforcement Learning Algorithm (MHRL) (Liu, Zeng, and Liu 2012).

By choosing different combinations and different purposes of these RL Neural Modules, either an efficient state-space decomposition or learning robustness can be acquired. In the system called Hierarchy, Abstraction, Reinforcements, Motivations Agent Architecture (HARM), such a decomposition of the decision space into hierarchical RL can be made automatically, based on different rewards received from the environment (Kadleček

2008). Each RL module then corresponds to some strategy in the environment, these strategies can be represented as abstract actions for the planning system (Vítků 2011). Compared to this, in the HANNS framework the purpose of particular RL modules can be determined either by the human designer (by connecting module to particular sources of data), or by the EA algorithm by connecting its data (and reward) inputs.

4.4.3 Stochastic Return Predictor Module

In (Kadleček 2008), the Stochastic Return Predictor (SRP) is described as a stand-alone subsystem which is composed of the RL algorithm and the ASM. The RL part learns (that is: updates information in the $Q(s,a)$ matrix) from sequences of actions, state transitions and rewards. Based on the current utility values in the $Q(s,a)$ matrix, the ASM selects actions to be executed in the next step.

```
   Data: Values of State Variables, Reward Value
   Result: Continuous Learning by Interaction
 1 Initialize ;                         // configure to given number of inputs/outputs
 2 while True do
 3     a_t =ASM.Select action(Q, s_t)
 4     Publish Selected Action a_t
 5     while MessageFilter.NewStateNotDetected() do
 6         Publish "NO-OPeration";                  // Wait for action to take effect
 7     end
 8     t = t + 1
 9     Observe New State s_{t+1}
10     Observe Reward Received r_{t+1}
11     Update values in Q(s_t, a'_t);                    // see the Eqs.4.8 and 4.9
12     Apply the Eligibility Trace to multiple state-action pairs;    // see the Eq.4.10
13     Update Importance input and configuration inputs
14     Update and publish Prosperity
15 end
```

Algorithm 1: Function of the Stochastic Return Predictor.

This Chapter will describe how the SRP is implemented in the HANNS framework. The modifications of $Q(\lambda)$ and SARSA-Lambda algorithms were implemented, but the following text will describe the main principles only on the $Q(\lambda)$ algorithm. The schematics of implementation of the Module is depicted in the Fig.4.9.

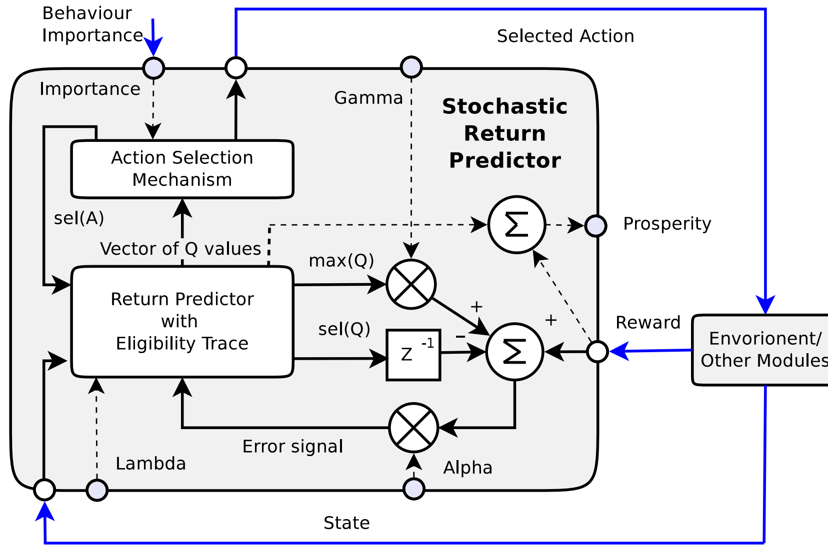


Figure 4.9: *Schematics of the SRP implementation, which is composed of the Q-Lambda algorithm and the ASM. The (sub-)system is implemented as a stand-alone ROS node, which can be used as a Neural Module. Small (grey and white) circles represent interfaces with other modules. The world state is represented by M state variables sampled each into given number of values. Action selection is encoded by the* $1ofN$ *code. Node selects one action at each time-step and expects information about new state and real-valued reward. The node has the following configuration inputs:* α, γ, λ, *Importance, which affect learning and Action Selection Methods. The Prosperity heuristics represents average between MCR and overall coverage of state space (number of visited states).*

The Fig.4.10 depicts implementation of the module as a ROS node and its interfacing with modem. The modem enables neural-like communication and is compatible with NengoROS API. The subsystem is implemented as generally as possible. Still, before the instantiating of the module, the number of data outputs and data inputs need to be connected. Each of data outputs represent one action (action selection is encoded by $1ofN$ code). Each of data inputs represent one state variable. The state variables expect values from the interval $\langle 0, 1 \rangle$. Number of values of variable is determined by the sampling interval. This means that continuous input values are discretized by sampling into predefined number of values. This adds to the generality of the module, because this discrete version of Q-Learning can be used in both: discrete and continuous domains.

The Algorithm 1 describes a high-level functionality of the SRP module. Compared to

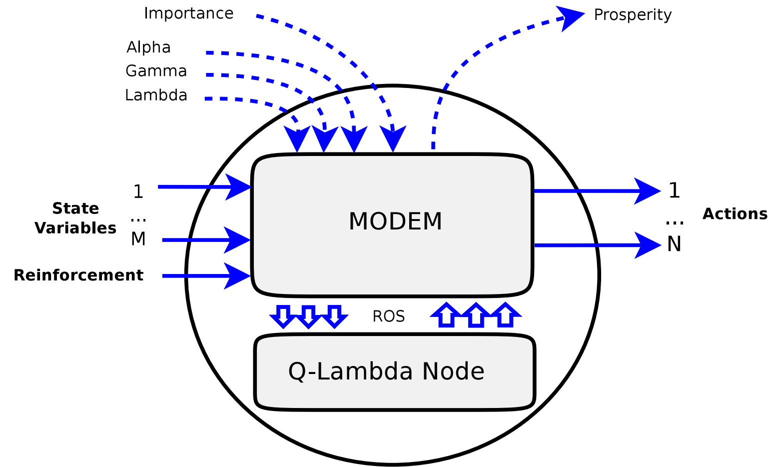


Figure 4.10: *Stochastic Return Predictor communication. The standalone ROS node communicates over the ROS network by means of own modem. Dashed arrows represent optional connections. Optional inputs, which are not connected maintain their default (predefined) values. Number of input and output data connections is chosen before instantiation of the module.*

common implementations, there are the following improvements.

The Eligibility trace is constrained to *finite length* of N previous steps, which saves computational resources and prevents bigger destabilization of learning convergence by incorrect value of λ parameter.

Importance-Based Action Selection Method enables the module to operate in non-episodical experiments. In light of disadvantages of classical ASMs, this module uses the modified $\epsilon - Greedy$ ASM, where value of ϵ is determined online, based on the Importance input. The higher importance of the behavior learned by this module, the less randomization is introduced in the action selection. On the other hand, if the choice of action is not important at the moment, the ASM prefers exploration - gaining new knowledge, rather than blind following the learned policy. The value of ϵ is computed according to the Eq.4.20.

Computation and Publishing of the Prosperity values is done inside the Neural Module. The Prosperity Function is a subjective heuristics telling how successful/useful the module is in the current network in the current simulation. The SRP has two main goals. First, to visit as much world states as possible (to cover the entire state-space regularly in the ideal case). Second, the Module should be able to receive the reward regularly, if it is necessary (which is determined by the current

value of Importance in this case). Multiple definitions of the Prosperity function were tested, but one of the simplest ones proved to be the most convenient. It is composed of two parts:

$$P_t = \frac{Cover_t + MCR_t}{2}, \tag{4.15}$$

where the Mean Cumulative Reward (MCR_t) defines how often the Module receives the reward:

$$MCR_t = \frac{\sum_i R_i}{i} \;\; \forall i \in 0..t, \tag{4.16}$$

and the $Cover_t$ represents how big part of the state-space has been explored during the current simulation:

$$Cover_t = \frac{\sum_{i=visited} s_i}{N}, \tag{4.17}$$

where N is total number of states in the state-space. The size of state space is determined from number of state variables and number of their values. These information are defined during the instantiation of the module.

Input Data Synchronization ensures that all data arrive synchronously. The NengoROS uses event-driven computation, where each event is triggered by receiving new data. Most of the data are passed by means of asynchronous ROS *"Publisher/Subscriber"*[2] method. This means that the state description and reward value may not be received in the correct order (at the same time in this case). In order to solve this problem, both, the reward and state descriptions are sent in one ROS message.

Action-Feedback Synchronization is a system that ensures that the action-reaction cycle will be processed correctly by this Neural Module. If the there is only one SRP connected in the loop with the world, everything works correctly. This means that for every action produced, the response from the environment (new state and reward) is obtained immediately. But in case that the closed-loop obtains more than two Modules in series, the architecture starts to operate as a FIFO (First-In First-Out) system. The architecture composed of $N-1$ Modules connected in series creates N-step long queue producing N-step delay in the action-perception loop.

[2]More information about ROS messages can be found here: `http://wiki.ros.org/Messages`.

Therefore the Module has to be able to assign correct actions to correct responses. This is solved by the $MessageFilter$ in this case. This can be seen in the Algorithm 1 at the line no.5. The filter decides whether the new state belongs to the last action selected by the Module. The SRP publishes new action and the filter waits until the state changes. If the state remains unchanged more than K time steps, the situation is evaluated as a case where the action had no effect on the environment, and the main cycle continues.

4.4.4 Reinforcement Learning Module

There are drawbacks and benefits of incorporating more complex Neural Modules in the hybrid networks. The main benefit is that complex Module can implement complex algorithm, which can reduce a complexity of topology required for solving the problem. The drawback is that the more complex system, the more specialized it usually is. The same is true for the SRP described in the previous chapter. The SRP chooses an action and expects that this action (or some transformation of this action) will be executed by the architecture. This restricts the resulting systems to contain only one decision-making system in most cases.

Solution to this problem is in separating the ASM from the RL part of the SRP Module. The Fig.4.11 depicts the schematics of the RL Neural Module without the ASM. Instead of choosing an action, this Module publishes the prosperity values of all actions in a given state. This means that it tells to the rest of the system *"how good which action is"*. This approach enables the RL Module to *"vote"* in a group of multiple decision-making Modules (not necessarily RL-based ones). As expected, this modification increases the complexity of the topology. This is caused by two main reasons: an additional Module (ASM) needs to be added. Furthermore, an additional feedback connection from the selected action to all RL Modules used is necessary, since the RL algorithm needs to know which action was selected.

In addition to features described for the SRP Module, I introduce more new concepts:

Importance-Based Action Selection Method It is expected that the ASM Module will not know which action is the most important at the moment. Rather, simple results of voting of (RL) Modules will be available. The Utility Values from all RL Modules would be summed and the action with the highest priority will have the

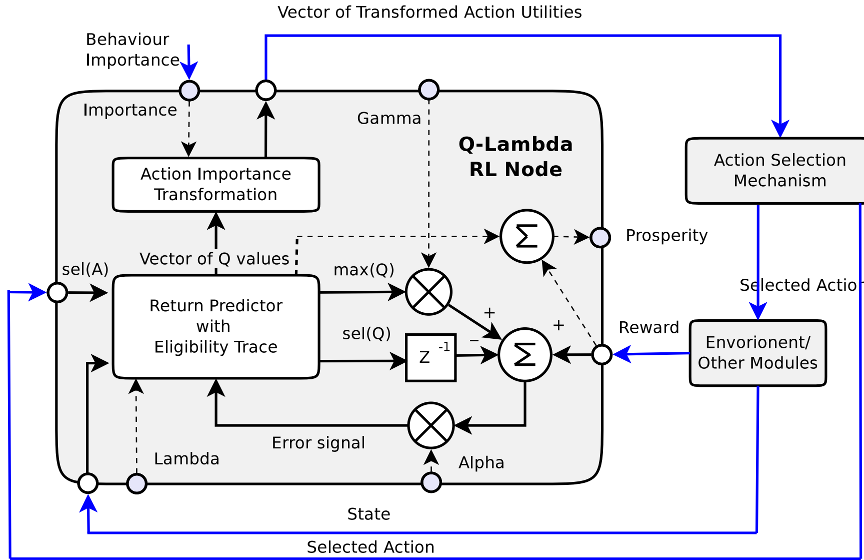


Figure 4.11: *Stochastic RL Neural Module. Compared to the SRP depicted in the Fig.4.9, this module does not contain ASM block. Instead of choosing one action and setting this action to value of 1, this module publishes modified Utility values of all actions in the current state. The higher the value of Importance, the more the output values will be scaled up. Then, some external ASM receives these published values and is used for selecting one action. Finally, the required information about the selected action is received from the outside as a data input.*

highest chance to be selected. Note that each of RL Modules knows how important is its behavior (defined by the Importance input). In order to enable Importance-Based ASM, the published Utility values are scaled according to the Importance value: the higher the Importance input is, the higher will be published. This ensure that the RL which implements most important strategy (in a current situation) will get the highest voting right.

4.4.5 Action Selection Mechanism Module

Similarly to the previous case, the Action Selection Mechanism is just a separation of the current ASM from the SRP. Again, the ASM Module implements the Importance-based ASM, according to the Eg.4.20. Note that the value on this Importance input affects randomization in the overall policy, which may be composed of multiple sub-policies produced by multiple RL Modules (or generally other Neural Modules). This Module

receives vector of real-valued Utility values and produces information about the selected action, which is encoded by $1ofN$ code.

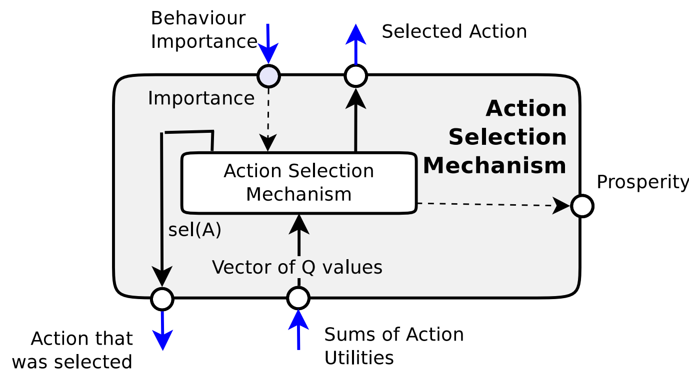


Figure 4.12: *Basic Schematics of the ASM Module. Each action selection is done according to the values of action utilities received from the outside. The vector of values is received, then, the computation begins and finally, the resulting action is encoded by means of $1ofN$ code. Similarly to the SRP, the Importance input controls the amount of randomization during the action selecting.*

4.4.6 Suggested Use of RL Modules

This sub-Chapter describes how these modules could be used in agent architectures, how should interact with each other and with the rest of the architecture.

4.4.6.1 Stochastic Predictor Module

In the context of layered (feedforward) architectures, the SRP Module should be connected in the last layer, where the outputs are connected onto agent's actuators. The anticipated use of SRP module is in the architecture composed of:

Pre-processing layer, which may convert sensory data streams into some lower-dimensional more stable patterns.

Learning and decision-making layer, which is implemented by single SRP.

Optional post-processing layer, which may transform data from the SRP into the format more suitable for actuators.

In case of usage of multiple SRPs in the architecture, there are the following theoretical constraints. At least one of them should be fulfilled in order to system operate correctly:

- Consequences of actions produced by individual SRPs are independent of each other.
- All SRPs choose to execute the same action most of the time.

These requirements are caused by the fact, that there is no feedback about the fact that the selected action was executed by the agent. Each SRP "does not know" whether the architecture took the vote of the SRP and executed the action, or whether something different has been chosen.

4.4.6.2 Reinforcement Learning Module with Separated ASM

This disadvantage mentioned above can be solved by separating the ASM from the RL algorithm. In this case, the multiple RL Modules operate in parallel, either cooperatively or concurrently. The potential architecture may look as follows:

Pre-processing layer, which may convert sensory data streams into some lower-dimensional more stable patterns.

Learning layer, which is implemented by multiple RL Modules. Each Module can learn own strategy, which is determined by connecting of its Reward input onto a particular source of signal. Modules can either cooperate on some strategy, or can learn the same strategy with multiple parameters.

Action-selection layer, typically includes some type of ASM Module, which chooses the action to be executed. The ASM select and executes the action and sends this information back to RL modules. Thus each of RL modules can learn from the experience correctly.

There are also other possibilities how to incorporate these RL-based Modules into hybrid networks, several of them will be presented later in the text.

4.5 Planning

In order to enable HANNS-based agents to perform even more deliberative choices of actions than the RL provides, the planning engine can be added. Most of the more complex agent architectures make explicit use of planning engines for planning their future actions. In order to reach the similar level of behavior complexity the HANNS framework should contain the planning Neural Module too.

4.5.1 Theoretical Foundation

Generally, the planner is a system which is able to propose a sequence of actions that lead from the current state to the specified goal state. Typically, planners work with set of world states, set of actions together with their effects. Executing of each action changes state of the environment in some way - causes transition from one state to another. The planner searches for the correct sequence of such transitions (actions) which consecutively transforms the current state into the specified goal state.

Therefore the planner engine requires the following main things in order to operate: definition of a planning problem, current state and goal state. The definition of a planning problem describes set of possible states and set of admissible actions together with their preconditions and effects. Probably the most known standard in planning problem definition is Planning Domain Definition Language (PDDL), which separates ***planning domain description*** and ***planning problem description*** (current state and goal).

4.5.1.1 Stanford Research Institute Problem Solver

Another well known planning language (and planning engine) is called *Stanford Research Institute Problem Solver* (STRIPS), which is older and simpler than the PDDL (Fikes and Nilsson 1971). The language is formally represented as a quadruple $\langle P, O, I, G \rangle$. The P is the set of conditions expressed by propositional variables describing the world state, I is the description of initial state and G is description of properties which are fulfilled in a goal state(s). O is the set of operators - actions, each operator consists of the quadruple $\langle \alpha_s, \beta_s, \gamma_s, \delta_s \rangle$. The elements α_s and β_s describe the constraints when the action can be applied, that is: describe which conditions must be true and which false in

the given situation. The elements γ_s and δ_s describe action effects after its application, that is: which propositional variables will become true and which false. Particularly, the current state of the world is described by a binary vector, where operators change values of bits on a specified position in a specified manner. The plan is a sequence of applicable operators that consecutively transform the description of initial state towards the state which fulfills the goal conditions (Nahodil and Vítků 2012b).

4.5.2 Design of the Planning Neural Module

The of problem description poses a challenge to the implementation of planner as a Neural Module, because such a Neural Module should maintain as domain independent as possible. There are two possible approaches how to solve the problem: to *load the domain description before the simulation* or to *let the planner learn the appropriate rules of the domain* itself. Since inputs of the planner can be connected to other modules that can produce unknown (read: human un-readable) data, the first approach cannot be used. This means that the planner has to implement some type of the algorithm described in the Alg.2. Such a planner constantly observes sequences of transitions $state_t \rightarrow action_t \rightarrow state_{t+1}$, stores frequent rules and uses these rules for creating new plans. Assuming that the resulting rule database is consistent with true rules of the domain, the planner will produce correct plans.

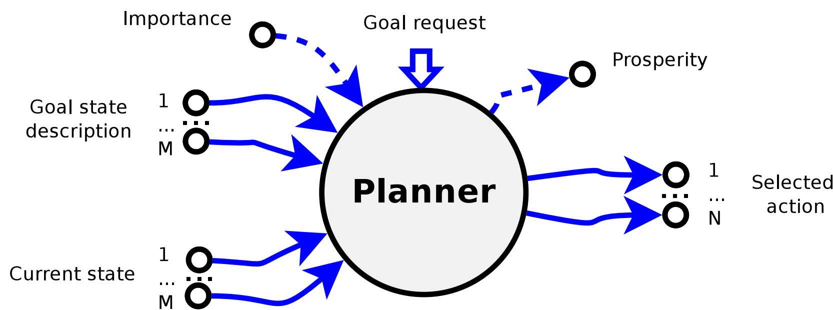


Figure 4.13: *Graphical representation of the Planning Neural Module. The M (lower) inputs represent description of the current state, while the M upper inputs represent a a description of the goal state. The planner chooses one action to be executed in the next step by publishing 1 on the appropriate output (coding $1 of N$). For testing purposes, the goal state can be specified externally too.*

```
   Data: Number of inputs, number of outputs
   Result: Continuous learning and providing planning for request
1  Initialize ;                  // configure planner to given number of inputs/outputs
2  while True do
3     Decoder.decode new state ;            // Read new data on inputs and decode them
4     Decoder.decode new goal ;                      // Read description of new goal
5     Observer.observe new rules ;                      // action - states - action
6     RuleMiner.update rule database ;             // filter and store frequent rules
7     if Importance > θ then
8        Planner.replan ;                  // Using new state and current database..
9        action = Plan.get first action
10    end
11    else
12       action = random
13    end
14    Encoder.encode action
15    Encoder.publish action
16 end
```

Algorithm 2: High-level operation of the required domain-independent planner.

The designed Neural Module uses the STRIPS planner, which operates over binary vectors. The planner has two input vectors: current state and the goal state (see the Fig.4.13). The input values are thresholded by $\theta = 0.5$ in order to get binary data. There are two modes of operation. The planner is in ***exploration mode***. It produces one action (see the Algorithm 2) and updates own library of rules. In case when the value of Importance is above the 0.5, the planner switches to the ***exploitation mode*** and tries to reach the goal state. During the planning, the planner uses the A^* algorithm for searching the graph of possible states. The heuristics is defined as a hamming distance of currently expanded state from the goal state.

In the current implementation, the planner observes all rules (in form of $state_t \rightarrow action_t \rightarrow state_{t+1}$) and remembers M most frequent ones in average. The size of rule database is defined by hand so that the planning is not too slow. In the future, the significant improvement could be made in employing some form of rule generalization.

Finally, note that my work on this Neural Module lies mainly in this higher-level design. The particular implementation and testing was part of the Bachelor Thesis of Lukáš Skála (Skála 2013), therefore experiments with this Module will be mentioned here only very briefly.

4.6 Sequence and Pattern Recognition

There are some decision-making systems may be capable of handling streams of large and noisy data (such as vision). An example of system which should be theoretically capable of processing of this type of data is called Hierarchical Temporal Memory (HTM) described in (Hawkins and Blakeslee 2004). Bit the fact is, that most of nowadays higher-level decision (and learning) systems are not well suited for processing such type of information. It is much more convenient to have low-dimensional and stable representations (that is: changing slowly in time). For example, the Planing Neural Module described in the Chapter 4.5.2 is able to operate only with limited number of states. Also, the planner expects that description of a state is not changing too often (so that the planner does not need to re-plan each time step).

Exactly this purpose can have sequence and pattern recognition sub-systems in agent architectures. Such a subsystem learns patterns from data, classifies them and publishes only information about given class. These pre-processed data can be then passed to some higher-level decision system, which is able to process them well. Based on a particular task, a successful agent architecture should be able to recognize either ***spatial patterns*** or ***temporal patterns*** (and possibly both types). This chapter will briefly describes implemented examples of such systems.

4.6.1 Spatial Pattern Recognition

The requirements is to create a Neural Module which converts high-dimensional to lower dimensional data. The sub-system should be domain independent and should operate in the "forward mode". This means that no labeled data are provided and some output (classification) should be provided from the beginning of the simulation. This results in requirement of ***unsupervised classification*** algorithm. As examples of suitable algorithms can be mentioned Principal Component Analysis (PCA), K-means algorithm, or the Self-Organizing Map (SOM).

4.6.1.1 Design of the Self-Organizing Map Neural Module

The SOM converts high-dimensional input vector to its lower dimensional representation. A typical SOM algorithm works in two phases: learning (adaptation of weights) and mapping (classification of input data).

```
   Data: SOM topology, learning rate, randomized weights
   Result: Continuous learning and providing classification
 1 Initialize
 2 while True do
 3     Input.receive data sample;                      // Vector of real values
 4     if Learning is on then
 5         for Each node in the map do
 6             Compute euclidean distance between sample and weights
 7             track BMU ;                    // Node with the smallest distance
 8         end
 9         move the BMU and its neighbors closer to data sample
10     end
11     Publish coordinates of the BMU ;        // ..in the lower dimensional space
12 end
```

Algorithm 3: Operation of the SOM Neural Module.

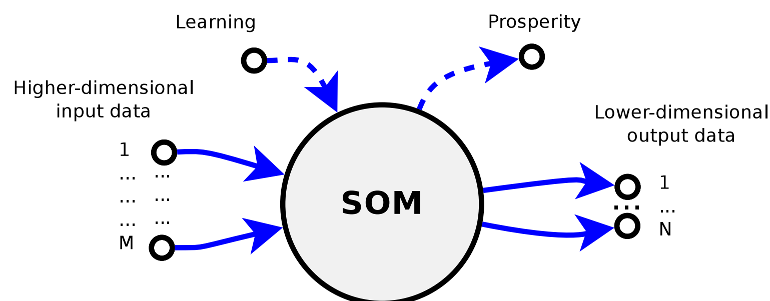


Figure 4.14: *Graphical representation of SOM Neural Module. The Module converts higher-dimensional data to their lower-dimensional representations. The optional configuration input defines whether the learning is enabled (enabled by default).*

The learning works as follows. For each data sample, the distance (of its weights) from

the sample is computed for all neurons in the network according to the equation:

$$d_i = \sum_{j=1}^{M} (I_j - W_{i,j})^2, \tag{4.18}$$

where d_i is a distance of neuron i to the data sample I and M is the dimensionality of input space. The weights of the BMU (the neuron with the minimum d_i) are modified according to the equation:

$$\mathbf{W}(t+1) = \mathbf{W}(t) + \eta \times (\mathbf{I}(t) - \mathbf{W}(t)). \tag{4.19}$$

The parameter η is s learning rate. Then, neighbors of the BMU are moved, also according to the Eq.4.19. But the learning rate η is decayed according to some rule (e.g. the Gaussian distribution) for these neighbors. The neighborhood is defined by the topology of the SOM, typical topologies are 1D chain or 2D grid. More on SOM topologies can be found e.g. in (Jiang, Berry, and Schoenauer 2009).

The designed SOM Neural Module uses the Algorithm 3 during the simulation. During the initialization, the input and output space dimensions and the learning rate are defined. And the SOM with randomized initial weight is instantiated. In the so-called "forward and online mode" the node computes and publishes the transformation at each simulation step from the beginning of the simulation. The learning is enabled by default, therefore the SOM adapts to data online, while also making the classification. Note that the SOM Neural Module has also been implemented and tested as a part of Bachelor Thesis of Lukáš Skála (Skála 2013).

Note that Neural Module with more complex algorithm for ***unsupervised clustering*** (based on a hierarchical clustering) was implemented and tested as a part of Bachelor Thesis of Pavol Sekereš (Sekereš 2013). The Module uses algorithm called *"Unweighted Pair Group Method Algorithm"* (UPGMA). The UPGMA is a modification of hierarchical clustering algorithm, which is able to process data presented in a sequential manner. New data update the hierarchy in a very computationally efficient manner, which is very convenient feature in the HANNS framework. The schematics of the Module looks similar to the SOM node (Fig.4.14), but the main advantage is that the optimal number of clusters does not have to be specified a priori.

4.6.2 Temporal Pattern Recognition

Another useful feature of data pre-processing subsystems is to be able to recognize (classify) the input sequences of data. Thanks to the classification of input sequences, the more stable (less often changing in time) patterns are created. This "gives a time" to decision making systems to plan next actions in a given situation. As an example of sequence classification can be mentioned notes of song and the name of the song (which changes considerably less often in time). The problem can be also called *Unsupervised sequence labeling.* The same requirement as for spatial patterns holds also for temporal patterns: the node should be able to operate in the "forward and online mode". Generally, the algorithm should be able to do: mine frequent sequences, classify (recognize) new sequences received on inputs.

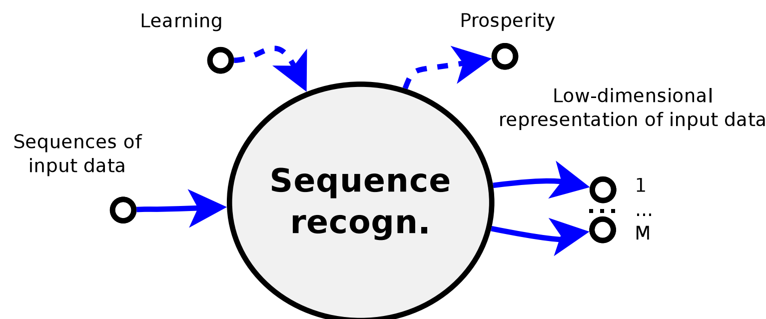


Figure 4.15: *Graphical representation of the Sequence Recognition Neural Module. The sequentially presented input is processed by the algorithm and the resulting (recognized) pattern is published. The learning is turned on by default, but can be controlled during the simulation too.*

As an example of such an algorithm capable of "unsupervised sequence labeling" can be mentioned Hidden Markov Models (HMM). As a part of his Diploma Thesis, my colleague Pavol Sekereš implemented and tested Neural Module which uses algorithm designed (by the original author) especially for this purpose. Based on given requirements (streaming of data, domain independent operation) the algorithm called ***Top-K Sequential patterns in Data Stream*** was selected (Fournier-Viger and Tseng 2011). Since a description of this algorithm is beyond the scope of this text, I will just refer to the Bachelor Thesis (Sekereš 2013). The graphical representation of such a sequence recognizing Neural Module can be seen in the Fig.4.15.

4.7 Neural Modules for Simulating Agent's Environment

Generally, there are three expected types of possible experiment setups, these are the following:

- Testing the architectures directly on some dataset. The dataset was usually loaded from the file by the specialized Neural Module and provided data in the NengoROS simulator.
- Simulation with discrete state simulator.
- Simulation with continuous state simulator.

This Chapter describes two simulators (for simulating agents' environment), which were for experiments. Both of them were used also as Neural Modules and were seamlessly connected in the HANNS framework as any other Neural Modules. The first simulator is an example of complete implementation of very simple simulation engine. Compared to this, the second also serves as an example of adding (Nengo)ROS interface to the existing simulator and using it as a Neural Module.

4.7.1 Discrete Grid-World Simulator

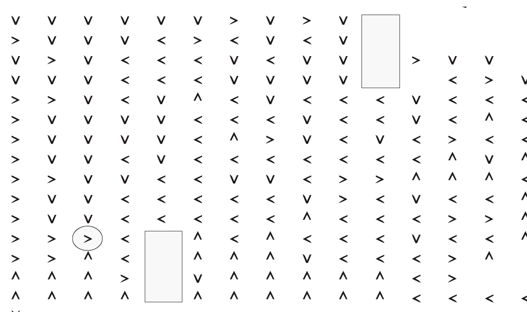


Figure 4.16: *An example of GridWorld simulator initialized with size 20x20 tales. There are two obstacles (big rectangles) and one source of reward (small one). The arrows visualize some learned strategy by the agent (see the Chapter 5).*

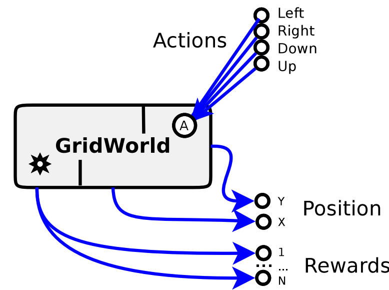
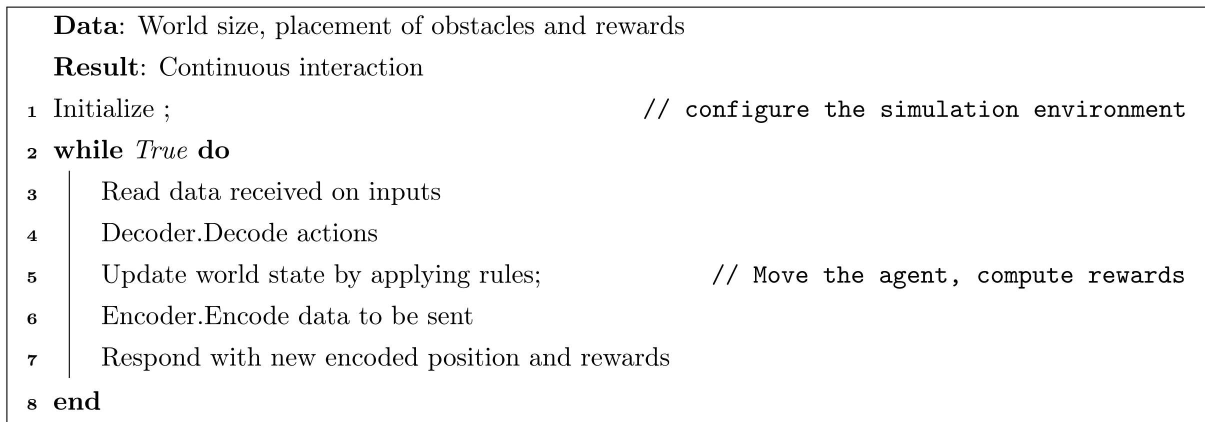


Figure 4.17: *Graphical representation of the GridWorld simulator. After receiving an action (encoded typically by $1ofN$ code) the simulator does the following: Makes transition into new state by applying world rules. Publishes new coordinates of the agent and new reward values.*



Data: World size, placement of obstacles and rewards
Result: Continuous interaction

```
1 Initialize ;                                     // configure the simulation environment
2 while True do
3     Read data received on inputs
4     Decoder.Decode actions
5     Update world state by applying rules;        // Move the agent, compute rewards
6     Encoder.Encode data to be sent
7     Respond with new encoded position and rewards
8 end
```

Algorithm 4: GridWorld simulator operation.

Simple Grid-World simulator was used for purposes of testing architectures that use discrete algorithms for decision making and do not include any sensory data preprocessing. The simulator consists of single world map consisted of tales arranged in a 2D grid of selected size. Each tale can be either free, occupied by obstacle or by some source of reward/punishment (of different types). An agent is allowed to move in four directions (up, down, left, right). If the agent tries to step onto obstacle, nothing happens. If the agent steps on a tale with source of reward/punishment, the corresponding response from the environment is received. Various sources of reward/punishment can be added to the environment during the instantiation. Each such a source has own output in the HANNS network.

The simulator itself is implemented as a ROS node and communicates in the same way as other Neural Modules. The graphical representation of the simulator can be seen in the Fig.4.17. During the initialization, the grid size, positions of obstacles and rewards

are defined. For each type of reward, there is one output added.

Encoding of input/output data is made in the following manner. Ideally, the input data should be from $\{0, 1\}$, where action to be taken is encoded by means of code $1ofN$ ($1of4$ in this case). In order to be compatible with possibly less compatible sources of signal, the decoder (see the Algorithm 4) takes the action with the highest value and considers it to be the one selected by the agent:

$$a_{selected} = max(a_i) \; \forall i \in 1...4. \tag{4.20}$$

On the other hand, the Encoder of data (agent's X,Y positions and rewards) publishes values from the interval $\langle 0, 1 \rangle$ (see the Fig.4.17). In case of the agent's position, the interval of the output value is sampled into N values, where the N is a size of particular dimension (X or Y). This ensures that the data will be from the recommended interval. Also, a Neural Module which receives these data has either compatible decoder, or receives at least approximate information about the situation (that is: lower number means lower coordinate).

4.7.2 Simulator with Realistic Physics - ViVAE

Discrete simulations do not reflect enough properties of real world, such as for example: dynamics, continuous changing of input data etc. In order to test agent architectures under such conditions, appropriate simulator had to be used. While one of the key concepts of the NengoROS simulator is ability to reuse SW, I decided to adopt some of existing simulators. I chose the Visual Vector Agent Environment (ViVAE) (Drchal et al. 2011), which is implemented in Java and was developed on CTU in Prague[3].

It is a simulator of 2D environment with simulated physics. Robots (visualized as squares with sensors, see the Fig.4.18) are able to move on different surfaces. Each surface has different friction and the agent can control speed of both wheels. For each robot, it is possible to configure number of sensors and their properties (maximum distance). A typical task of agent is to navigate through the environment with the highest speed possible.

[3]More information about the original version of the simulator can be found at: `http://cig.felk.cvut.cz/projects/robo/`.

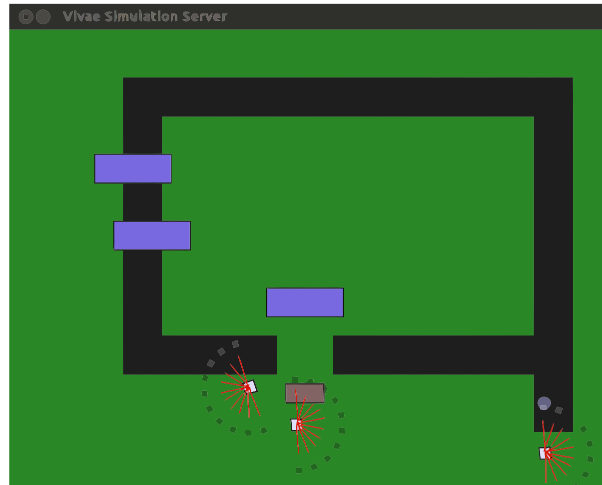


Figure 4.18: *An example of ViVAE simulator window. The simulator contains contains three robots. Each robot has friction (small grey squares), obstacle sensors (red lines) and a speed sensor. Each surface has a different friction (black road with low friction and green grass with bigger friction). Therefore the robot is able to gain maximum speed on the surface with low friction. Also, there are three violet obstacles. Each robot has two actuators, which independently control the speed of robot's wheels.*

I modified this simulator by adding the ROS interface. The modification results in a *ViVAE Simulator Server*, which enables the NengoROS to directly control entire simulator across the ROS network. The description of the interface is beyond the scope of this text, but here is a list of some important functions:

- Start/stop the simulation
- Turn the visualization on/off
- Spawn/Kill the agent (which adds/removes an agent from the simulation) with given number of sensors. This is similar to Spawn service in the Turtlesim demo[4]
- Control existing agents
- Read sensory data from existing agents

For each of newly spawned agents, the ViVAE Simulator Server adds new ROS publisher and subscriber to the node. This increases the number of input/output connections of the

[4]Turtlesim service demo available at: `http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams`

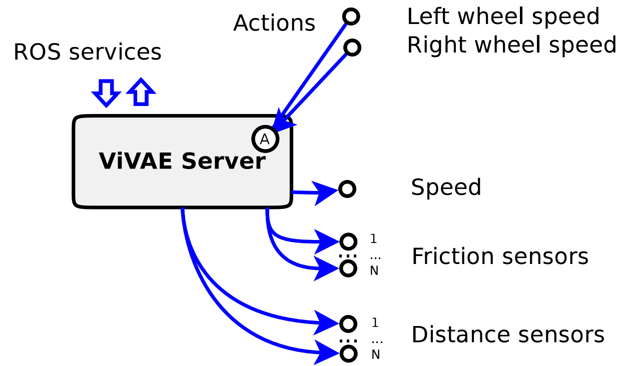


Figure 4.19: *Graphical representation of the ViVAE Simulator Server as a Neural Module. In the picture, the simulator contains one spawned agent with N friction and distance sensors. The Neural Module has two inputs (speed for both wheels) and $2N + 1$ outputs, each for one sensor. The Simulator Server also provides ROS services, which are directly accessible from the NengoROS scripting interface. The services provide control of the simulation setup and state.*

corresponding Neural Module in the NengoROS simulator. The course of simulation is similar to the previous case: after receiving actuator commands (motor speeds from the interval $\langle -1, 1 \rangle$), the simulator state is updated (computing dynamics, resolving collisions etc..) and the resulting sensory data are encoded and sent back to outputs of the Neural Module. Note that no further pre-processing nor post-processing of data is used here. The robots will accept only positive values on actuators and their motors saturate on the value of 1. All outputs from sensory data are also real-valued from the recommended interval.

Chapter 5

Experiments

In order to present concepts outlined in the Chapter 3 and test implementations of various Neural Modules (and simulators) presented in the Chapter 4, it was necessary to conclude variety experiments. This chapter will described several of them, starting from those simpler ones and aiming towards those more complex ones. There are three types of methods of evaluation of agent architectures; testing: against an artificial dataset, in discrete simulator and in continuous simulator with simulated physics. There are the following types of experiments: simpler experiments focused on presenting basic concepts of the HANNS architectures, more complicated hand-designed hybrid architectures and finally: evolutionary-designed hybrid architectures that were optimized for a particular task.

5.1 Hand-Designed Architectures - Testing Neural Modules

The Chapter will start with selected simpler experiments that were concluded mainly in order to verify correct functionality of both: Neural Modules for architectures and external simulators. Also, the basic concepts of HANNS-based simulations will be shown here.

5.1.1 Hand-Designed Agent Controllers - Navigation Task

First experiment shows simple architecture with purpose of navigating agent in a virtual environment. The environment setup is depicted in the Fig.5.1, where the agent should navigate on the black road. The objective of the experiment is to maintain as high average speed as possible.

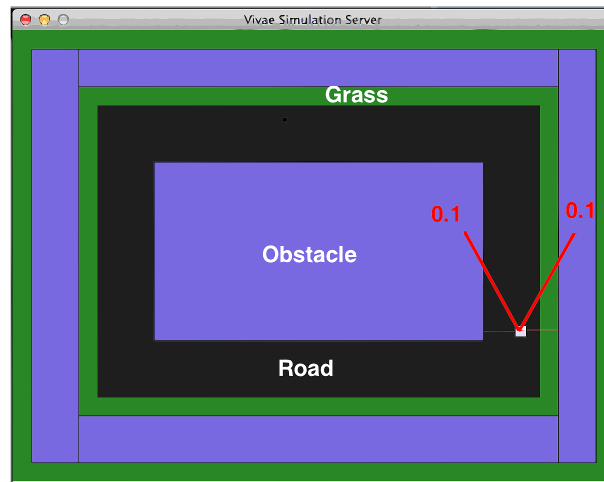
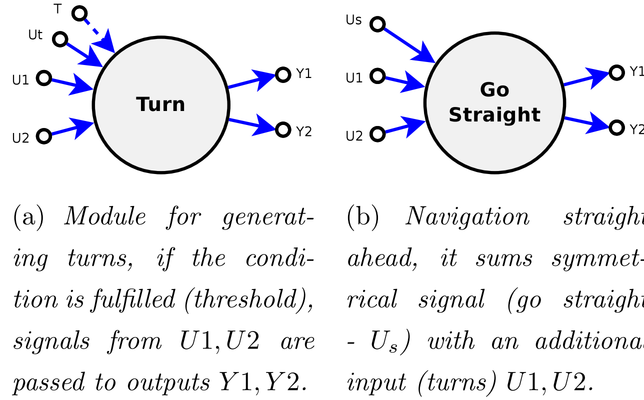


Figure 5.1: *ViVAE environment with simple "maze". The agent should be able to navigate on the black road as fast as possible. The two obstacle sensors publish values from $\langle 0, 1\rangle$ representing distance to the nearest obstacle. Various surfaces have various frictions (see the Section 4.7.2). The robot has two wheels with controllable speed.*

5.1.1.1 Controller Design

In order to build the architecture which implements the required task, a hybrid network of Neural Modules was created. The task can be decomposed into two primitive behaviors in this case: the robot needs to ***navigate straight ahead*** and to ***handle turns in one direction***. Therefore the network consists of two Neural Modules and has feedforward topology. One Neural Module detects and handles turns and the second one handles navigation straight ahead and composition of both behaviors together. The design of the architecture uses simple approach: *go straight ahead, if the left sensor detects too small value: turn left.*



(a) *Module for generating turns, if the condition is fulfilled (threshold), signals from* $U1, U2$ *are passed to outputs* $Y1, Y2$.

(b) *Navigation straight ahead, it sums symmetrical signal (go straight -* U_s*) with an additional input (turns)* $U1, U2$.

Figure 5.2: *Two simple Neural Modules for navigation task. One generates turns and the other one navigates straight ahead. Two output values control speeds of two wheels of a robot. On the left, there is picture of robot using only two distance sensors and two wheels.*

5.1.1.2 Neural Modules

Two Neural Modules were designed, their graphical representation is in the Fig.5.2. First Module detects the turn and produces *turning behavior*, if the sensor value is under a given threshold, initiate the turning behavior for a given duration time. This this is implemented by the Eq.5.1:

$$
\begin{aligned}
\text{if } U_t(t) < T(t) \quad &: \quad \begin{cases} Y_{1a}(r) = U_{1a}(r) \\ Y_{2a}(r) = U_{2a}(r), \quad r \in t, \ldots, t+d \end{cases} \\
\text{else if } U_t(t) \geq T(t) \quad &: \quad \begin{cases} Y_{1a}(t) = 0 \\ Y_{2a}(t) = 0, \end{cases}
\end{aligned} \tag{5.1}
$$

where t is a current simulation step and d is a duration of turning behavior (triggered by crossing the threshold) in time steps. The behavior *go straight ahead* and composition of both behaviors is implemented in the second Neural Module, marked "Go Straight", which computes the following equations:

$$
\begin{aligned}
Y_{1b}(t) &= U_{1b}(t) + U_s(t) \\
Y_{2b}(t) &= U_{2b}(t) + U_s(t)
\end{aligned} \tag{5.2}
$$

5.1.1.3 Resulting Architecture

The resulting architecture is represented as a two-layer feedforward hybrid network. The architecture is connected in the closed loop with the Neural Module implementing the ViVAE simulator server. The agent reads the sensory data (see the Fig.5.1) and sends commands controlling speed of two wheels of the robot. The complete experiment setup can be seen in the Fig.5.3.

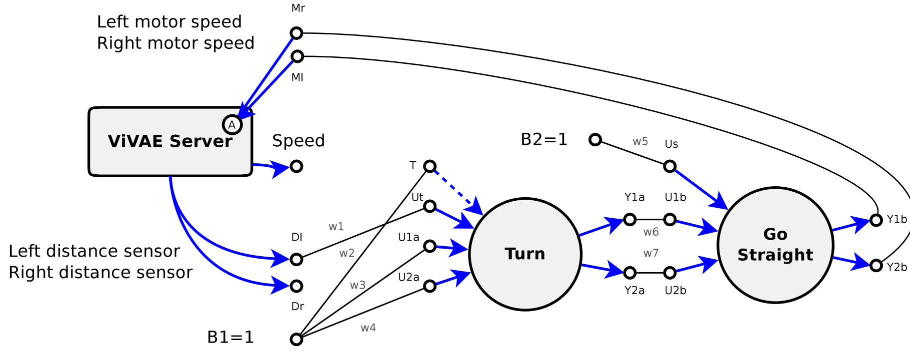


Figure 5.3: *Simple hand-designed architecture composed of two Neural Modules and two sources of bias. It is connected in the closed-loop with the simulator. First Neural Module detects turns and produces turning behavior, the second one produces "go straight ahead" behavior and combines both together. Thin lines between Neural Modules represent weighted connections.*

Thin lines between Neural Modules represent weighted connections between them. The network from the Fig.5.3 gives the following resulting equations, for the first Neural Module:

$$\begin{aligned} \text{if } D_l(t)W_1 < B_1W_2 \quad &: \quad \begin{cases} Y_{1a}(r) = B_1W_3 \\ Y_{2a}(r) = B_1W_4, \quad r \in t,\dots,t+d \end{cases} \\ \text{else if } D_l(t)W_1 \geq B_1W_2 \quad &: \quad \begin{cases} Y_{1a}(t) = 0 \\ Y_{2a}(t) = 0. \end{cases} \end{aligned} \tag{5.3}$$

For the second Neural Module, it is:

$$\begin{aligned} Y_{1b}(t) &= B_2W_5 + Y_{1a}(t)W_6 \\ Y_{2b}(t) &= B_2W_5 + Y_{2a}(t)W_7. \end{aligned} \tag{5.4}$$

So, the resulting system is controlled as follows:

$$\begin{aligned}
&\text{if } D_l(t)W_1 < B_1W_2 \quad : \quad \begin{cases} Y_{1b}(r) = B_2W_5 + B_1W_3W_6 \\ Y_{2b}(r) = B_2W_5 + B_1W_4W_7, \quad r \in t,\ldots,t+d \end{cases} \\
&\text{else if } D_l(t)W_1 \geq B_1W_2 \quad : \quad \begin{cases} Y_{1b}(t) = B_2W_5 \\ Y_{2b}(t) = B_2W_5. \end{cases}
\end{aligned} \tag{5.5}$$

5.1.1.4 Behavior of Resulting Architecture

Here, both values of bias (B_1, B_2) were set to 1, the connection weights $W_{i\in\{1,7\}} \in \langle -1, 1\rangle$ and duration d were experimentally chosen in order to obtain desired behavior. The agent runs straight ahead with predefined speed. If the left turn is detected, it changes the speeds of motors so that it navigates through the turn in an efficient way.

The values of weights were following: $W_{i\in\{1,7\}} = [1, 0, -0.44, -0.11, 0.44, 1, 1]$ and the duration $d = 30$ steps. From the Fig.5.4 can be seen that the turning behavior is triggered by zero value on the left sensor. The "Turn" Neural Module produces negative values (breaking). The graph shows the agent, which navigated through two turns (while one turn was used on the straight row). It can also be seen how the speed is decreased during turns.

The behavior of resulting agent controller, together with interactive plots provided by the NengoROS simulator can be seen from the video[1]. Since the architecture is designed experimentally, the resulting behavior is not optimal. Rather, the purpose of this experiment is to show how a simple agent architecture can be represented and designed with help of the HANNS framework and tested in the NengoROS simulator.

Recapitulation Here a simple experiment was described - a navigation task. The basic principles of designing and simulating HANNS the architectures were explained on a particular example. All the presented Neural Modules can be reused in other systems. The resulting behavior of the agent is given only by the connection weights between Modules, which is similar to designing classical ANNs. The experiment in the Section 5.2.2 will compare this hand-designed architecture with similar one, that was designed by the Evolutionary Algorithm.

[1]Video available online at: `http://goo.gl/RGb6F5`.

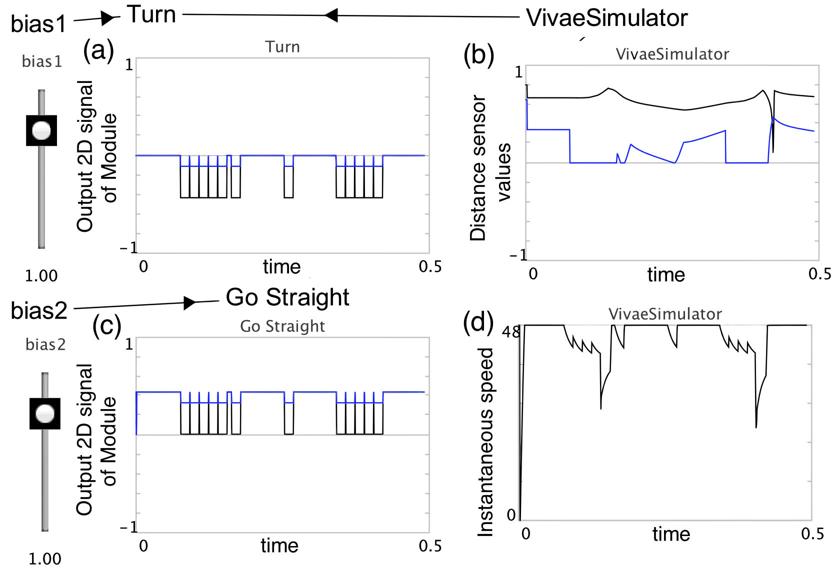


Figure 5.4: *Course of sensory and actuator data during the navigation task. Graphs (a) and (c) show output values of Neural Modules "Turn" and "Go Straight". The (b) and (d) graphs show output values from the ViVAE simulator. In (b), the lower average value is the left sensor value. In (d), the instantaneous speed is of agent shown, big negative spikes correspond to turns, small ones to small corrections in the path. It can be seen that if the turn is detected (graph (b), events when the sensor value touches the value 0), the "Turn" module produces braking (negative speed signal). The "Go Straight" Module (in (c)) produces constant signal, which adds to the values received from the "Turn" module, shown in (a).*

5.1.2 Q-Learning Based Agent Architecture

The following agent architecture was used for testing the Physiological Neural Module (PNM) (see the Chapter 4.3.2), Stochastic Return Predictor Module (SRP) (described in the Chapter 4.4.3) and the GridWorld simulator (Chapter 4.7.1). The main task is to efficiently learn how to obtain reward in the discrete environment. The simulation was setup as non-episodic, which means that the agent was let in the environment for given number of steps. Then, the overall efficiency of agent's behavior was then evaluated.

5.1.2.1 Architecture Design

The architecture was tested in the GridWorld of size 20 x 20 with two obstacles and one attractor, the environment can be seen in the Fig.5.7, left. The agent can move in

four directions and receives the reinforcement after reaching the position containing the reward. The architecture uses SRP configured to have 4 outputs and 2 data inputs. Each of data inputs represents one axis in the GridWorld and therefore each of input variables is sampled into 20 discrete values. One Physiological Neural Module was connected into its input, both reward and Importance. The resulting experiment setup can be seen in the Fig.5.5.

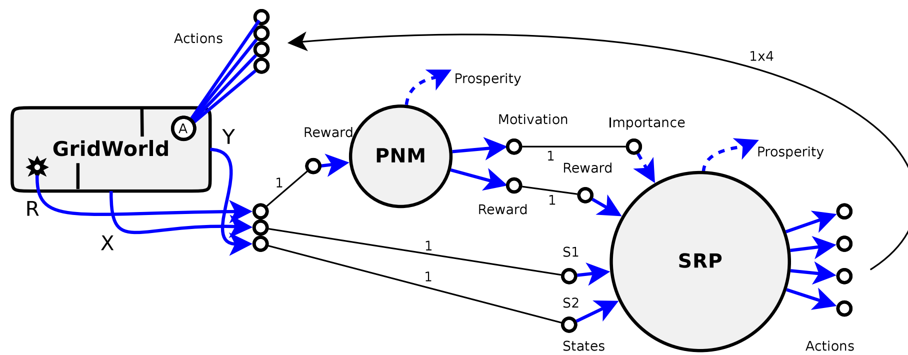


Figure 5.5: *Setup of experiment testing SRP Module with one source of reward and one Physiological Module. The physiology feeds the SRP with information about received reward and the current importance of efficient behavior. The SRP module receives information about the agent's position in the map and is allowed to move with the agent in four directions.*

The GridWorld publishes data about X and Y agent's position in the map. The map has size of 20x20, so there is 20 discrete values for each variable. The SRP Neural Module was configured to have two state variables, which are sampled into 20 values each. Since there are 4 actions allowed by the agent, the SRP was configured to have 4 outputs. This means that the SRP operates with 3-dimensional Q-matrix of size $20x20x4$, which stores Utility values for all actions in all states (see the Chapter 4.4). This configuration of input/output connections is sufficient to use the Module. The rest of all parameters is left with default values. Particularly, the default parameters for the SRP were: $\alpha = 0.5$, $\gamma = 0.3$, $\lambda = 0.04$, length of Eligibility trace $traceLen = 20$ and minimum randomization $\epsilon_{min} = 0.1$. Finally, the PNM was configured to have value of Decay $D = 0.01$ in this experiment.

5.1.2.2 Testing the Learning

In the non-episodic experiment, the agent was placed in the map on a random initial position and let to interact with the environment. At the beginning, the SRP generates random actions. When the source of reward is found, the system starts to learn a strategy how to obtain the reward from various positions. This is done by using the Importance-based Action Selection Mechanism (ASM) (see the Eq.4.20). During the low Importance of behavior, the strategy can be completely randomized. Compared to this, when the Importance is high, the SRP will use Greedy ASM (with predefined $\epsilon_{min} = 10\%$).

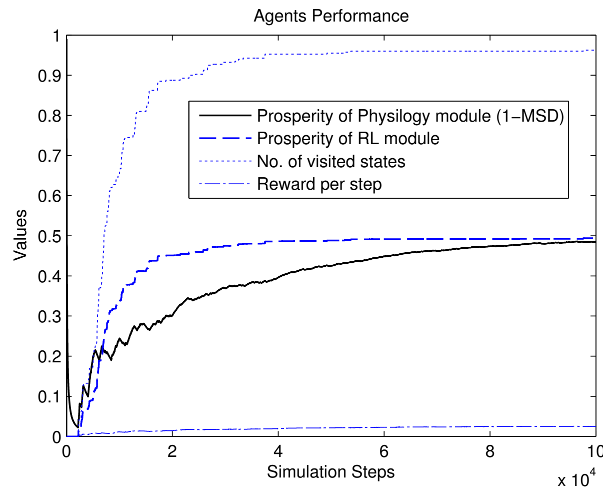


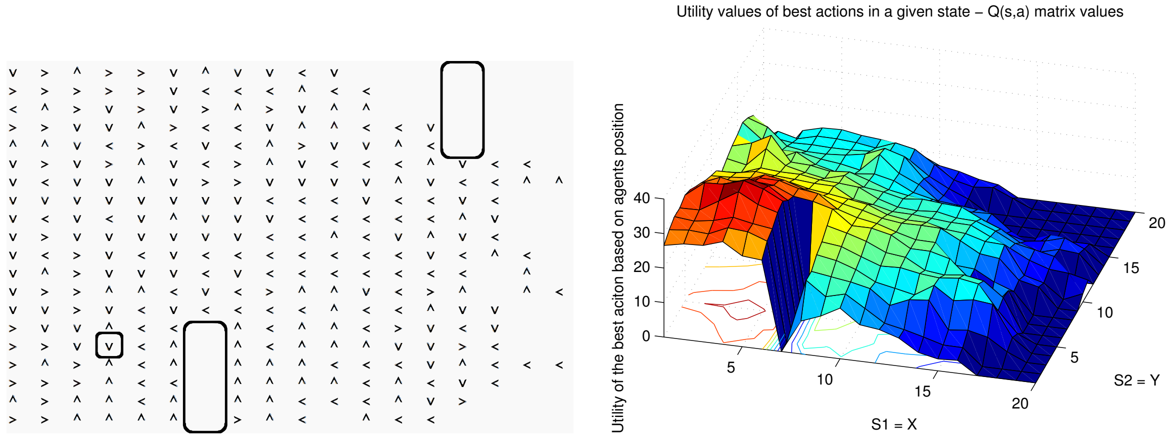
Figure 5.6: *Course of agent's learning during 100000 time steps of continuous simulation. After 40000 steps, almost all environment states were explored, but the agent continues to improve the learned strategy.*

The course of learning, together with the example of use of the NengoROS simulator can be seen in the video online[2]. The learning is depicted also in the Fig.5.6. The graph shows the course of Prosperity for both Neural Modules. For the SRP, also the *reward per step* and *no. of visited states* is depicted. It can be seen that after about 40000 simulation steps, the agent explored almost entire state space (excluding obstacles). From the course of Prosperity of the PNM it can be seen that the agent kept improving the learned strategy continuously. An example of learned strategy can be seen in the Fig.5.7, where the arrows represent agent's Greedy strategy. The environment contains one source of reward and two obstacles. It can be seen that after the simulation, the agent was able to navigate from almost any state towards the reward, while successfully avoiding obstacles.

[2]NengoROS simulator; course of learning, video online: `http://goo.gl/bY6Y2q`.

5.1.2.3 Influence of Physiology Decay on Agent's Behavior

The purpose of the PNM is to balance between exploration and exploitation. The balance is controlled by increasing the Importance of the behavior with a given dynamics. If the Importance rises slowly, the agent has enough time to explore. Currently, the exploration is implemented as random walk. This means that optimal dynamics of PNM should depend on a particular task. Bigger (or more complicated) environments will typically require slower dynamics than smaller environments. This experiment tests the expected influence of the Decay parameter of the PNM on agent's behavior.

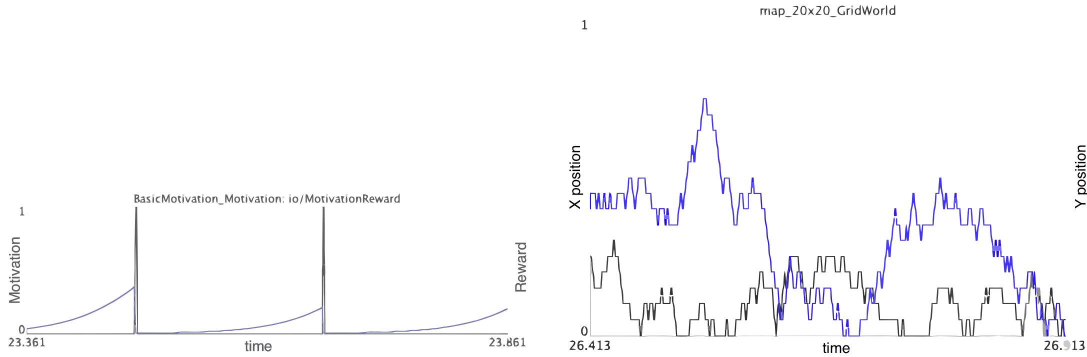


(a) *Visualization of agent's strategy that was learned during 40000 time steps of continuous simulation. There are two obstacles and one source of reward.*

(b) *Corresponding Utility values in a given state. The Z-axis shows the Utility value of the best action on the X, Y coordinates in the environment.*

Figure 5.7: *An example of learned strategy after 40000 simulation steps. On the left, there is a graphical representation of the Greedy strategy. Small arrows represent the action with the highest Utility value in a given state. So following arrows from any state represents the Greedy strategy. Positions without arrows represent places, where no action was learned. Graph on the right shows Utility value of the best action learned. It can be seen that the nearer the source of reward, the expected (discounted) reward is higher.*

Here are two examples of Decay set to $D_1 = 0.002$ and $D_2 = 0.01$ showing how the agent's behavior is influenced. The Fig.5.8 on the left shows how the amount of motivation corresponds with receiving rewards. The graph on the right shows the agent's positions in the map in time. If the speed of Decay is increased to $D_2 = 0.01$ per simulation step (see the Eq.4.1), the agent's motivation to reach the reward rises faster. This means that



(a) *Course showing slowly increasing motivation and binary events of receiving reward. With the increasing motivation, the agent increasingly tends to exploit the knowledge until the reward is reached.*

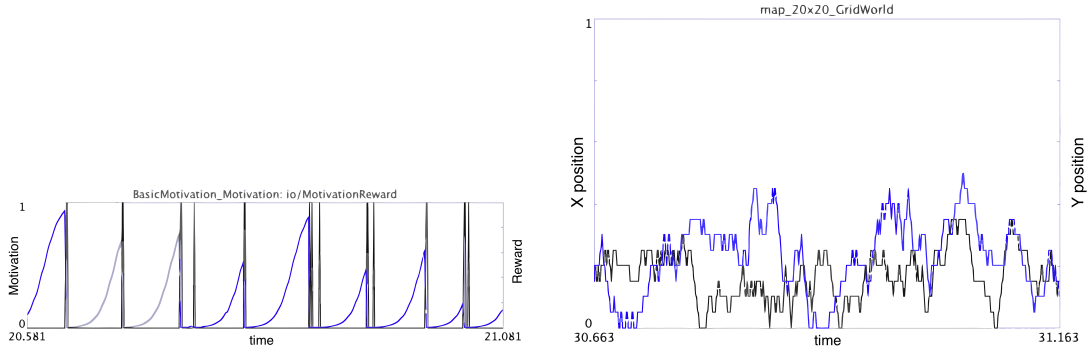
(b) *An example of agent's wandering around the environment in time. The X axis represents simulation steps, lines show current X and Y coordinates in the map.*

Figure 5.8: *Influence of Decay parameter of the Physiological Neural Module on exploration/exploitation ratio. Decay* $D_1 = 0.002$. *As the motivation for reward increases slowly, the reward is obtained with the corresponding frequency. The agent has time to explore. The graph on the right shows how agent moves through the environment.*

the agent tries to reach the reward more often and there is less time for exploration. This can be seen in the Fig.5.9 on the right. The average agent's position is nearer to the discovered source of reward at the coordinates $[X, Y] = [2, 3]$.

It can be concluded that the Decay parameter has the expected influence on agent's behavior. Also, the optimal value of this parameter depends on a properties of the environment (size, complexity etc.). In order to obtain good balance between exploration and exploitation of knowledge, this parameter has to be set correctly.

Recapitulation This Section shown an architecture composed of more complex Neural Modules. One of the Modules has inner dynamics, while the other one implements learning/adaptation mechanism. The SRP Module serves as an example of typical Neural Module. It continuously receives data on inputs, learns from them and produces some useful information/behavior on own outputs. The functionality of SRP and PNM was tested and shown in a discrete environment. The domain-configured SRP was able to successfully learn from interaction with the environment. With the help of PNM, the agent was able to successfully balance between knowledge exploration and exploitation. This hand-designed architecture will be compared to



(a) *Course showing slowly increasing motivation and binary events of receiving reward. It can be seen that receiving the reward correlates with amount of motivation.*

(b) *The corresponding positions in the map in time. The agent tends to stay near the reward source on 2,3.*

Figure 5.9: *Influence of Decay parameter of the Physiological Neural Module on exploration/exploitation behavior. Decay* $D_1 = 0.01$*. Compared to the Fig.5.8, the agent tends to stay nearer the source of reward - less exploration.*

the architecture designed by various configurations of Evolutionary Algorithms in the next Chapters.

5.2 Evolutionary-Based Design of Architectures

One of the main goals of this Thesis was to design a framework, which can unify the communication between subsystems of very different nature. The common communication interface then allows us to automatize design of new architectures by (e.g.) some type of Evolutionary Algorithm. Here, an architecture and its functionality is defined by set of Neural Modules and connections between their inputs/outputs. Since the space of all possible connections between all Neural Modules implemented would be too big to be searched, the following experiments assume the following constraints. The set of Modules is predefined and space of all possible connections between Modules is constrained to feedforward topologies. Also, if a Neural Module has configurable properties of inputs/outputs, these set to chosen values before the evolution.

The following Section will describe the common setup of Evolutionary Algorithms that are

used in the experiments. In each experiment, only table with parameters of a particular algorithm will be mentioned.

5.2.1 Evolutionary Algorithms Used

This section will hold a brief description of Evolutionary Algorithms (EAs)[3] that were used in the following experiments. There were two types of algorithms used here:

Genetic Algorithm (GA) - is a standard type of EA, which operates over binary vectors (as genomes) of predefined length.

Real-Valued Genetic Algorithm (RGA) - is a modification of standard GA. The only difference is that RGA operates with vectors of real-valued numbers (genomes). This requires special operators: crossover and mutation, which will be described below. Note that there may be more suitable algorithms for operating with real-valued vectors (such as Evolutionary Strategies (ES) (Hansen and Kern 2004)), but in order to keep both algorithms used simple and similar, the RGA was used here.

Table 5.1: *Typical parameters of GA and RGA used.*

$PopSize$	$MaxGens$	P_{mut}	P_{cross}	$Elites$
50	80	0.05	0.8	1

The Algorithm 5 shows that a standard generational model of EA was used in both cases here. The evolution of population $PopSize$ is constrained to maximum number of generations $MaxGens$ with Elitism set to $Elites = 1$ (which explicitly preserves only the best individual found so far).

The standard Roulette-wheel selection was used here. In both cases, one-point crossover of genomes was applied with the probability of P_{cross} for each pair of selected individuals. The mutation is applied with the probability P_{mut} for each gene. In case of GA, simple bit-flip is used. In case of RGA, the mutation was implemented as sampling the Gaussian Function with the standard deviation of $\sigma = 1$ with mean value of $\mu = gene_i$. After applying mutation, the value is corrected to be in the predefined interval of $gene_i \in$

[3]Note that the EA is used to denote Evolutionary Algorithm in general term, that is: superclass of GA and RGA. While GA and RGA denote particular algorithms described in this section.

```
   Data: Randomly initialized population Pop_0
   Result: Genome representing the best architecture
 1 for i ∈ 0,...,MaxGens do
 2     for ind ∈ 0,...,PopSize do
 3         ind_i.fitness = ind_i.genome.evaluate();    // evaluate all (simulate for N steps)
 4     end
 5     Pop_{i+1}(0,..,Elites − 1) = getNbestIndsFrom(Pop_i, Elites);          // apply elitism
 6     while Pop_{i+1} not full do
 7         select two individuals based on their fitness
 8         crossover them with P_cross ;                        // apply evolutionary operators
 9         mutate each of their genes with P_mut
10         place them into Pop_{i+1} ;                            // fill the target population
11     end
12 end
```

Algorithm 5: Generational model of Evolutionary Algorithm used for neuro-evolutionary design of agent architectures. The algorithm is common for both GA and RGA, which operate over vectors of boolean/real-valued variables. Single-valued fitness is obtained as agent's performance during the simulation of predefined duration of N steps.

$\langle minGene, maxGene \rangle$. The Table 5.1 shows typical values of GA and RGA used in the following experiments.

5.2.2 EA-Designed Agent Controllers - Navigation Task

Here, an experiment showing EA-based design of simple agent architecture is described. The main principles of the approach will be described here. This architecture is similar, and can be directly compared to the one described in the Section 5.1.1. Here, the objective is to automatically design an architecture which navigates in the environment (see the Fig.5.1) as fast as possible.

5.2.2.1 Architecture Design

The architecture is composed of predefined set of Neural Modules connected in a given topology. In this case, it is composed of the same two Neural Modules as described in the Fig.5.2. Similarly to the hand-designed architecture, the Modules are ordered in a feed-

forward fashion. All inputs/outputs are fully connected between layers. The approach is similar to classical neuro-evolution of feedforward ANNs, therefore the RGA is used for finding connection weights between these Modules.

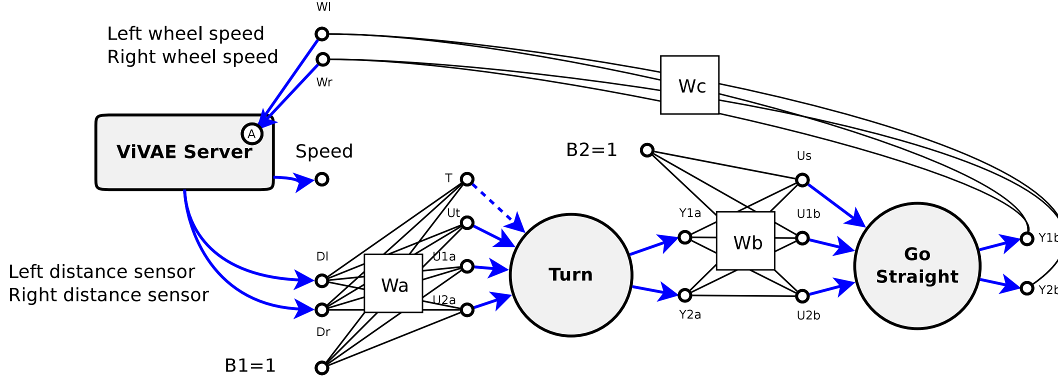


Figure 5.10: *Principle of encoding of a hybrid agent architecture into genome. The topology is represented as a feedforward hybrid network, where inputs/outputs between layers are fully connected. The RGA is supposed to optimize the connection weights in order to obtain the highest average speed in the simulation.*

The architecture is composed of identical Neural Modules as the one described in the Section 5.1.1. The principle of encoding of the architecture into a genome is depicted in the Fig.5.10. The genome is a vector of 25 real-valued numbers from the interval $\langle -1, 1 \rangle$. In the Fig.5.10, the genome is divided into three matrixes W_a, W_b, W_c of different sizes, where each matrix defines connection weights between layers. These matrixes are placed into a single vector. The length of a genome is then given by sizes of these matrixes as follows:

$$\begin{aligned} l &= mn + op + qr,\ W_a \in \mathbb{R}^{m \times n}, W_b \in \mathbb{R}^{o \times p}, W_c \in \mathbb{R}^{q \times r} \\ l &= 3 \times 4 + 3 \times 3 + 2 \times 2 = 25. \end{aligned} \tag{5.6}$$

Table 5.2: *Parameters of RGA used in the navigation task.*

$PopSize$	$MaxGens$	P_{mut}	P_{cross}	$Elites$
50	80	0.05	0.8	1

For each genome (genotype), the architecture (phenotype) is built and placed in the simulation. The agent is then allowed to move in the environment for given number of

simulation steps. The quality of the individual (fitness) is computed as an average speed during entire simulation, that is:

$$f = \frac{\sum_t v_t}{T}, \; t \in 0, \ldots, T, \tag{5.7}$$

where T is the number of simulation time steps and v_t is a current speed of the agent, which is obtained from one output of the ViVAE server Module (see the Fig.5.10). The parameters of RGA were empirically set to the values shown in the Table 5.2.

5.2.2.2 Resulting Automatically Designed Architectures

The performance of the resulting architecture can be seen in the video[4]. From the real-time graphs, it can be seen that the strategy used by the agent is completely different, despite the fact that it uses the same Neural Modules. Even the best architectures found by the RGA had slightly lower fitness than the hand-designed one. This is caused by the fact, that the best RGA-designed agents use the strategy: go straight ahead, turn in the grass and navigate in the opposite direction. The lower fitness was caused by changing the direction of travel at the beginning of the simulation. Despite the lower fitness value, the graphs show that the performance of RGA-designed architectures is better and more robust (the agent is able to recover from crashing into the obstacle) than the hand-designed one (without ability of the crash recovery).

Another interesting strategy designed by the RGA can be seen in another video[5] on the bottom. From the *Summator* (equivalent of the *Go Straight* module in the Fig.5.4) real-time graphs, it can be seen that one output of the *GoStraignt* module is connected to wheel with negative weight. The agent applies high signal to wheels and decelerates towards the next turn .

Recapitulation This experiment presented and tested a method of encoding simple hybrid feedforward agent architecture into genome used by the RGA. It was shown how the RGA is able to automatically design architecture suitable for a given task. Important is the fact, that the RGA was able to use given set of Neural Modules in a different way than anticipated. Moreover, the best resulting architectures perform slightly better than the human-designed one. This shows that the HANNS

[4]Behavior of typical best RGA-designed architecture, video online at: `http://goo.gl/85dpo0`.

[5]Two examples of RGA-designed architectures - navigation task, online at: `http://goo.gl/cSbNAR`.

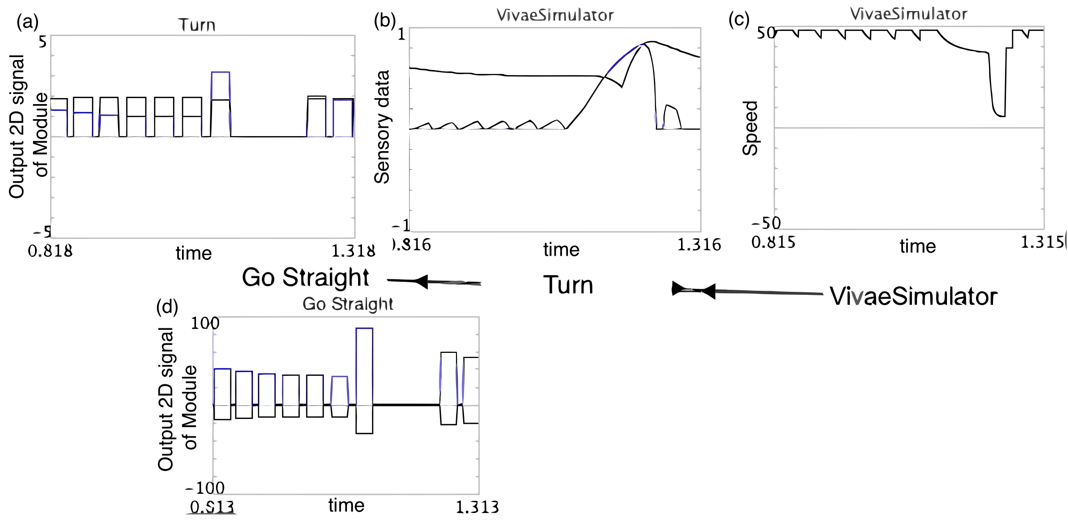


Figure 5.11: *Example of behavior of RGA-designed architecture for navigation in the maze. Screenshot from the NengoROS simulator during navigation through the left turn. Big "peak" in (d) - corresponds to the entering the turn. After the peak, the "Turn" Module produces signal with low values (which is transformed by weighted connections to robot's wheels). After going out of the turn, the agent follows data on the left sensor. Data of the left sensors can be seen in the (b), where the signal with starting lower value is the left sensor. Graph on the (c) shows current speed. Graph (a) shows output values of the "Turn" Module. The behavior is different from the behavior of the hand-designed architecture, shown in the Fig.5.4.*

framework is able to use known sub-systems (represented as Neural Modules), for example algorithms in new and potentially unknown ways.

5.2.3 Artificial Neural Network of 3rd gen. vs. Hybrid Network

This experiment briefly shows the ability of the NengoROS to employ two different systems into one architecture. The first is Neural Module and the second part is Spiking Artificial Neural Network (SNN), which is supported by the original Nengo simulator. More information about on the SNN support can be found online[6] or in the Appendix A.2.

[6]More information about the original Nengo simulator can be found online at: `http://nengo.ca`.

This experiment is also a simple demonstration of one of the main principles of the HANNS. The user does not need to know exact description of the task. Often, the user *knows some information* about the domain, which can be set as starting point (by defining which Neural Modules should be used) for automatic design of the HANNS. It is shown how this starting point can greatly improve systems's performance and speed of searching for the solution.

5.2.3.1 Task Description

The evolutionary design has the following task here: to approximate behavior of the fuzzy-logic gate implementing Łukasiewicz weak disjunction: $y = max\{x, y\}$, where x and y are values of membership function. The two approaches are tested and the results compared, the first experiment tries to approximate the function by finding weights in the fully recurrent Spiking Artificial Neural Network (SNN). In the second experiment, the same network is used, but the $Fuzzy - OR$ Module is also added to the network as a domain knowledge.

Since the main focus in the ALife domain is to evaluate systems on streams of data produced by the environment, a similar approach is used here. Evaluation of performance of EA-designed systems is not done on a predefined test dataset. Instead, the signal generator is used. It produces pseudo-random 2D signal[7], which is fed into both: the designed system and the *Plant* (Fuzzy-OR gate) to be approximated. The principle is shown in the Fig.5.12. The Mean Squared Error (MSE) is computed during N simulation steps and its inverse is set to be the *fitness value* of the individual.

Here is shown how a problem can be solved by RGA-designed SNN. The performance of a SNN-based system is then compared with the hybrid system. The hybrid system has the same setup, but it features also one added $Fuzzy - OR$ gate.

Table 5.3: *Parameters of RGA used for evolving the (Hybrid) SNN.*

$PopSize$	$MaxGens$	P_{mut}	P_{cross}	$Elites$
30	100	0.05	0.8	1

Here will be described common setup for both experiments. Evaluation of one individual took $N = 3000$ simulation steps. Particularly, the simulation was simulated for $t = 3$

[7] Information how the pseudo-random signals are generated in Nengo: `http://goo.gl/VWqKh1`.

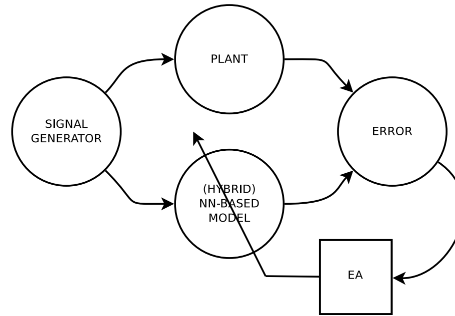


Figure 5.12: *Principle of EA-based optimization of connection weights. The generator generates 2D signal, which is fed to both, the Plant and the optimized model. Average error across the N simulation steps is computed and its inverse is set as the fitness value (Vítků and Nahodil 2013).*

seconds with resolution $dt = 0.001$ (since the SNNs require explicit representation of time). The generator was set to generate preudo-random continuous signal from the interval $\langle -2, 2 \rangle$.

The setup of RGA is described in the Table 5.3. Allowed values of genes are from the interval $gene_i \in \langle 0, 1 \rangle$. Based on the fact that no negative weights were allowed, we can see that no inhibitory neurons are simulated here and no negative signal will be produced by the network.

5.2.3.2 Optimization of ANN-Based Model

Here, the first experiment will be described. The evolution has task to optimize weights in the fully recurrent SNN. The network contains 4 neurons of 3rd generation. These neurons communicate by means of spikes between each other. Particularly, the Leaky-Integrate and Fire (LIF) neuron model is used. Based on steady input current to the soma, the LIF neuron may produce spikes on its output - axon. The "tuning curve" defines dependency of firing rate on input current to the soma (see Figs.A.2 and A.4 in the Appendix A.1). Aside of spiking neurons, there are also two input neurons and one output neuron. The schematics can be seen in the Fig.5.13 (all without dashed lines).

The two input neurons (marked as $F2S$) convert continuous input values to series of spikes. Oppositely, the output node (marked as $S2F$) converts the series of spikes into real-valued output. The $F2S$s are in fact LIF neurons, where the real-valued input is the current to the soma. The $S2F$ node serves as a post-synaptic current integration: the

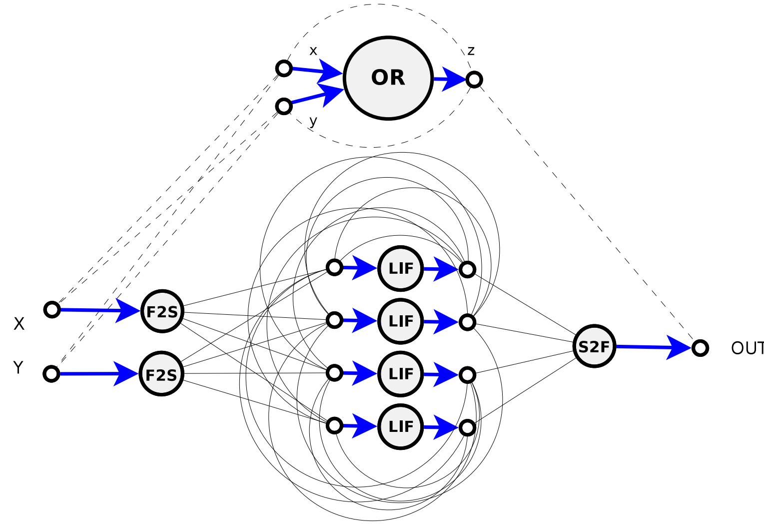


Figure 5.13: *Optimized connections in the (hybrid) model. The solid thin lines correspond to optimized weights of the fully recurrent SNN (dashed connections are set to weight=0). In the hybrid network, also the dashed lines are optimized, which enables the evolution to use also the Fuzzy-OR node in the system.*

.

higher the firing rate (leaky integration), the higher output value (see e.g. Chapter 4.1.3 in (Gerstner and Kistler 2002)). Note that intercept[8] values for all neurons used are set to value $int = 0$ and max. firing rate is set to $mr = 25$. This means that no negative values are represented/processed by the SNN.

Four hidden LIF neurons are fully connected. The topology with entire SNN is depicted in the Fig.5.13. Weights of all thin solid black lines are optimized by the evolution. The length of entire genome is then computed as:

$$l = D_{input} \times N_{hidden} + N_{hidden}^2 + D_{out} \times N_{hidden} = 8 + 4^2 + 4 = 28, \tag{5.8}$$

where D_{input} and D_{output} are dimensions of input or output respectively and N_{hidden} is number of hidden neurons. The evaluation of the fitness is then computed as follows:

$$f = \frac{1}{MSE} = \frac{1}{\sum_t (out_m(t) - out_p(t))^2}, \; t \in 0, ..., N, \tag{5.9}$$

[8]Intercept and max. firing rate are explained e.g. online at: `http://nengo.ca/docs/html/configuring.html`.

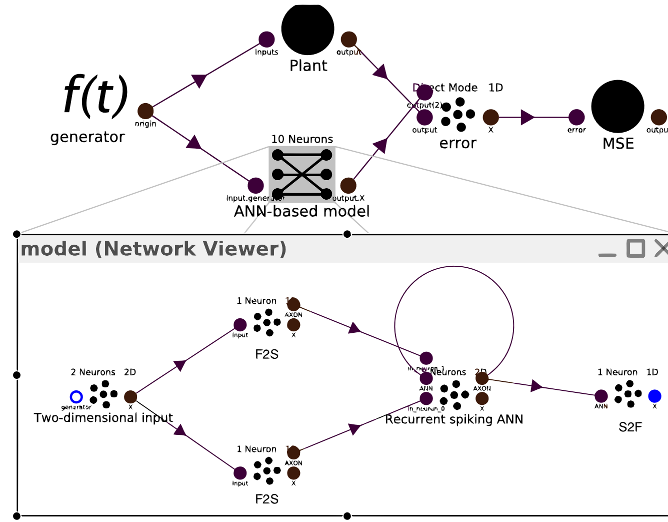


Figure 5.14: *Example of hybrid modular system in the NengoROS GUI. Generator $f(t)$ generates 2D signal which is def into the ANN-based model and Plant. The error between two signals is obtained and the MSE of entire simulation is computed. Screenshot from the NengoROS simulator. The Model is represented as a sub-network in the Nengo GUI.*

where the $out_m(t)$ is the output of the SNN and $out_p(t)$ is the output of the system to be approximated - the $Fuzzy - OR$ module heres.

The Fig.5.14 shows an example of connecting the fully recurrent SNN with the rest of the experiment. The $f(t)$ generates 2D signal, which is fed into the plant and ANN-based model. Finally, the MSE is computed from the instantaneous error online during the simulation. The Fig.5.15 shows the course of the evolution - fitness of the best individual in the population. It can be seen that in the generation 50, the fitness converged to the value around $f = 40$. This means that the MSE of the best individual is around $MSE = \frac{1}{40} = 0.025$.

From the Fig.5.16 and video[9] we can see that the typical best evolved SNN can approximate the plant relatively well. The output of the SNN implements rather the equation $out = a + b$, than the $Fuzzy - OR$ operation. But, due to nature of the input signal, this causes relatively small MSE.

[9]Example of typical best SNN approximating Fuzzy-OR available online at: `http://goo.gl/HIcBY4`.

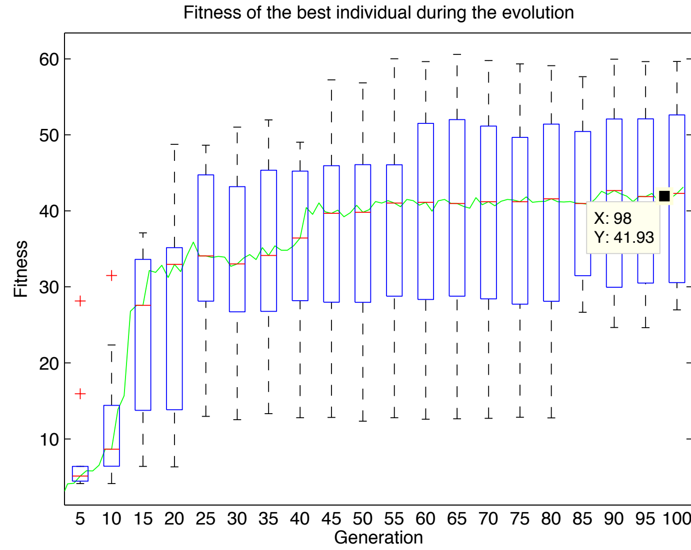


Figure 5.15: *Course of evolution of the ANN-based system. The fitness of the best individual is shown. After around 50 generations, the fitness converged to the value of f=40. The shown course is averaged across the 10 runs of RGA.*

5.2.3.3 Optimization of Hybrid Model

The results of evolved SNN are compared to the identical system. Only in this case, the a-priori knowledge about the task was added in form of the $Fuzzy - OR$ Neural Module. The resulting hybrid network is depicted in the Fig.5.13. In this case, both solid and dashed thin lines are optimized. This means that the length of genome is increased from 28 to $l = 35$, which increases the space to be searched by the evolution.

Note that this simple example of hybrid system combines both: the spiking and real-valued types of communication between Modules. The graph in the Fig.5.17 shows the course of evolution of such a hybrid network. It can be seen that the evolution was able to produce significantly better approximation of the plant than classical ANN-based model. Fitness value of the best typical individual converged on the value about $f = 70$, which corresponds to the $MSE = 0.014$ Moreover in some cases, the evolution was able to find much better individuals with the value of $MSE = \frac{1}{f} = \frac{1}{570} = 0.0018$. This means that the evolution of the hybrid model was able to approximate the plant with almost twice better average precision than the solution based on the SNN. In some cases, the solution found was almost perfect (almost only the $Fuzzy - OR$ Module was connected to the inputs/outputs of the system).

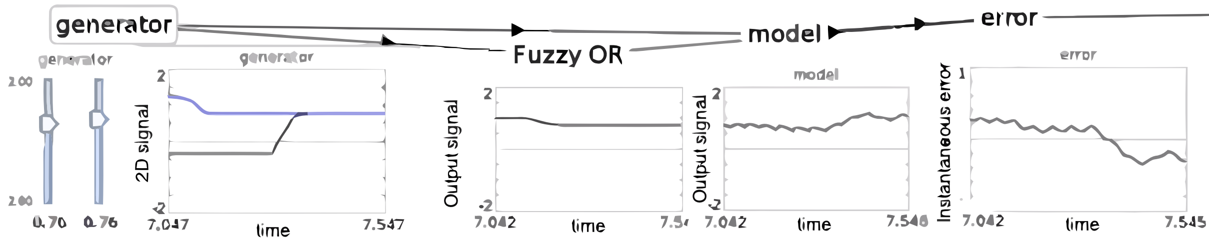


Figure 5.16: *An example of performance of the ANN-based system. From the left: graph ("generator") shows the 2D input signal, the "Fuzzy OR" shows the signal computed by the plant, the "model" shows output signal of the SNN and finally instantaneous error between two signals. It can be seen that the SNN approximates rather the sum of input signals.*

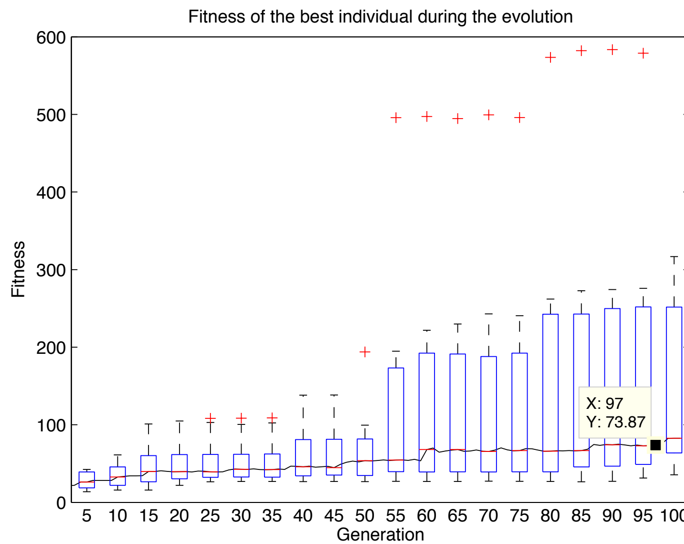


Figure 5.17: *Results of agent controlled by hybrid ANN.*

Recapitulation This experiment shown how the NengoROS simulator can be used for simulating hybrid systems, which even employ different types of communication. It also shown how the simulator can be used for evolutionary-based optimization of connection weights in these systems, in order to approximate the desired behavior. It was also shown that the hybrid system with added a-priory knowledge (in form of correctly selected set of Neural Modules) can greatly enhance the speed of evolution and the accuracy of the solution obtained. Here, the evolution was able (in some cases) to correctly identify that for the best approximation, mainly the $Fuzzy-OR$ node should be connected to both inputs and one output.

5.2.4 EA-designed Agents with Motivation-Driven RL

This experiment shows EA-based design of an architecture which implements motivation-driven Reinforcement Learning. The agent is situated in the same environment as in the experiment in the Section 5.1.2, only here the map is slightly smaller: 15×15, but with the same relative placement of objects. Again, the goal of the agent is to be able to learn how to obtain the reward in the environment. The experiment shows the ability of the evolution to design architecture that are suitable for the task. Experiments comparing the RGA and GA were made. Also, various definitions of the fitness function are tested and discussed.

5.2.4.1 Architecture Design

Here, the architecture design and its representation for the EAs will be explained. This architecture has the same goal and uses the same Neural Modules as the one shown in the Fig.5.5. It contains one Physiological Neural Module (PNM) and one Stochastic Return Predictor (SRP) connected in a closed-loop with the GridWorld simulator.

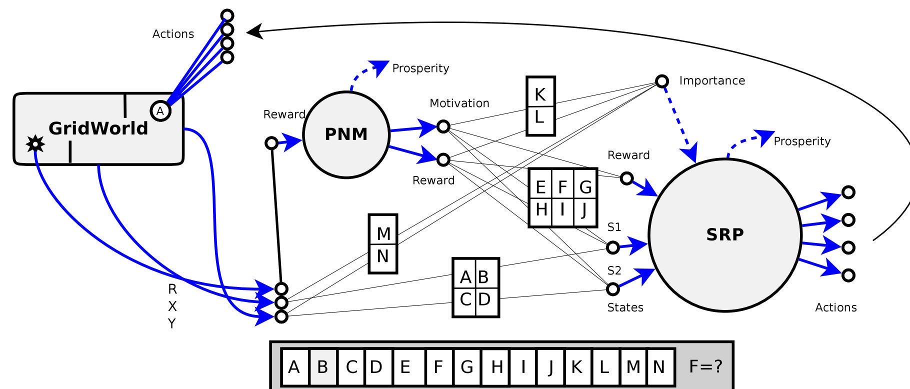


Figure 5.18: *Principle of encoding of hybrid agent architecture as a feedforward Hybrid Neural Network (HNN). The connection weights between Modules are optimized by the evolution, with the goal of providing the desired behavior.*

The Fig.5.18 shows how the architecture design is encoded in the genome, which can be used by the RGA. There are some simplifications, such as that the PNM is placed in the layer 0. Also, the connections between the SRP and the GridWorld (actions) are hardwired in a desired way. Again, the genome is divided into several sub-matrixes, where each one represents connections between nearby Neural Modules. Both Modules, the SRP

and the PNM are configured identically to the experiment in the Section 5.1.2. This allows us to compare the from the the RGA and GA with the hand designed architecture.

For better explanation of the Fig.5.18, the equations for each transformation will be shown. For the state inputs:

$$\begin{aligned} S_1(t) &= AX(t) + BY(t) + FM_{PNM}(t) + IR_{PNM}(t) \\ S_2(t) &= CX(t) + DY(t) + GM(t) + JR(t), \end{aligned} \tag{5.10}$$

where $M_{PNM}(t)$ is the value of Motivation and $R_{PNM}(t)$ the value of reward produced by the PNM at time step t. Equation for the Importance input is similarly:

$$I(t) = KM_{PNM}(t) + LR_{PNM}(t) + MX(t) + NY(t), \tag{5.11}$$

and finally the Reward input of the SRP Module:

$$R_{SRP}(t) = EM_{PNM}(t) + HR_{PNM}(t). \tag{5.12}$$

5.2.4.2 Defining the Agent's Goals

As discussed earlier in the text, it is difficult to ***generalize what the architecture should do***. That is: to find the domain independent evaluation of agent's desired abilities. This Section will test methods of evaluation of agent's performance that were proposed in the Section 3.3.3.6. The two proposed approaches of evaluating the agent's performance will be tested, mainly:

All of the Modules should be used as efficiently as possible. For the purpose of evaluating the effectiveness of the Module usage, each of the Modules should publish own value of the Prosperity function.

Architecture fulfills predefined inner needs. This can be represented for example by hard-wired PNM, which then produces need for reward. In this case, the Prosperity of the PNM also evaluates the agent's ability to fulfill the need.

Table 5.4: *Parameters of GA and RGA used for RL-based architectures.*

PopSize	*MaxGens*	P_{mut}	P_{cross}	*Elites*
50	80	0.05	0.8	1

This experiment compares the first two methods with each other. For each method, own definition of the fitness function is defined and used. For each method, the courses of evolution of the GA and RGA are compared.

Here, the common **setup of all experiments in this section** will be described. The parameters of RGAs and GAs were empirically set to values shown in the Table 5.4. From the Fig.5.6 it was estimated that a well-designed architecture should be able to learn useful strategy in 20000 simulation steps (even on a bigger map). Therefore evaluation of the fitness value for each agent is determined from the non-episodic simulation of 20000 steps.

5.2.4.3 Composed-Objective Fitness

In the first experiments, the "Composed-Objective Fitness" was defined according to the first possibility described above. This means that every Neural Module evaluates own Prosperity - *"how well is used in the simulation"* and the quality of the architecture is composed of these Prosperities. Here, the "Composed Fitness" (CF) is defined as a normalized sum of Modules' Prosperities. The overall prosperity of the architecture is:

$$P_{arch} = CF_{arch} = \frac{\sum_i P_i}{N}\ ,\ i \in 1, 2, \ldots, N, \tag{5.13}$$

where N is number of all Neural Modules in the architecture. Note that this problem is of multi-objective nature and therefore the Multi-Objective EA (MOEA) (Deb 2011) could be required to find the global optimum efficiently. In order to keep the evolutionary part simple, this (single-objective) definition of Composed Fitness CF was tested.

The Fig.5.19 compares the course of the best fitness in the population during the run of EAs with Composed-Objective fitness (CORGA) and the GA (COGA). It can be seen that similarly good solution has been found much faster by the COGA. This shows that finding binary weights is less complex task in this case. The Table 5.5 on the left shows typical genome of the best solutions (for both, GA and RGA). The resulting genome and fitness is compared with the hand-designed architecture.

Table 5.5: *Comparison of typical agent architectures' resulting genomes and their fitness values. Left: hand-designed, CORGA-designed and COGA-designed. Right: examples of SOGA and SORGA-designed agents (which are described in the Section 5.2.4.4).*

See Fig.5.18	A	B	C	D		
	E	F	G	H	I	J
	K	L	M	N		
Hand-designed $CF = 0.555$ $SF = 0.625$	1	0	0	1		
	0	0	0	1	0	0
	1	0	0	0		
COGA - best Ind $CF = 0.494$	0	1	1	0		
	1	0	0	1	1	0
	0	1	0	0		
CORGA - best Ind $CF = 0.491$	0.06	1	1	0		
	1	0	0	0	0	0
	0	1	0.23	0		

SOGA - Ind1 $SF = 0.699$	1	0	0	1		
	0	0	0	1	0	0
	1	1	1	1		
SOGA - Ind2 $SF = 0.697$	1	1	0	1		
	0	0	0	1	0	1
	1	1	1	1		
SORGA - Ind1 $SF = 0.745$	0	1	1	0.71		
	0	0	0	1	1	0
	0.89	0	1	1		
SORGA - Ind2 $SF = 0.723$	0	0.51	1	0		
	0	0	0	1	0	0
	0.76	0	0.3	0.44		

The Fig.5.20 describes the course life of typical best agent found by the COGA (CORGA behavior is similar). The network is designed in such a way, that Prosperity of the SRP Module is maximized, while the prosperity of Physiology is ignored. This produces relatively high fitness around CF=0.5. Number of visited states suggests that the ASM here implements random strategy. From the Table 5.5 we can see that the state variables are represented well (only exchanged X and Y), but the Motivation output of the PNM is not connected to the reward input of the SRP. Also, both typical best agents have the common value of weight $E = 1$, causes the SRP to receive reward every time step, proportional to the value of the Motivation:

$$R_{SRP}(t) = E \times Motivation_{PNM}(t) + H \times Reward_{PNM}(t). \tag{5.14}$$

It can be seen that when the Physiology receives reward, the *Reward per step* value of the SRP Module temporarily decreases. The Fig.5.20 on the right shows Utility values for the best actions in the map. It can be seen that almost all visited states have the same utility values, only during receiving the (real) reward the utility decreases. We can see that the architecture does not learn any useful strategy.

This experiment shown that the definition of CF as in the Eq.5.13 is not suitable for this task. The evolutionary search can stuck in local optimum (maximizing the prosperity of only some Modules), which may not create architecture capable of required behavior. The SRP has two main goals: to receive reward (on its input) as often as possible while

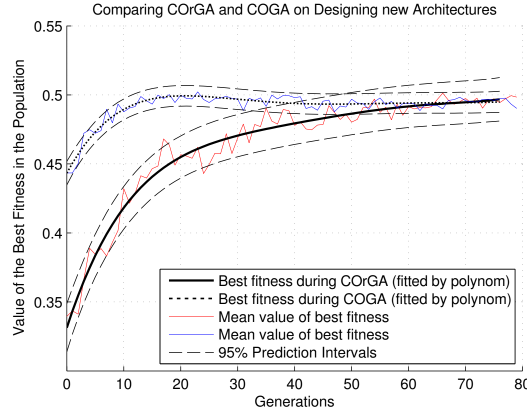


Figure 5.19: *Evolutionary design of (hybrid) ANN agent architecture - comparing performance of the Composed Objective EA and GA. The values are averaged from 10 runs of EA (GA) and fitted by the polynomial.*

maintain exploration as high as possible. These requirements were obtained by completely random strategy with the source of reward connected to the Motivation output of the PNM. This caused that the agent *learned rather how to avoid the reward*, rather than to obtain it.

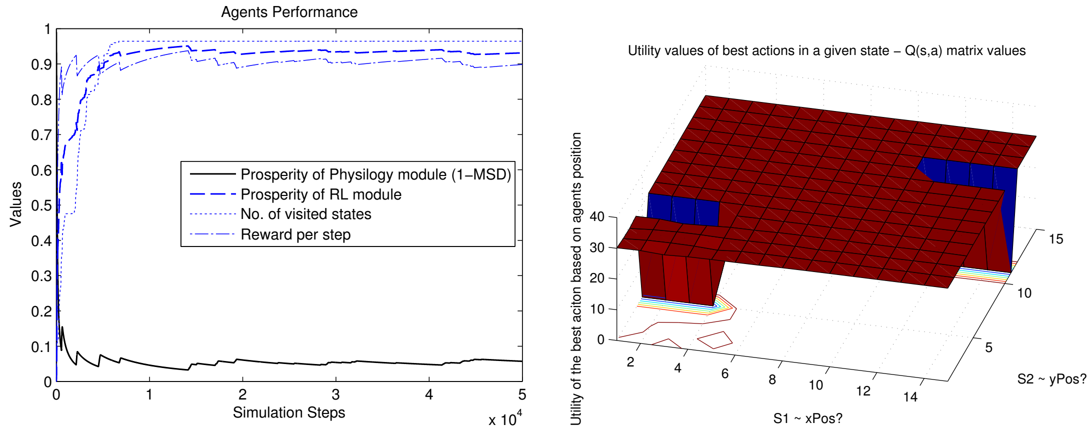
5.2.4.4 Single-Objective Fitness

Since the Composed-Objective fitness as defined in the Eq.5.13 was not suitable for this task, the Single-Objective fitness function (SF) was defined. This SF is an example of second type of evaluation as described in the Section 5.2.4.2 - it evaluates how well is the agent able to fulfill some predefined needs. The SF is defined as the prosperity of the PNM (see the Eq.4.3) of the agent:

$$SF_{arch} = P_{PNM} = 1 - MSD. \tag{5.15}$$

This means that the Single-Objective RGA/GA (SORGA/SOGA) has the goal ***to build the architecture which minimizes agent's need*** for a given resource (represented by the reward). It is "up to the EA's decision" whether: the task requires use of other Modules (e.g. learning in this case), how to use those Modules (e.g. how to represent informations in the SRP in this case) etc.

The experiment setup (parameters of RGA, GA, length of simulation etc) is the same as



(a) *Course of life of typical best agent found by the COGA (CORGA is similar). The prosperity of PNM is completely ignored. No. of visited states suggests that the ASM here implements random strategy. Reward per step is high due to connecting reward input of SRP Module to motivation output of the Physiology.*

(b) *Content of memory of the agent designed by the CORGA. Despite the fact that X, Y coordinates are represented correctly (axes only swopped), the Q-Lambda algorithm clearly does not learn any useful strategy. The module receives reward almost each step (if the motivation is high).*

Figure 5.20: *Description of behavior and knowledge of typical best agent produced by the CORGA and COGA algorithms. It can be seen that the behavior is not expected: the PNM does not receive practically any reward. The reward connection of the SRP is connected to other sources than the actual reward, therefore it does not learn any useful strategy (receives reward almost each simulation step, regardless the action produced).*

in the previous case. The Fig.5.21 shows the course of evolution for both, SORGA and SOGA. Compared to the previous experiment with CF, both algorithms converge faster. Again, the GA finds similarly fit solution considerably faster than the RGA.

The Table 5.5 shows two typical solutions found by the SORGA and SOGA, the following part will describe the results from the SORGA and SOGA separately.

SOGA-designed architectures. Both individuals in the Table 5.5 have the following properties:

- Reward input of the SRP Module is wired correctly, so that the RL part works as expected.
- Both environment ***state variables and motivation output of the PNM***

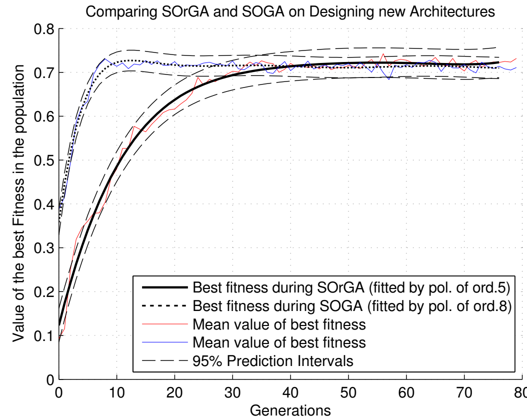


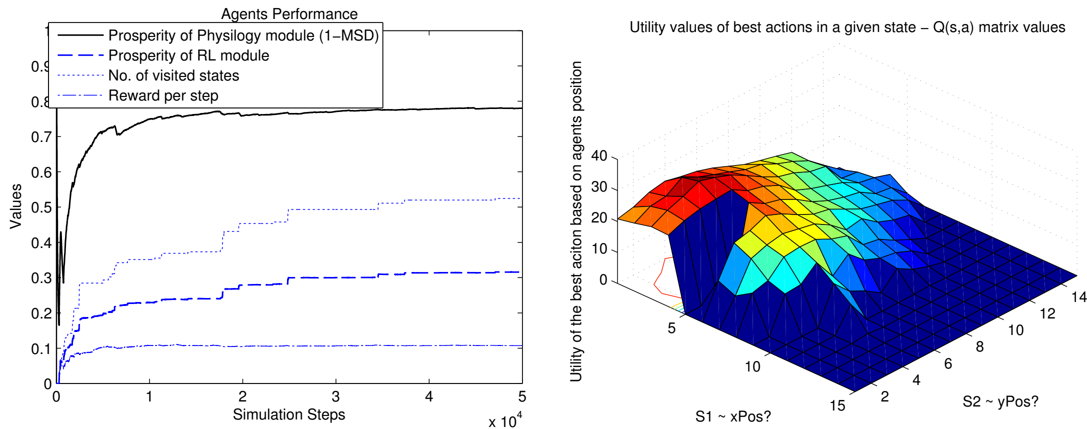
Figure 5.21: *Evolutionary design of (hybrid) ANN agent architecture - comparing performance of the Single-Objective EA and GA. Again, the GA finds the similar fitted individual considerably faster.*

are connected to the Importance input of the SRP Module. This means that an uncommon type of motivation-driven RL is used here. The *Importance is directly proportional* to the motivation produced by PNM and *to the agent's position/distance from the reward source.*

- Binary reward output of the PNM is connected to the motivation source too, but this has no significant influence on agent's behavior.

Also note that the $Ind1$ has correctly wired state variables to agent's position, while the $Ind2$ has one dimension "diagonalized" (both, X and Y coordinates are connected to the $S1$ variable). This means that only one half of the $Q(s, a)$ matrix was used here, but still the architecture performed relatively well. The Fig.5.22 shows behavior of a typical best architecture found by the SOGA. The architecture performs similarly to the hand-designed one (see Fig.5.6). Compared to the hand-designed, this one ***has bigger motivation to stay near the reward source***, so that the overall prosperity of the PNM is higher, while the amount of explored states is lower.

SORGA-designed architectures. Compared to the SOGA, finding new architectures by means of the SORGA required more generations. But the SORGA has wider possibilities how to weight signals between particular Modules in the network. For instance, from the Table 5.5 it can be seen that the $Ind2$ used only half of the $Q(s, a)$ memory. The $Ind1$ uses swopped coordinates with one dimension slightly



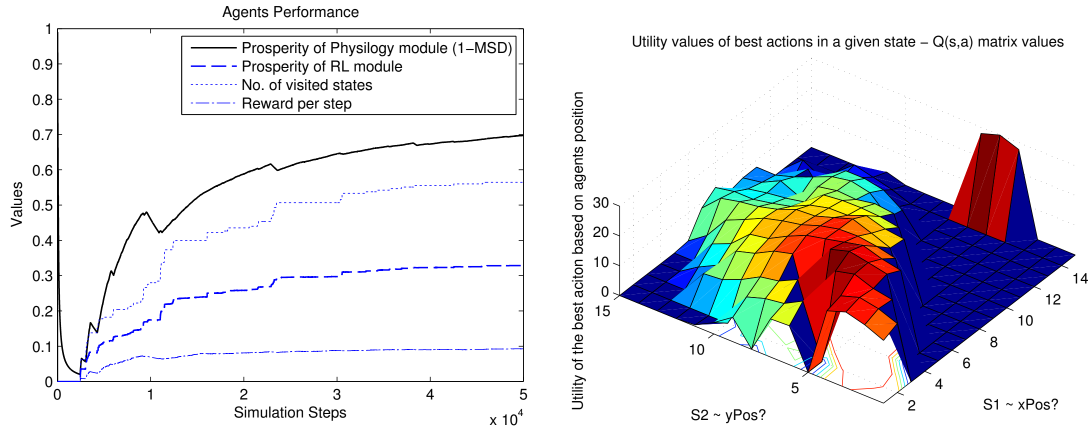
(a) *Course of life of typical best agent found by the SOGA - Ind1. The behavior is similar to the hand-designed architecture (see Fig.5.6).*

(b) *Knowledge of selected agent found by the SOGA - Ind1. Representation is identical to the hand-designed architecture. Not all environment states are explored here.*

Figure 5.22: *Analyzing the typical architecture found by the SOGA (marked as Ind1). It can be seen that the higher motivation (combined from two sources) causes the agent to stay nearer the reward source (once found) than in the hand-designed architecture. Compared to this, the CORGA approach is able to weight the amount of importance produced by particular sources.*

diagonalized. In both architectures, the motivation-driven RL is used. Again, the amount of motivation originates from the PNM and the agent's position in the map (therefore the agent is ***afraid of going further away from the food***). The Fig.5.23 shows the course of learning of the $Ind1$ and its contents of the memory. The knowledge is diagonalized (see the Fig.5.23(b) and the Fig.5.24(b)), but the learned data correspond to the reward source, the position of obstacle is visible too. The separated peak in is caused by the fact that the reward output of the PNM is connected to the $S1$ input. This means that during receiving the reward, the perceived X position "jumps" to the maximum value.

Recapitulation These experiments shown how the HANNS framework can be used for automatic design of agent architectures for a given task. The evolution uses predefined set of Neural Modules ordered in a predefined topology. The connection weights can be optimized by both, GA and RGA in order to provide desired behavior. Two of proposed ways of measuring the quality of behavior were described and tested. It was shown that the definition of Composed Fitness caused the evolution

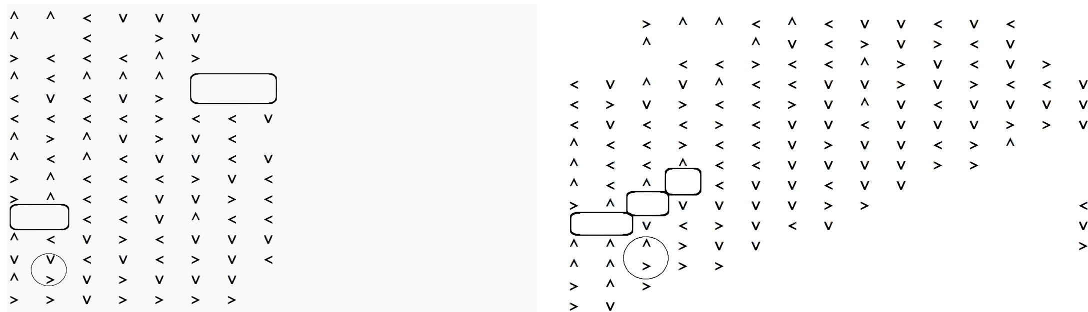


(a) *Course of life of typical best agent found by the SORGA - Ind1. Compared to the SOGA-designed architecture (Fig.5.22), this agent learns slower (slower convergence of Prosperity of the Physiological Module). This is caused by sub-optimal representation of knowledge in the Q-Lambda Module.*

(b) *Knowledge of selected agent found by the SORGA - Ind1. The Q-Lambda Module has swopped axes and value of the $S2$ variable is computed as follows: $S2 = X + 0.71Y$. This causes slower convergence of learning. The "peak" in the graph is caused by connecting reward output to the $S1$ input.*

Figure 5.23: *Behavior and knowledge learned in SORGA-designed architecture. Despite the fact the convergence of learning is slower, the overall results of behavior are similar to the SOGA-designed architecture (Fig.5.22).*

to find the local optimum, which produces undesired behavior. This means that the CF is not suitable here. Compared to this the Simple Fitness (measuring how is the agent able to meet own needs) worked as expected. In both cases, the GA found solution faster than the RGA, but the RGA was able to provide more interesting solutions. For example, the SORGA-designed $Ind1$ employed previously unknown attribute of the environment - position of the reward source near the coordinates $[X, Y] = [0, 0]$. Therefore the designed agent has tendency to stay near the low coordinates. Solutions provided by the evolution typically use less efficient representation of the knowledge in the SRP's memory, but are able to use the Module in an unexpected manner. Generally, the point is here, that such a simple set of connection weights can represent relatively complex system (that is: we could write complete equations of the system as shown in the Section 5.1.1.3).



(a) *Greedy policy of the SORGA-designed agent "Ind2". The addressing in the memory is rotated and compressed (see the Table 5.5) according to the equation* $S_1 = 0.51Y$

(b) *Greedy policy of the SORGA-designed agent "Ind1". Actions correspond to the Utility values in the Fig.5.23. Addressing in the memory is diagonalized due to multiple weighted inputs of the state variables (see the Table 5.5).*

Figure 5.24: *Visualization of the greedy policy learned by typical SORGA-designed agents. Arrows show actions with the highest utility in the state. There are marked (deformed) obstacles and and approximate position of the reward source. Because the agent "jumps" between states in the memory, the learned policy is not human-readable well. But, (as seen e.g. from the behavior in the Fig.5.23) the learned policy works relatively well (together with the motivation-driven randomization).*

Chapter 6

Conclusion

This thesis had one main goal: to bridge the abyss between various types of research by providing a platform for simpler integration of results of different research fields in computer science. One of main goals was to propose a simple approach how to combine various pieces of code together in various ways.

The thesis proposes a novel framework called Hybrid Artificial Neural Network Systems (HANNS), which inspired in Modular Neural Networks. It is able to represent each pice of code as a stand-alone sub-system, called Neural Module. The Neural Module has input and output connections defined in unified way, so that it is able to communicate with the rest of the network. If the algorithm uses some incompatible type of communication, the appropriate transformation are implemented inside Neural Module (e.g. symbol grounding). This way, various types of Neural Modules can be employed in one hybrid network, from classical neuron modules, or Fuzzy Logic nodes towards more complex ones, such as clustering algorithms, planning or Reinforcement Learning for example. By such a unification of representation of Neural Modules and communication between them, it is possible to combine the subsystems in almost arbitrary new ways. The user is able to "wire the modules" together in order to build new (agent) architecture which produces some desired, or interesting behavior.

Therefore the proposed framework can be seen as some kind of super-class of Artificial Neural Networks, Neuro-Fuzzy systems, logic circuits and more. Compared to these systems, which are composed of small nodes, the proposed framework is able to reduce the complexity of the system's topology by encapsulating more complex algorithms/sub-systems into one Neural Module. This has similar benefits to Modular/Hybrid Neural

Networks.

By defining such a unified representation of subsystems, it is possible to employ some search algorithm for designing new networks of Neural Modules - new architectures. Such a search algorithm can be then used for discovering new, useful ways of combining known algorithms/parts together. Since the size of space of all possible combinations of (even small set of) Neural Modules is huge, several constraints were put to allowable configurations of Neural Modules. The principle of automatic design is shown on examples of agent architectures, where Neural Modules are placed in a feedforward three-layered topology.

6.1 Fulfillment of Thesis Goals

Here will be described how the main goals of the thesis were fulfilled. The goals were the following.

Goal 1 - Providing a tool that will enable fast integration of current knowledge.
During deciding which platform to define/use for integration of different systems, the author found the Robotic Operating System (ROS). The ROS tries to define and distribute pieces of code that are re-usable in more domains. Therefore the author employed the ROS during defining the framework. The ROS node can be transformed into Neural Module (with unified connections) without need of modification of the node. This is done by adding simple Encoder and Decoder. From now on, the ROS node is represented as a Neural Module with given number of input and output connections. This allows integration of the Module with other Neural Modules - other subsystems. This goals is therefore considered as fulfilled.

Goal 2 - Provide a framework that will be able to build hybrid systems by hand.
In order to be able to physically connect existing pieces of code implemented in ROS in similar way as ANNs, the simulator NengoROS was created by fusing existing simulator of large-scale SNNs with the ROS. While the Nengo part is used as a front-end and simulation engine, the ROS part runs on a background and waits to be used. After defining encoder and decoder for each ROS node (and therefore representing it as a Neural Module), the user is able to wire the Neural

Modules together. This is possible either in the Jython scripting interface, Java, or in the GUI of the original Nengo simulator. The behavior of newly created architectures can be observed directly by using real-time graphs.

Goal 3 - Explore possibilities of automatic design of new hybrid systems.
The author focused on constraining the space of all possible mutual configurations of Neural Modules. The proposed framework defines agent architectures as three-layer feed-forward hybrid networks containing predefined set of Neural Modules. It was shown that such a constrained configuration space can be then efficiently searched by the Evolutionary Algorithm. Experiments shown, that by using even a simple set of predefined Modules, the evolution was able to produce some unexpected results. The resulting automatically-designed architectures use either unexpected (and successful) combinations of Neural Modules, and/or by exploit a non-anticipated property of the task. And this is exactly fulfillment of the third goal - automatic exploration of new ways how the current pieces of code can be connected together.

The conclusion can be made that all goals stated at the beginning of the thesis were fulfilled.

6.2 Main Findings of the Thesis

The framework for defining agent architectures was tested on various experiments, covering simulations in both discrete and continuous-time domain. Tested architectures included simple Neural Modules, more complex top-down designed Neural Modules featuring learning, spiking neurons etc.

The main finding in the thesis is that the concept of (semi-)automatic design of HANNS works as expected. The Evolutionary Algorithm is able to pick the form of representations of information in the Module's memory. Even on very simple tasks and using very simple Neural Modules, the EA was able to successfully employ these modules in a completely different way than was expected. This feature could be increasingly interresting with more complex Neural Modules used in the architectures, because the EA could be able to discover some completely new ways of employing the current algorithms.

Another finding is the following. Correct definition of the objective that will be solved by the agent is crucial for automatic design of a suitable architecture. It was experimentally shown how the objective of the agent can be evaluated either from outside (computing average distance from the required behavior) or from the inside - by measuring values of selected Prosperity outputs.

It was also shown that defining agent's goals is not an easy task in general. While using the fitness value, which was composed of Prosperities of multiple modules, the evolution was unable find an desired solution. This issue could be solved by employing some multi-objective optimization technique (e.g. MOEA), or by some better way of combining the Prosperity values together.

6.3 Known Limitations of the Research

Aside many benefits of the presented framework and simulator, there are some drawbacks too. Probably the main problem during the automatic design of architectures is in the speed of the NengoROS simulator. By instantiation of too many connection weights, the requirements of the simulator grow fast than would be suitable. This drawback cold be easily solved by further modification of the Nengo engine or by bypassing the engine during the simulation completely.

On the theoretical side, the automatic search for new architecture is constrained to hand-defined set of Neural Modules (architecture template). This could be improved in the future by adding some heuristic selection of Neural Modules from the library.

6.4 Future Directions and Practical Use

Based on the past research, the author proposes the following possible directions of future research:

- Experimental testing of automatic ILP-based configuration of input/output dimensions before starting of optimization of the architecture.

- Further integration and testing new Neural Modules
- Exploring the possibilities of using the Multi-Objective EA for the fitness functions composed of multiple Prosperity outputs of multiple Neural Modules.
- To propose an efficient method of configuring various subsystems in a similar way. That is to answer the questions like: how to setup filtering of input values, how to setup inner coefficients of algorithm etc.
- Automatic adding of Neural Modules in the architecture template. The choice can be based for example on Neural Modules' keywords and particular Classes of Neural Modules.

On the practical side, this approach of automatic designing of new architectures could be used in domains, where the used does not know the entire problem. This means that the user is not able to define the desired architecture exactly. Here, the evolutionary design of architectures can be used efficiently: The user defines set of Neural Modules that should be in the network and the EA tries to the architecture for a given task.

Another note towards practical use. By direct employing the ROS, it is possible to use the entire proposed framework not only in simulated environments, but in a real-world applications. Currently, the ROS supports many robotic systems directly (Quigley et al. 2009; Collective of Authors 2014).

One of the main benefits of this approach is the fact that the HANNS framework is capable of very compact representations of complex systems. If a big amount of top-down design is used (that is: "big"" Neural Modules are defined), there can be only small amount of connection weights that affect and define the resulting behavior of the system.

Bibliography

Abbott, Laurence and Terrence J. Sejnowski, eds. (1999). *Neural Codes and Distributed Representations: Foundations of Neural Computation*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-51100-2.

Agrawal, Rakesh and Ramakrishnan Srikant (1994). "Fast Algorithms for Mining Association Rules in Large Databases". In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 487–499. ISBN: 1-55860-153-8. URL: `http://dl.acm.org/citation.cfm?id=645920.672836`.

Araabi, B.N., S. Mastoureshgh, and M.N. Ahmadabadi (2007). "A Study on Expertise of Agents and Its Effects on Cooperative Q-Learning". In: *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on* 37.2, pp. 398–409. ISSN: 1083-4419. DOI: `10.1109/TSMCB.2006.883264`.

Ashlock, Daniel (2010). *Evolutionary Computation for Modeling and Optimization*. 1st. Springer Publishing Company, Incorporated. ISBN: 1441919694, 9781441919694.

Auda, G. and M. Kamel (1999). "Modular neural networks: a survey." eng. In: *Int J Neural Syst* 9.2, pp. 129–151.

Bakker, Bram and Jürgen Schmidhuber (2004). "Hierarchical reinforcement learning with subpolicies specializing for learned subgoals." In: *Neural Networks and Computational Intelligence*. IASTED/ACTA Press, pp. 125–130. URL: `http://dblp.uni-trier.de/db/conf/nci/nci2004.html%5C#BakkerS04`.

Baldwin, J. Mark (1896). "A New Factor in Evolution (Continued)". English. In: *The American Naturalist* 30.355, pages. ISSN: 00030147. URL: `http://www.jstor.org/stable/2453231`.

Bekolay, Trevor, Carter Kolbeck, and Chris Eliasmith (2013). "Simultaneous unsupervised and supervised learning of cognitive functions in biologically plausible spiking neural

networks”. In: *35th Annual Conference of the Cognitive Science Society*. Cognitive Science Society, pp. 169–174.

Bengio, Yoshua (2009). “Learning Deep Architectures for AI”. In: *Found. Trends Mach. Learn.* 2.1, pp. 1–127. ISSN: 1935-8237. DOI: 10.1561/2200000006.

Boahen, Kwabena (2006). “Neurogrid: emulating a million neurons in the cortex.” In: *Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society.*

Boers, E. J. W., M.V. Borst, and I.G. Sprinkhuizen-Kuyper (1995). *Evolving Artificial Neural Networks using the "Baldwin Effect"*.

Boers, Egbert J. W. and Herman Kuiper (1992). “Biological metaphors and the design of modular artificial neural networks”. PhD thesis.

Briot, Jean-pierre, Thomas Meurisse, and Frédéric Peschanski (2006). “Architectural Design of Component-based Agents: a Behavior-based Approach”. In: *AAMAS’06 4th Int. Workshop on Programming Multi-Agent Systems (ProMAS’06*. Springer, pp. 35–49.

Brooks, R.A. (1986). “A robust layered control system for a mobile robot”. In: *Robotics and Automation, IEEE Journal of* 2.1, pp. 14–23. ISSN: 0882-4967. DOI: 10.1109/JRA.1986.1087032.

Broomhead, D.S. and D. Lowe (1988). “Multivariable Functional Interpolation and Adaptive Networks”. In: *Complex Systems* 2, pp. 321–355.

Brown, Gavin et al. (2005). “Diversity creation methods: A survey and categorisation”. In: *Journal of Information Fusion* 6, pp. 5–20.

Busoniu, Lucian et al. (2010). *Reinforcement Learning and Dynamic Programming Using Function Approximators*. 1st. Boca Raton, FL, USA: CRC Press, Inc. ISBN: 1439821089, 9781439821084.

Clune, J., B.E. Beckmann, et al. (2009). “Evolving coordinated quadruped gaits with the HyperNEAT generative encoding”. In: *Evolutionary Computation, 2009. CEC ’09. IEEE Congress on*, pp. 2764–2771. DOI: 10.1109/CEC.2009.4983289.

Clune, Jeff, Jean-Baptiste Mouret, and Hod Lipson (2013). “The evolutionary origins of modularity”. In: *Proceedings of the Royal Society Biological Sciences* 280, pp. 1–8. DOI: 10.1098/rspb.2012.2863.

Collective of Authors (2014). “Robotický operační systém ROS míří z akademické sféry do průmyslu”. In: *AUTOMA - časopis pro automatizační techniku* 20.3, p. 39. ISSN: 1210-9592.

Conradie, Alex, Risto Miikkulainen, and Christiaan Aldrich (2002). "Intelligent Process Control Utilizing Symbiotic Memetic Neuro-Evolution". In: *Proceedings of the 2002 Congress on Evolutionary Computation*, p. 6. URL: http://www.cs.utexas.edu/users/ai-lab/?conradie:cec02.

Crawford, Eric, Matthew Gingerich, and Chris Eliasmith (2013). "Biologically Plausible, Human-scale Knowledge Representation". In: *35th Annual Conference of the Cognitive Science Society*, pp. 412–417.

Cybenko, George (1989). "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems (MCSS)* 2, pp. 303–314.

Deb, Kalyanmoy (2011). "Multi-objective Optimisation Using Evolutionary Algorithms: An Introduction". English. In: *Multi-objective Evolutionary Optimisation for Product Design and Manufacturing*. Ed. by Lihui Wang, Amos H. C. Ng, and Kalyanmoy Deb. Springer London, pp. 3–34. ISBN: 978-0-85729-617-7. DOI: 10.1007/978-0-85729-652-8_1.

Dennett, Daniel C. (2003). "The Baldwin Effect: A Crane, Not a Skyhook". In: *And Learning: The Baldwin Effect Reconsidered*. Mit Press.

Dietterich, Thomas G. (2000). "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition". In: *J. Artif. Int. Res.* 13.1, pp. 227–303. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=1622262.1622268.

Drchal, J. et al. (2011). *Fyzikální simulátor mobilních robotů - ViVAE: Visual Vector Agent Environment - ViVAE*. URL: https://github.com/HKou/vivae.

Durr, P., C. Mattiussi, and D. Floreano (2010). "Genetic Representation and Evolvability of Modular Neural Controllers". In: *Computational Intelligence Magazine, IEEE* 5.3, pp. 10–19. DOI: 10.1109/MCI.2010.937319.

Džeroski, Saso and Bernard Ženko (2004). "Is Combining Classifiers with Stacking Better Than Selecting the Best One?" In: *Mach. Learn.* 54.3, pp. 255–273. ISSN: 0885-6125. DOI: 10.1023/B:MACH.0000015881.36452.6e.

Eliasmith, C. (2013). *How to Build a Brain: A Neural Architecture for Biological Cognition*. Oxford Series on Cognitive Models and Architectures. Oxford University Press, USA. ISBN: 9780199794690. URL: http://books.google.cz/books?id=eh1pAgAAQBAJ.

Eliasmith, Ch. and Ch. H. Anderson (2003). *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. The MIT press, Cambridge, ISBN: 0-262-05071-4.

Eliasmith, Chris, Terrence C. Stewart, et al. (2012). "A Large-Scale Model of the Functioning Brain". In: *Science* 338.6111, pp. 1202–1205. DOI: 10.1126/science.1225266. eprint: http://www.sciencemag.org/content/338/6111/1202.full.pdf.

Elman, Jeffrey L. (1990). "Finding structure in time". In: *Cognitive Science* 14.2, pp. 179–211. DOI: 10.1016/0364-0213(90)90002-E.

Erol, K., D. Nau, and J. Hendler (1994). "HTN Planning: Complexity and Expressivity." In: *In AAAI-94, Seattle*.

Fekiac, Jozef, Ivan Zelinka, and Juan C. Burguillo (2011). "A Review of Methods for Encoding Neural Network Topologies in Evolutionary Computation". In: *Proceedings 25th European Conference on Modelling and Simulation ECMS*. ISBN: 978-0-9564944-2-9, pp. 410–416.

Fidjeland, A. K., E. B. Roesch, et al. (2009). "NeMo: A platform for neural modelling of spiking neurons using GPUs." In: *In Proc. 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*.

Fidjeland, A. K. and M. P. Shanahan (2010). "Accelerated simulation of spiking neural networks using gpus". In: *In Proc. IEEE International Joint Conference on Neural Networks*.

Fikes, R. and N. Nilsson (1971). "STRIPS: a new approach to the application of theorem proving to problem solving". In: *Artificial Intelligence 2*, pp. 189–208.

Fišer, P. et al. (2010). "On logic synthesis of conventionally hard to synthesize circuits using genetic programming". In: *IEEE 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pp. 346–351.

Fournier-Viger, Philippe and VincentS. Tseng (2011). "Mining Top-K Sequential Rules". English. In: *Advanced Data Mining and Applications*. Ed. by Jie Tang et al. Vol. 7121. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 180–194. ISBN: 978-3-642-25855-8. DOI: 10.1007/978-3-642-25856-5_14.

Freund, Yoav and Robert E. Schapire (1997). "A Decision-theoretic Generalization of On-line Learning and an Application to Boosting". In: *J. Comput. Syst. Sci.* 55.1, pp. 119–139. ISSN: 0022-0000. DOI: 10.1006/jcss.1997.1504.

Fuller, Robert (2001). *Neuro-Fuzzy Methods*. Vacation School, Neuro-Fuzzy Methods for Modelling and Fault Diagnosis.

Fullér, Robert (1995). *Neural Fuzzy Systems*. Abo Akademi University. ISBN: 951-650-624-0.

Gamez, D, A K Fidjeland, and E Lazdins (2012). "iSpike: a spiking neural interface for the iCub robot". In: *Bioinspiration and Biomimetics* 7.

Garis, Hugo de et al. (2010). "A World Survey of Artificial Brain Projects, Part I: Large-scale Brain Simulations". In: *Neurocomput.* 74.1-3, pp. 3–29. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2010.08.004.

Gashler, M., C. Giraud-Carrier, and T. Martinez (2008). "Decision Tree Ensemble: Small Heterogeneous Is Better Than Large Homogeneous". In: *Machine Learning and Applications, 2008. ICMLA '08. Seventh International Conference on*, pp. 900–905. DOI: 10.1109/ICMLA.2008.154.

Gerstner, Wulfram and Werner Kistler (2002). *Spiking Neuron Models. Single Neurons, Populations, Plasticity.* Cambridge University Press.

Guckelsberger, Christian and Daniel Polani (2014). "Effects of Anticipation in Individually Motivated Behaviour on Survival and Control in a Multi-Agent Scenario with Resource Constraints". In: *Entropy* 16.6. Current IF=1.53, pp. 3357–3378. ISSN: 1099-4300. DOI: 10.3390/e16063357.

Hansen, Nikolaus and Stefan Kern (2004). "Evaluating the CMA Evolution Strategy on Multimodal Test Functions". English. In: *Parallel Problem Solving from Nature - PPSN VIII.* Ed. by Xin Yao et al. Vol. 3242. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 282–291. ISBN: 978-3-540-23092-2. DOI: 10.1007/978-3-540-30217-9_29.

Happel, Bart L.M. and Jacob M. J. Murre (1994). "The Design and Evolution of Modular Neural Network Architectures". In: *Neural Networks* 7, pp. 985–1004.

Harnad, Stevan (1990). *The Symbol Grounding Problem.* URL: http://cogprints.org/3106/.

Hawkins, J., S. Ahmad, and D. Dubinsky (2011). *Hierarchical Temporal Memory including Cortical Learning Algorithms.* Tech. rep. Numenta. URL: http://numenta.org/resources/HTM_CorticalLearningAlgorithms.pdf.

Hawkins, Jeff and Sandra Blakeslee (2004). *On Intelligence.* Times Books, p. 174. ISBN: 0805074562.

Hinton, G. E. and S. J. Nowlan (1987). "How learning can guide evolution". In: *Complex Systems* 1, pp. 495–502.

Hopfield, John J (1982). "Neural networks and physical systems with emergent collective computational abilities". In: *Proc Natl Acad Sci U S A* 79.8, pp. 2554–2558. URL: http://www.ncbi.nlm.nih.gov/pubmed/6953413.

Hornik, Kurt, M. Stinchcombe, and H. White (1989). "Approximation capabilities of multilayer feedforward networks". In: *Journal Neural Networks* 2, pp. 359–366.

Choo, Xuan and Chris Eliasmith (2013). "General Instruction Following in a Large-Scale Biologically Plausible Brain Model". In: *35th Annual Conference of the Cognitive Science Society*. Cognitive Science Society, pp. 322–327.

Ijspeert, Auke Jan et al. (2007). "From Swimming to Walking with a Salamander Robot Driven by a Spinal Cord Model". In: *Science* 315. DOI: 10.1126/science.1138353.

Izhikevich, Eugene M. (2003). "Simple Model of Spiking Neurons". In: *IEEE Transactions of Neural Networks* 14, pp. 1569–1572.

Izhikevich, Eugene M. and Gerald M. Edelman (2008). "Large-Scale Model of Mammalian Thalamocortical Systems". In: *PNAS*. Vol. 105. The Neurosciences Institute, 10640 John Jay Hopkins Drive, San Diego, CA, 92121., pp. 3593–3598.

Jaeger, Herbert and Harald Haas (2004). "Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication". In: *Science* 304, pp. 78–80. DOI: doi:10.1126/science.1091277.

Jiang, Fei, Hugues Berry, and Marc Schoenauer (2009). "The Impact of Network Topology on Self-organizing Maps". In: *Proceedings of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation*. GEC '09. Shanghai, China: ACM, pp. 247–254. ISBN: 978-1-60558-326-6. DOI: 10.1145/1543834.1543869.

Jiang, Ju (2007). "A Framework for Aggregation of Multiple Reinforcement Learning Algorithms". AAINR34511. PhD thesis. Waterloo, Ont., Canada, Canada. ISBN: 978-0-494-34511-5.

Jiang, Ju and M.S. Kamel (2006). "Aggregation of Reinforcement Learning Algorithms". In: *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, pp. 68–72. DOI: 10.1109/IJCNN.2006.246661.

Jilk, David J. et al. (2008). "SAL: an explicitly pluralistic cognitive architecture". In: *Journal of Experimental & Theoretical Artificial Intelligence - Pluralism and the Future of Cognitive Science*. Vol. 20, pp. 197–218.

Jordan, M.I. (1986). "Serial order: a parallel distributed processing approach. Technical report, June 1985-March 1986". In: URL: http://www.osti.gov/scitech/servlets/purl/6910294.

Kadlecek, D. and P. Nahodil (2008). "Adopting animal concepts in hierarchical reinforcement learning and control of intelligent agents". In: *Proc. 2nd IEEE RAS & EMBS Int. Conf. Biomedical Robotics and Biomechatronics BioRob 2008*, pp. 924–929. DOI: 10.1109/BIOROB.2008.4762882.

Kadleček, David (2008). “Motivation driven reinforcement learning and automatic creation of behavior hierarchies”. PhD thesis. Czech Technical University in Prague, Faculty of Electrical Engineering.

Khan, Gul Muhammad, F. Miller Julian, and David Halliday (2009). “In Search Of Intelligent Genes: The Cartesian Genetic Programming Computational Neuron (CGPCN)”. In: *Evolutionary Computation, 2009. CEC ’09. IEEE Congress on Computing & Processing (Hardware/Software)*, pp. 574–581.

Kordík, Pavel (2006). “Fully automated knowledge extraction using group of adaptive models evolution”. PhD thesis. Czech Technical University in Prague, FEE, Dep. of Comp. Sci. and Computers.

Kordík, Pavel and Jan Černý (2012). “On performance of meta-learning templates on different datasets”. In: *International Joint Conference on Neural Networks (IJCNN), The 2012*, pp. 1–7.

Koutník, Jan and Miroslav Šnorek (2004). “Single categorizing and learning module for temporal sequences”. In: *IEEE International Joint Conference on Neural Networks*. Vol. 4, pp. 2977–2982.

Kraskov, Alexander et al. (2007). “Local field potentials and spikes in the human medial temporal lobe are selective to image category.” In: *Journal of Cognitive Neuroscience* 19, pp. 479–492.

Krasnogor, Natalio (2012). “Memetic Algorithms”. English. In: *Handbook of Natural Computing*. Ed. by Grzegorz Rozenberg, Thomas Bäck, and JoostN. Kok. Springer Berlin Heidelberg, pp. 905–935. ISBN: 978-3-540-92909-3. DOI: 10.1007/978-3-540-92910-9_29.

Krenker, Andrej, Janez Bešter, and Andrej Kos (2011). “Artificial Neural Networks - Methodological Advances and Biomedical Applications”. In: ed. by Kenji Suzuki. ISBN: 978-953-307-243-2. InTech, Chapters published April 11. Chap. Introduction to the Artificial Neural Networks, pp. 3–18. DOI: 10.5772/644.

Krichmar, Jeffrey L., Nikil Dutt ans Jayram M. Nageswaran, and Micah Richert (2010). “Neuromorphic modeling abstractions and simulation of large-scale cortical networks”. In: *ICCAD ’11 Proceedings of the International Conference on Computer-Aided Design*, pp. 334–338.

Leung, F. H F et al. (2003). “Tuning of the structure and parameters of a neural network using an improved genetic algorithm”. In: *Neural Networks, IEEE Transactions on* 14.1, pp. 79–88. ISSN: 1045-9227. DOI: 10.1109/TNN.2002.804317.

Levy, Steven (1992). *Artificial life: the quest for a new creation*. New York, NY, USA: Random House Inc. ISBN: 0-679-40774-X.

Liadal, Terese (2006). *ACT-R: A cognitive architecture*. Tech. rep. Universität des Saarlandes, Wintersemester.

Liu, Zhibin, Xiaoqin Zeng, and Huiyi Liu (2012). “A Modular Hierarchical Reinforcement Learning Algorithm”. In: *Intelligent Computing Theories and Applications*. Ed. by De-Shuang Huang et al. Vol. 7390. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 375–382. ISBN: 978-3-642-31575-6. DOI: 10.1007/978-3-642-31576-3_48.

Lohn, Jason D. et al. (2004). “Evolutionary Design of an X-Band Antenna for NASA’s Space Technology 5 Mission”. In: *in Proceedings of the 2004 IEEE Antenna and Propagation Society International Symposium and USNC/URSI National Radio Science Meeting*, pp. 2313–2316.

Maass, W. (1996). “Networks of Spiking Neurons: The Third Generation of Neural Network Models”. In: *In Journal Neural Networks* 10, pp. 1659–1671.

Marder, E. and R. L. Calabrese (1996). “Principles of rhythmic motor pattern generation.” In: *Physionlogical Reviews* 76, pp. 687–717.

Markram, Henry (2006). “The Blue Brain Project”. In: *Nature Reviews Neuroscience* 7, pp. 153–160.

Matsuoka, Kiyotoshi (1985). “Sustained oscillations generated by mutually inhibiting neurons with adaptation”. In: *Biological Cybernetics*. Vol. 52. ISSN: 1432-0770. Springer, pp. 367–376. DOI: 10.1007/BF00449593.

Mcgarry, Kenneth, Stefan Wermter, and John Macintyre (1999). “Hybrid neural systems: from simple coupling to fully integrated neural networks”. In: *Neural Computing Surveys* 2, pp. 62–93.

Mingus, Brian (2011). *Comparison of Neural Network Simulators*. cited 2012. University of Colorado Boulder. URL: http://grey.colorado.edu/emergent/index.php/Comparison%5C_of%5C_Neural%5C_Network%20%5C_Simulators.

Mitchell, Melanie (1998). *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press. ISBN: 0262631857.

Moscato, Pablo (1989). *On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts - Towards Memetic Algorithms*. Technical report.

Murre, J. M. J., R. H. Phaf, and G. Wolters (1989). “CALM networks: a modular approach to supervised and unsupervised learning”. In: *Proc. Int Neural Networks IJCNN. Joint Conf*, pp. 649–656. DOI: 10.1109/IJCNN.1989.118647.

Nageswaran, J.M. and Bren Sch. Donald (2009). “Efficient simulation of large-scale Spiking Neural Networks using CUDA graphics processors”. In: *International Joint Conference on Neural Networks, 2009. IJCNN 2009*, pp. 2145–2152.

Newman, Stuart A. (2002). “Developmental mechanisms: putting genes in their place.” eng. In: *J Biosci* 27.2, pp. 97–104.

Nguyen, V. A., J.A. Starzyk, G. Wooi-Boon, et al. (2012). “Neural Network Structure for Spatio-Temporal Long-Term Memory”. In: *IEEE Transactions on Neural Networks and Learning Systems*. Vol. 23, pp. 971–983. DOI: `10.1109/TNNLS.2012.2191419`.

Nguyen, Vu Anh, J. A. Starzyk, A. L. P. Tay, et al. (2010). “Spatio-temporal sequence learning of visual place cells for robotic navigation”. In: *Proc. Int Neural Networks (IJCNN) Joint Conf*, pp. 1–8. DOI: `10.1109/IJCNN.2010.5596952`.

Nolfi, Stefano (1999). *How Learning and Evolution Interact: The Case of a Learning Task which Differs from the Evolutionary Task*.

Nwana, Hyacinth S. (1996). “Software agents: An overview”. In: *Knowledge Engineering Review* 11, pp. 205–244.

Opitz, David and Richard Maclin (1999). “Popular Ensemble Methods: An Empirical Study”. In: *Journal of Artificial Intelligence Research* 11, pp. 169–198.

O’Reilly, Randall C. (1996). “The Leabra Model of Neural Interactions and Learning in the Neocortex”. PhD thesis. Pittsburgh, PA 15213: Carnegie Mellon University.

Ozawa, S., K. Tsutumi, and N. Baba (1999). “Evolution of a dynamical modular neural network and its application to associative memories”. In: *Proc. Third Int Knowledge-Based Intelligent Information Engineering Systems Conf*, pp. 145–148. DOI: `10.1109/KES.1999.820140`.

Pei, Zhongcai et al. (2012). “Adaptive control of a quadruped robot based on Central Pattern Generators”. In: *10th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 554–558.

Pellier, Damien. *Presentation on Hierarchical Task Network Planning*. Cited 2014. URL: `http://www.math-info.univ-paris5.fr/~moraitis/webpapers/10.HTN-4pp.pdf`.

Perrett, D. I., E. T. Rolls, and W. Caan (1982). “Visual neurones responsive to faces in the monkey temporal cortex.” eng. In: *Exp Brain Res* 47.3, pp. 329–342.

Poggio, Tomaso, Ulf Knoblich, and Jim. Mutch (2010). *CNS: a GPU-based framework for simulating cortically-organized networks*. Massachusetts Institute of Technology. URL: `http://dspace.mit.edu/bitstream/handle/1721.1/51839/MIT-CSAIL-TR-2010-013.pdf`.

Ponulak, Filip and Andrzej Kasinski (2011). "Introduction to spiking neural networks: Information processing, learning and applications." In: *Acta neurobiologiae experimentalis* 71.4, pp. 409–433. ISSN: 1689-0035. URL: http://view.ncbi.nlm.nih.gov/pubmed/22237491.

Quigley, Morgan et al. (2009). "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. URL: http://pub1.willowgarage.com/~konolige/cs225B/docs/quigley-icra2009-ros.pdf.

Rast, Alexander D. et al. (2010). "Scalable event-driven native parallel processing: the SpiNNaker neuromimetic system". In: *CF '10 Proceedings of the 7th ACM international conference on Computing frontiers*, pp. 21–30. DOI: 10.1145/1787275.1787279.

Rojas, Raúl (1996). *Neural Networks: A Systematic Introduction*. New York, NY, USA: Springer-Verlag New York, Inc. ISBN: 3-540-60505-3.

Ross, M. (2002). "Hierarchical Reinforcement Learning: A Hybrid Approach". PhD thesis. The University of New South Wales, School of Computer Science and Engineering.

Russell, Stuart J. and Peter Norvig (2003). *Artificial Intelligence: A Modern Approach*. 2nd ed. Pearson Education. ISBN: 0137903952.

Sardina, S. et al. (2006). "Hierarchical Planning in BDI Agent Programming Language: a Formal Approach". In: *AAMAS 06 Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pp. 1001–1008.

Sekanina, Lukáš (2010). *Evoluční hardware - Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Ed. by Jiří Lažanský and Ivan Zelinka. Gerstner. ISBN: 978-80-200-1729-1. Academia, p. 328.

Sekereš, P. (2013). "Learning of Temporal Sequences of Behaviour for Artificial Creature". Supervisor: Doc. Ing. Nahodil Pavel CSc. (in English). Bachelor Thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, dept. of Cybernetics. URL: http://cyber.felk.cvut.cz/research/theses/detail.phtml?id=421.

Sharad, Mrigank et al. (2012). *Proposal For Neuromorphic Hardware Using Spin Devices*. URL: http://dblp.uni-trier.de/db/journals/corr/corr1206.html%5C#abs-1206-3227.

Schmidhuber, Juurgen, Daan Wierstra, and Faustino Gomez (2005). "Evolino: Hybrid Neuroevolution/Optimal Linear Search for Sequence Learning". In: *International Joint Conference on Artificial Intelligence - IJCAI*, pp. 853–858.

Skála, L. (2013). "Hybrid Decision-making System of Artificial Creature Combining Planner and Neural Network". Supervisor: Doc. Ing. Nahodil Pavel CSc. (in En-

glish). Bachelor Thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, dept. of Cybernetics. URL: `http://cyber.felk.cvut.cz/research/theses/detail.phtml?id=396`.

Stanley, Kenneth O., Bobby D. Bryant, and Risto Miikkulainen (2005). "Real-time Neuroevolution in the NERO Video Game". In: *IEEE Transactions on Evolutionary Computation*, pp. 653–668. URL: `http://nn.cs.utexas.edu/?stanley:ieeetec05`.

Stanley, Kenneth O., David B. D'Ambrosio, and Jason Gauci (2009). "A hypercube-based encoding for evolving large-scale neural networks." eng. In: *Artif Life* 15.2, pp. 185–212. DOI: `10.1162/artl.2009.15.2.15202`.

Stewart, Terrence C., Trevor Bekolay, and Chris Eliasmith (2011). "Neural Representations of Compositional Structures: Representing and Manipulating Vector Spaces with Spiking Neurons". In: *Connection Science* 22, pp. 145–153. DOI: `10.1080/09540091.2011.571761`.

Stewart, Terrence C. and Chris Eliasmith (2013). "Parsing Sequentially Presented Commands in a Large-Scale Biologically Realistic Brain Model". In: *35th Annual Conference of the Cognitive Science Society*. Cognitive Science Society, pp. 3460–3467.

Strogatz, Steven H (1997). "Spontaneous synchronization in nature". In: *Frequency Control Symposium*. IEEE.

Sugita, Yuuya and Martin V. Butz (2008). "Towards Emergent Strong Systematicity in a Simple Dynamical Connectionist Network". In: *Computer Technologies and Information Sciences; Biology and Medicine*. Department of Cognitive Psychology, Universität Würzburg.

Sutton, Richard S. and Andrew G. Barto (1998). *Reinforcement Learning: An Introduction*. 1st. Cambridge, MA, USA: MIT Press. ISBN: 0262193981.

Suzuki, R. and T. Arita (2007). "Repeated Occurrences of the Baldwin Effect Can Guide Evolution on Rugged Fitness Landscapes". In: *Proc. IEEE Symp. Artificial Life ALIFE '07*, pp. 8–14. DOI: `10.1109/ALIFE.2007.367652`.

Suzuki, Reiji and Takaya Arita (2004). "Interactions between learning and evolution: the outstanding strategy generated by the Baldwin effect." eng. In: *Biosystems* 77.1-3, pp. 57–71. DOI: `10.1016/j.biosystems.2004.04.002`.

Tang, Yichuan and Chris Eliasmith (2010). "Deep networks for robust visual recognition". In: *Proceedings of the 27th International Conference on Machine Learning, June 21-24, 2010, Haifa, Israel*.

Tay, A.L.P. et al. (2007). "The Hierarchical Fast Learning Artificial Neural Network (HieFLANN);An Autonomous Platform for Hierarchical Neural Network Construc-

tion". In: *Neural Networks, IEEE Transactions on* 18.6, pp. 1645–1657. ISSN: 1045-9227. DOI: 10.1109/TNN.2007.900231.

Thomas, David B. and Wayne Luk (2009). "FPGA accelerated simulation of biologically plausible spiking neural networks". In: *In Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM)*.

Tijsseling, A. G. (2005). "Sequential information processing using time-delay connections in ontogenic CALM networks". In: *IEEE Trans. Neural Netw.* 16.1, pp. 145–159. DOI: 10.1109/TNN.2004.839355.

Togelius, Julian, Faustino Gomez, and Tom Schmidhuber (2008). "Learning what to ignore: Memetic climbing in topology and weight space". In: *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, pp. 3274–3281. DOI: 10.1109/CEC.2008.4631241.

Togelius, Julian, Tom Schaul, et al. (2008). "Countering Poisonous Inputs with Memetic Neuroevolution". English. In: *Parallel Problem Solving from Nature – PPSN X*. Ed. by Günter Rudolph et al. Vol. 5199. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 610–619. ISBN: 978-3-540-87699-1. DOI: 10.1007/978-3-540-87700-4_61.

Valdivieso, Pedro A. Castillo et al. (2006). "Lamarckian Evolution and the Baldwin Effect in Evolutionary Neural Networks". In: *CoRR* abs/cs/0603004.

Valsalam, V. K. and R. Miikkulainen (2011). "Evolving Symmetry for Modular System Design". In: *Evolutionary Computation, IEEE Transactions on* 15.3, pp. 368–386. DOI: 10.1109/TEVC.2011.2112663.

Vašíček, Zdeněk and Lukáš Sekanina (2004). *Evoluční návrh kombinačních obvodů*. Czech. cited 2012. Fakulta informačních technologií, Vysoké učení technické Brno. URL: http://www.elektrorevue.cz/clanky/04043/index.html.

Vieira, José, Fernando Morgado Dias, and Alexandre Mota (2004). "Neuro-Fuzzy Systems: A Survey". In: *5th WSEAS NNA International Conference on Neural Networks and Applications*. Udine, Italy.

Voegtlin, Thomas (2002). "Recursive Self-organizing Maps". In: *Neural Netw.* 15.8-9, pp. 979–991. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(02)00072-2.

Wang, D. and M. A. Arbib (1990). "Complex temporal sequence learning based on short-term memory". In: 78.9, pp. 1536–1543. DOI: 10.1109/5.58329.

Wiering, M.A. and H. van Hasselt (2008). "Ensemble Algorithms in Reinforcement Learning". In: *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on* 38.4, pp. 930–936. ISSN: 1083-4419. DOI: 10.1109/TSMCB.2008.920231.

Wilamowsky, Bogdan M. (2003). "Neural Network Architectures and Learning". In: *ICIT'03 - IEEE International Conference on Industrial Technology*. Vol. 1. ISBN: 0-7803-7852-0. Maribor, Slovenia, pp. 1–12. DOI: 10.1109/ICIT.2003.1290197.

Wildie, M. et al. (2009). "Reconfigurable acceleration of neural models with gap junctions". In: *In Proc. International Conference on Field-Programmable Technology*.

Wiles, Janet and James R. Watson (2001). "How Learning Can Guide Evolution in Hierarchical Modular Tasks". In: *Proceedings of the 23rd Annual Conference of the Cognitive Science Society*, pp. 1130–1135.

Wolpert, D. H. and W. G. Macready (1997). "No Free Lunch Theorems for Optimization". In: *Trans. Evol. Comp* 1.1, pp. 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893.

Wooldridge, M. (1995). "Conceptualising and Developing Agents". In: *In Proceedings of the UNICOM Seminar on Agent Software*. London, pp. 40–54.

Yao, Xin, S. M. Ieee, and Yong Liu (1996). "A New Evolutionary System for Evolving Artificial Neural Networks". In: *IEEE Transactions on Neural Networks* 8, pp. 694–713.

Yu, Qiang et al. (2013). "Rapid Feedforward Computation by Temporal Encoding and Learning With Spiking Neurons". In: *Neural Networks and Learning Systems, IEEE Transactions on* 24.10, pp. 1539–1552. ISSN: 2162-237X. DOI: 10.1109/TNNLS.2013.2245677.

Yudanov, Dimitri (2010). "GPU-based simulation of spiking neural networks with real-time performance and high accuracy". In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*.

Zadeh, Lotfi A. (1994). "Fuzzy Logic, Neural Networks, and Soft Computing". In: *Commun. ACM* 37.3, pp. 77–84. ISSN: 0001-0782. DOI: 10.1145/175247.175255.

Zainer, Riadh and Fumio Nagashima (2002). *Recurrent Neural Network Language for Robot Learning*. Tech. rep. Fujitsu Laboratories LTD.

Zhang, Xiaoqin Shelley et al. (2012). "An Ensemble Architecture for Learning Complex Problem-Solving Techniques from Demonstration". In: *ACM Trans. Intell. Syst. Technol.* 3.4, 75:1–75:38. ISSN: 2157-6904. DOI: 10.1145/2337542.2337560.

Author's Publications Related to the Thesis

Peer-Reviewed Journals

Vítků, J. and P. Nahodil (2014a). "Automaticky navrhované evoluční architektury chování agentů". In: *Automa*. Accepted for publication, Author's participation = 70%. ISSN: 1210-9592.

Vítků, J. and P. Nahodil (2014d). "Towards Evolutionary Design of Complex Systems Inspired by Nature". In: *Acta Polytechnica - Journal of Advanced Engineering*. Accepted for publication, Author's participation = 70%. ISSN: 1805-2363.

Indexed by Web of Science

Nahodil, P. and J. Vítků (2012a). "How to Design an Autonomous Creature Based on Original Artificial Life Approaches". In: *Beyond Artificial Intelligence*. Author's participation = 40%. Plzeň: Springer, pp. 161–180. ISBN: 978-3-642-34421-3. DOI: 10.1007/978-3-642-34422-0_11.

Nahodil, P. and J. Vítků (2012b). "Learning of Autonomous Agent in Virtual Environment". In: *26th European Conference on Modelling and Simulation (ECMS)*. Author's participation = 40%, pp. 373–379. ISBN: 978-0-9564944-4-3. DOI: 10.7148/2012-0373-0379.

Vítků, J. and P. Nahodil (2013). "Autonomous Design of Modular Intelligent Systems". In: *27th European Conference on Modelling and Simulation ECMS 2013*. Ed. by Webjørn Rekdalsbakken, Robin T. Bye, and Houxiang Zhang. Author's participation = 60%. Alesund: European Council for Modelling and Simulation, pp. 379–389. ISBN: 978-0-9564944-6-7. DOI: 10.7148/2013-0379.

Vítků, J. and P. Nahodil (2014b). "Q-Learning Algorithm Module in Hybrid Artificial Neural Network Systems". In: *Modern Trends and Techniques in Computer Science*. Ed. by Radek Silhavy et al. Vol. 285. Springer, Advances in Intelligent Systems and Computing. Author's participation = 60%, Note: will be added in the WoS and Scopus as other proceedings in this Springer series (http://www.springer.com/series/11156). Springer International Publishing, pp. 117–127. ISBN: 978-3-319-06739-1. DOI: 10.1007/978-3-319-06740-7_11.

Vítků, J. and P. Nahodil (2014c). "Reusable Reinforcement Learning for Modular Self Motivated Agents". In: *Proceedings of 28th European Conference on Modeling and Simulation*. Ed. by Flaminio Squazzoni et al. Author's participation = 60%, Note: will

be indexed in WoS as previous issues. Brescia, Italy: European Council for Modelling and Simulation, pp. 352–358. ISBN: 978-0-9564944-8-1. DOI: 10.7148/2014-0352.

Not Indexed by Web of Science

Nahodil, P. and J. Vítků (2011). "New Way How to Build an Autonomous Creatures". In: *International Conference: Beyond Artificial Intelligence, Interdisciplinary Aspects of Artificial Intelligence.* Author's participation = 40%. Pilsen, pp. 34–41. URL: http://www.kky.zcu.cz/en/publications/1/JanRomportl_2011_BeyondAI.pdf.

Nahodil, P. and J. Vítků (2013). "Hybridní neuronové systémy pro návrh architektur autonomních agentů v oblasti umělého života". In: *Kognícia a Umelý Život XIII.* Author's participation = 40%. Opava, Stará Lesná, pp. 197–204. ISBN: 978-80-7248-863-6.

Nahodil, P. and J. Vítků (2014). "Evoluční architektura chování umělých bytosti - agentů v daném prostředí". In: *Kognitivní věda a umělý život XIV (KUZ XIV).* Author's participation = 40%. Zaječí u Břeclavi: Slezská univerzita v Opavě, pp. 155–164. ISBN: 978-80-7248-951-0.

Vítků, J. (2012). "Ethology-Inspired Advanced Problem Solving Mechanism". In: *POSTER 2012 -16th International Student Conference on Electrical Engineering.* ISBN 978-80-01-05043-9. Czech Technical University in Prague, pp. 1–6.

Vítků, J. and P. Nahodil (2012). "Nové hybridní rozhodovací mechanismy v oblasti umělého života". In: *Kognice a umělý život (KUZ XII).* Author's participation = 60%. Průhonice: Agentura Action M, pp. 254–263. ISBN: 978-80-86742-34-2.

Other Author's Publications

Indexed by Web of Science

Nahodil, P. and J. Vítků (2012c). "Novel Theory and Simulations of Anticipatory Behaviour in Artificial Life Domain". In: *Advances in Intelligent Modelling and Simulation.* Author's participation = 35%. Springer, pp. 131–164. ISBN: 978-3-642-28887-6. DOI: 10.1007/978-3-642-28888-3_6.

Not Indexed by Web of Science

Vítků, J. (2011). "An Artificial Creature Capable of Learning from Experience in Order to Fulfill More Complex Tasks". Supervisor: Doc. Ing. Nahodil Pavel CSc. (in English). Diploma Thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, Dept. of Cybernetics. URL: http://cyber.felk.cvut.cz/research/theses/detail.phtml?id=176.

Citations of Author's Publications

Guckelsberger, Christian and Daniel Polani (2014). "Effects of Anticipation in Individually Motivated Behaviour on Survival and Control in a Multi-Agent Scenario with Resource Constraints". In: *Entropy* 16.6. Current IF=1.53, pp. 3357–3378. ISSN: 1099-4300. DOI: 10.3390/e16063357.

Appendix A

Additional Knowledge on Spiking Neural Networks

Since the basic principles of Spiking Neural Networks (SNNs) are not widely known, this Appendix will briefly mention some of them. Selected models of spiking neurons are shown. Then, basics of Neural Engineering Framework (NEF) are described. Finally, the table describing features of SNN simulators is shown.

A.1 Selected Models of Spiking Neuron

This section briefly describes two selected models of spiking neurons, this is appendix to section 2.1.1.2, which describes models of artificial neurons. Note that these two models of neuron are supported by Nengo simulator and therefore can be already used for simulation of hybrid networks. The drawbacks and benefits of each model are mentioned here too.

A.1.1 Leaky Integrate-and-fire Model of Neuron

One of the most common type of 3rd gen. neuron is Leaky integrate-and-fire (LIF) model, well described e.g. in (Gerstner and Kistler 2002). This model is composed of differential equations which represent behavior of model based on actual values on input and state

of the neuron.

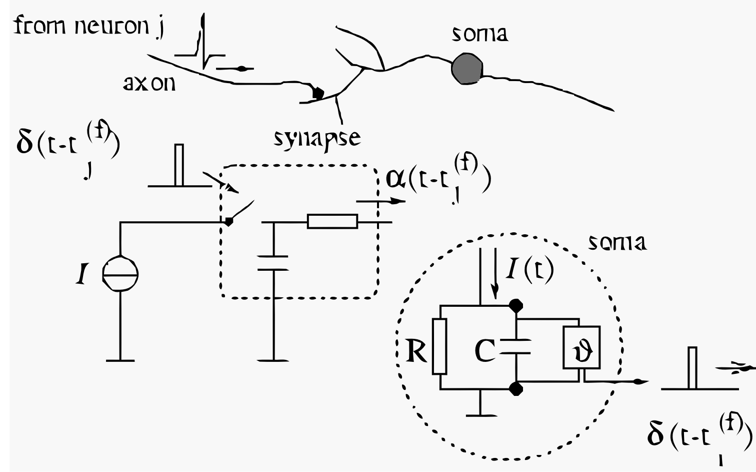


Figure A.1: *The basic circuit is the module inside the dashed circle on the right-hand side. A current $I(t)$ charges the RC circuit. The voltage $u(t)$ across the capacitance (points) is compared to a threshold. If $u(t) = \theta$ at time $t_i^{(f)}$ an output pulse $(t - t_i^{(f)})$ is generated. Left part: A presynaptic spike $(t - t_j^{(f)})$ is low-pass filtered at the synapse and generates an input current pulse $\alpha(t - t_j^{(f)})$. (Gerstner and Kistler 2002).*

Scheme of LIF neuron can be seen in the fig.A.1. The driving current is split into two parts:

$$I(t) = I_R + I_C, \tag{A.1}$$

where the current on resistor can be computed as $I_R = u/R$ and the current on capacitor C can be from the definition of capacity: $C = q/u$ (q is charge) computed as $I_C = C\ du/dt$. This results in the following equation:

$$I(t) = \frac{u(t)}{R} + C\frac{du}{dt}. \tag{A.2}$$

We multiply this equation by R and introduce the time constant $\tau_m = RC$ of the "leaky integrator", this results in the standard form of LIF equation:

$$\tau_m \frac{du}{dt} = -u(t) + RI(t). \tag{A.3}$$

The variable u denotes membrane potential of neuron and τ_m is membrane time constant. Spikes are generated as formal events characterized by a "firing time" $t^{(f)}$, which is defined by a threshold function:

$$t^{(f)} : \ u(t^{(f)}) = \theta. \tag{A.4}$$

After producing the spike at time $t^{(f)}$, the potential is reset to a new, smaller value:

$$lim_{t \to t^{(f)}, t > t^{(f)}} u(t) = u_r, \tag{A.5}$$

where u_r is called resting membrane potential of neuron. This model also supports absolute refactory period. This denotes the procedure when the dynamics of model is interrupted for refactory time Δ^{abs} and restart the integration at time $t^{(f)} + \Delta^{abs}$ with initial condition u_r.

A.1.2 Izhikevich's Simple Model of Neuron

The benefit of the LIF model of neuron is that it represents behavior of neuron straightforwardly. The downside is that it is computationally expensive. Model of neuron with the best trade-off between computational requirements and biological plausibility is called Izhikevich's simple model of neuron. It reduces Hodgkin-Huxley model of neuron into two-dimensional system of ordinary differential equations (Izhikevich 2003). The equations describing this model are as follows:

$$v' = 0.04v^2 + 5v + 140 - u + I \tag{A.6}$$

$$u' = a(bv - u). \tag{A.7}$$

Also this model uses explicit after-spike reseting:

$$if \ v \geq 30mV, \ then \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{A.8}$$

In these equations, variable v represents the membrane potential of neuron, u represents a membrane recovery variable which provides negative feedback to v. The parameters a, b, c, d enable of changing the behavior of neuron model. By tuning these parameters, we are able to reproduce behavior of many types of cortical neurons.

A.2 Neural Engineering Framework

The Neural Engineering Framework (NEF) is extensively used in the selected simulator Nengo for top-down engineering of large-scale artificial neural networks. This section briefly introduces to main principles of neural engineering.

Neural Engineering Framework focuses to applying the theory of signals and systems to nervous systems, thus to explicit engineering neural ensembles to represent some value. More concretely, Nengo uses pseudo-randomly generated groups of neurons. Each group represents some information, while connections between these groups define what each group computes. Neural Engineering Framework is used to compute connection weights between these ensembles of neurons in order to provide desired computation.

The main common attribute for all of these types of networks is the fact that they try to somehow represent the main function implemented by the biological neuron, that is: *encoding input values into neural firing rates*, or (in case of sensory-input neurons) *encoding physical magnitudes into neural firing rates*[1], rather than individual spikes (Eliasmith and Anderson 2003). Curves in the graph A.2 represent **response functions**, they tell us how neural activity relates to soma currents.

But we need to know other thing than response functions, we would like to know how the neuron responses to the *external stimuli* representing real-world physical magnitudes. This relation is called **tuning curve** of neuron. *"The tuning curve of a neuron is typically found by presenting the system that the neuron is in with a series of systematically varied stimuli, and recording the neuron's response"*. The figureA.3 shows the tuning curve representing the relationship between actual horizontal eye position (physical magnitude) and rate output of neuron. The figure A.4 shows tuning curves of two neural ensembles generated by Nengo neural simulator according to some preferred properties.

[1]Note that this is one of the simplest cases of types of neural coding.

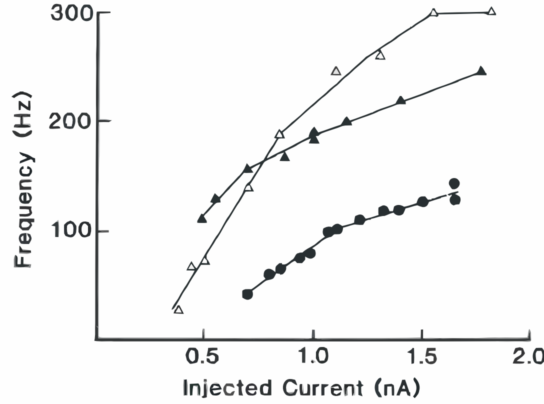


Figure A.2: *Three stereotypical neuron response functions from human cortical regular spiking cells.: current in nA injected directly to neuron soma, output is firing rate.*

The NEF uses neural ensemble to encode given value x, the information are then decoded from the ensemble. Neural Engineering Framework is based on two main aspects: **neural representation** and **neural transformation**, I will describe them now.

The sequence of applying computation is as follows:

- We present the neural ensemble some real valued input
- The ensemble applies tuning curves, which is nonlinear transformation
- Because we know what we want as output and we know input transformation, we can **compute** the output weights ϕ_i of neurons **by minimizing the output error**.

The following two sub-sections will describe how real-world values are encoded into neural ensemble and how are then decoded (or transformed in desired way).

A.2.0.1 Neural Encoding Process

First, the (real-valued) input is connected to the termination of Neural ensemble. This ensemble encodes the value into an activity of population of neurons. This transformation (from x axis to multidimensional result on y axis) is depicted in the A.4. Note that this process is non-linear. This input transformation can be written as $a_i(x)$, where a_i is transformation implemented by the neuron i and x is the encoded value.

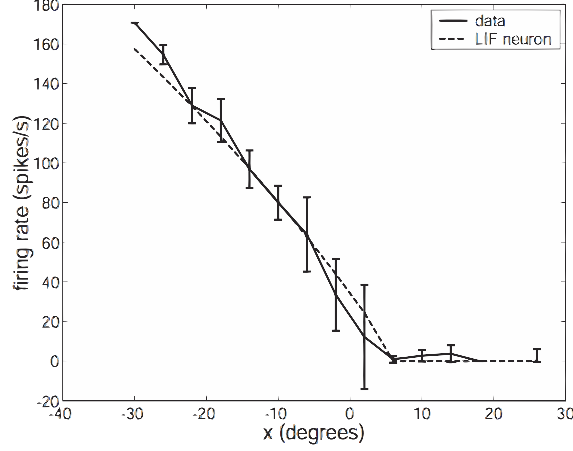


Figure A.3: *A typical neuron tuning curve that codes for horizontal eye position. The grey line indicates a leaky integrate- and-fire (LIF) neuron approximation to the actual tuning curve (black line). This is a tuning curve (as opposed to a response function) because it is a function of some variable x,7,A typical neuron tuning curve in the nuclei prepositus hypoglossi that codes for horizontal eye position. The grey line indicates a leaky integrate-and-fire (LIF) neuron approximation to the actual tuning curve (black line). This is a tuning curve (as opposed to a response function) because it is a function of some variable x, not just current entering the soma.*

As mentioned earlier, the encoding process in biological neuron is composed of two parts:

- highly complex transformation of physical magnitudes (inputs) to soma current. This includes collecting all spikes on synapses, dendritic transformation etc. This process generates current on soma.
- second part converts soma current to neuron activity (simplified: firing rate). This is called response function and depicted in A.2.

The result of these two processes is called *tuning curve* and is depicted in A.3 and A.4. Neural Engineering Framework describes these two processes separately, transformation from "inputs" to soma current is written as $J(x)$ and transformation of soma current to neuron activity as $G(y)$. Thus, a general expression for neural encoding process for each neuron is:

$$a_i(x) = G_i[J_i(x)]. \tag{A.9}$$

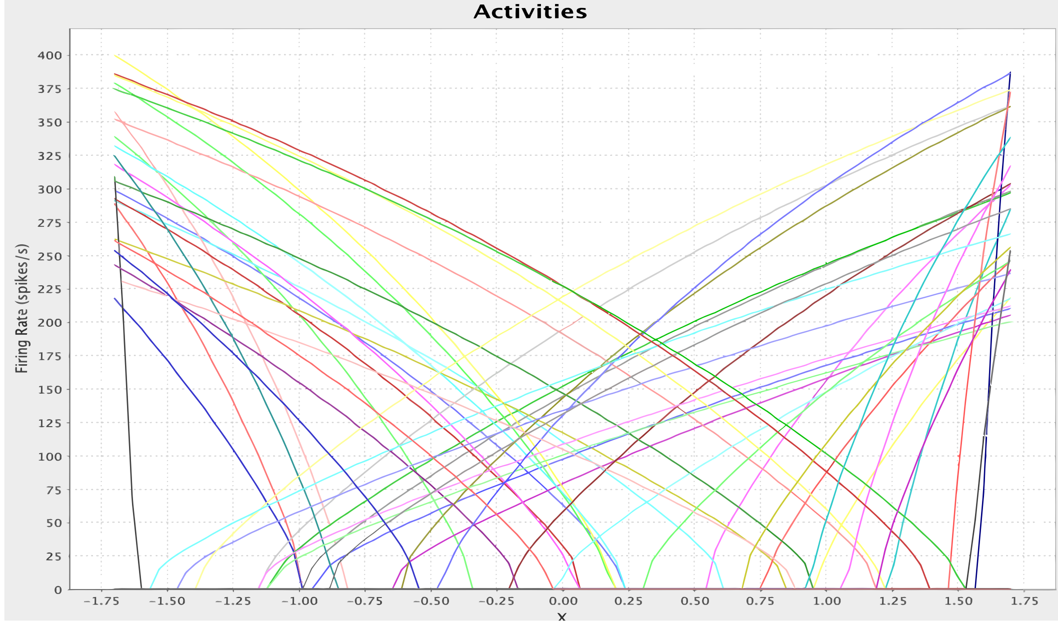


Figure A.4: *Graph showing generated tuning curves for one neural ensemble with 50 nodes in the Nengo simulator. Note that ensemble has radius set to 1.7.*

The soma current, $J(x)$ results from combination of two different currents, called *"bias"* current J^{bias} and *"driving"* current $J^d(x)$.

$$J_i(x) = J_i^{bias} + J_i^d(x) \tag{A.10}$$

Where bias current represents some constant input to the neuron, while driving current depends directly on input x, so we can write $J^d(x)$ as:

$$J_i^d(x) = \alpha_i x, \tag{A.11}$$

where α_i represents weight of input to of neuron i. Note the similarity with equation for more conventionally used ANNs of second generation. These equations are shown on case when neural ensemble has only one input (thus only one weighted input α), but this can be simply extended for more-dimensional inputs. At this place, the equation of input transformation $a_i(x)$ is rewritten for case of Leaky Integrate and Fire (LIF) model of neuron, but the equations above describe all types of neurons (we can use e.g. second generation of neurons with sigmoid transfer function, Izhikevich model etc..). I will just

rewrite the equation of encoding for one neuron once more:

$$a_i(x) = G_i[\alpha_i x + J_i^{bias}]. \tag{A.12}$$

The properties of this encoding are well visible in the figure A.4, where the representation is highly redundant - *"overcomplete"* and each neuron represents some part of input space. This is similar to one of encoding possibilities of information in the biological systems. Compared to this, engineering techniques are *"complete"*, this means non-redundant. Note that completeness of this representation can be fluently tuned by number of neurons in the population, with respect to represented domain.

A.2.0.2 Neural Decoding Process

While each ensemble is meant to represent some value, the NEF provides mathematical aparat to directly compute the decoding weights ϕ_i for each neuron, so that the *whole population* of neurons represents the value. This means that we are searching for ϕ_i in the following equation:

$$\tilde{x} = \sum_i a_i(x)\phi_i. \tag{A.13}$$

Despite the highly non-linear encoding process, this linear decoding can be used to successfully estimate a magnitude that was originally encoded. There are two main kinds of decoders:

Representational decoder is used to retrieve the same information we encoded (into ensemble).

Transformational decoder attempts to extract information other than what the population is taken to represent.

This means that neural ensemble with representational decoder should implement identity transformation in the ideal case, but ensemble with the transformational decoder can implement some desired transformation of input value $f(x)$. In this situation, we know input values to the ensemble, we know input transformations and we know what the output should be: x, or some arbitrary function of input $f(x)$. Because of the fact that

these output transformations are linear, values form particular output weights ϕ_i can be computed directly for each neuron simply by **minimizing the output error**.

In order to do this, here is the expression for the error that can be minimized to determine the values of our decoders:

$$E = \frac{1}{2} \int_{-1}^{1} \left[x - \sum_{i=1}^{N} a_i(x) \phi_i \right]^2 \mathrm{d}x. \tag{A.14}$$

This equation represent *mean square error*, it computes integral (i.e. average) over the x on interval $\langle -1, 1 \rangle$, which is the radius of neural ensemble (i.e. constrains to input space). The expression in brackets represent the difference of desired value and the decoded one (which is weighted output from all neurons).

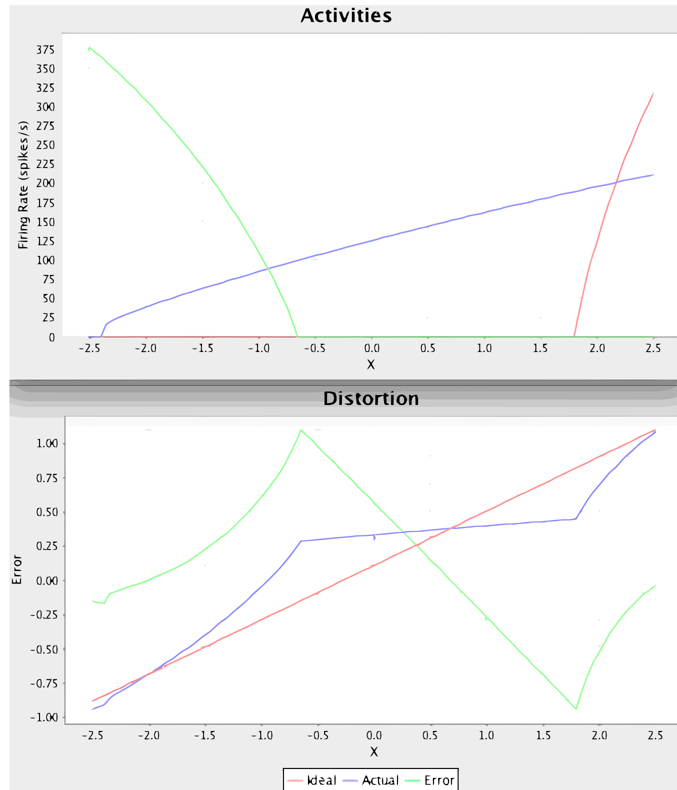


Figure A.5: *Example of imperfect representational transformation by tuning curves. The upper graph shows tuning curves of 3 neurons in the population, lower graph shows imperfect transformation. Note that we can clearly see how the transformation was computed.*

Each time we drag and drop Neural ensemble into Nengo network, Nengo generates neural ensemble (encodings) and solves this equation, that is: finds the vector of decoders

in order to represent the input value[2] as accurately as possible. Example of result of minimization of the value of E can be seen in the figure A.5. In the pictures there is clearly visible how the output transformation was computed, and where is used which neuron.

A.3 Comparison of current ANN simulators

This appendix describes the comparison and selection of ANN simulators designed for simulating large-scale networks of neurons. The fundamental requirements for the simulator can be seen in the Chapter 3.4. The table A.1 shows only selected simulators, which are not proprietary and are focused on simulating networks of neurons, not single neurons.

[2]By default Nengo solves this equation for origin "x"-identity. We can add our custom origin which implements custom transformational decoding.

Table A.1: *List of ANN simulators, inspired in (Mingus 2011)*

Comparison of only the most important features of open-source ANN simulators						
Name	**Language**	**Acceleration**	**Network Design**	**Learning algorithms**	2nd **gen.**	3rd **gen.**
Emergent	C++	MPI,GPU	GUI, scripts, C++	Many types	T	T
Torch5	C	-	Lua scripts	-	T	-
Topographica Neural Map Simulator	C++/Python	-	GUI, Python, C++	Hebbian, SOM, ..	T	T
Stuttgart Neural Network Simulator (SNNS)	C++	-	GUI	Backprop,Hebbian, ..	T	-
Simbrain	Java	-	GUI	Hebbian, SOM	T	T
Neuron	C,C++,Fortran	MPI	Visual scripts	Possible	T	T
Neuroph	Java	-	Java	Backprop,Hebbian, ..	T	-
Nengo	Java	GPU	GUI, Jython	Modulated Hebb-like	T	T
The light, efficient neural network simulator (LENS)	C	-	GUI	Quickprop, Kohonen, ..	T	-
Fast Artificial Neural Network (FANN)	C	-	GUI, many languages	Quickprop, Kohonen, ..	T	-
Encog	Java, .NET	GPU	Java, C#, .NET	Kohonen, Hopfield, ..	T	-
Cortical Neural Simulator (CNS)	C	GPU	Matlab, C	-	-	T
Brian	Python	GPU	Python	STDP	-	T
Biological Neural Network (BNN) Toolbox	Matlab	-	Matlab	Possible	-	T
NeMo + SpikeStream	C	GPU	C, Matlab, C++, PyNN	STDP	-	T

The most sophisticated and also widely used simulator **Emergent** was tried in the first place. It features extensive Graphical User Interface (GUI) and comes with many sophisticated biologically inspired learning algorithms. Simulator also includes build-in virtual environment for direct testing of learned behavior A.6. The downside of Emergent is in the fact that it is one big monolithic application. In order to be able to extend it, I simply was not able to compile it from source on any computer, this can be problem while trying to move the simulator across the computers, or even platforms.

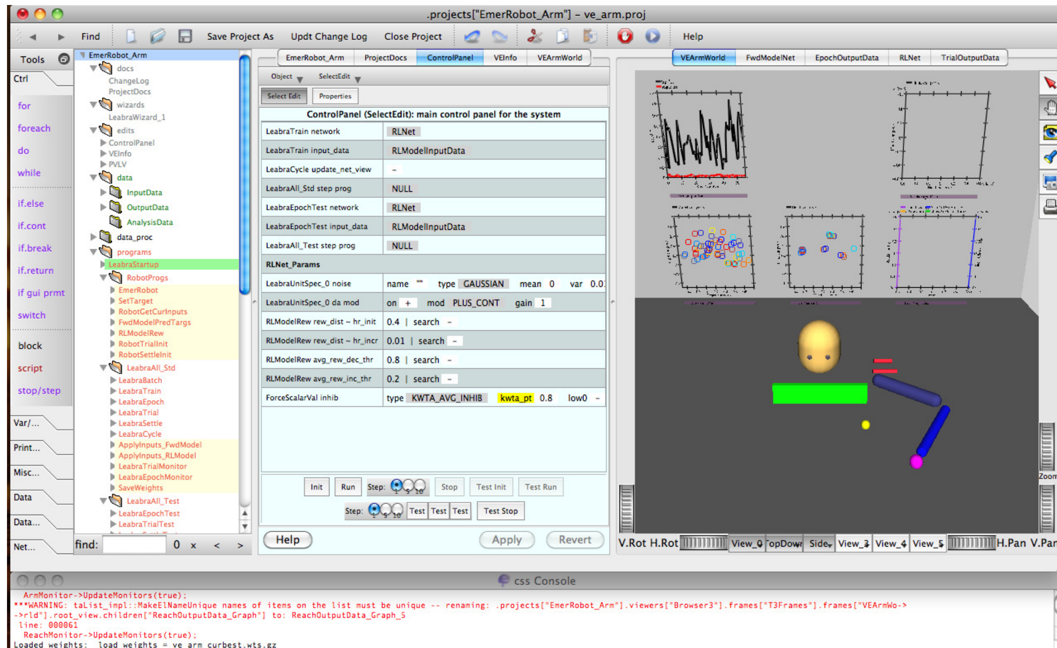


Figure A.6: *Example of experiment in Emergent simulator featuring inbuilt virtual environment.*

The **Biological Neural Network Toolbox** (BNN) for Matlab was used in initial stages for simpler simulations. This simulator does not support any learning mechanism and development of this tool is stopped now. While this simulator does not support any simulation acceleration, it is a good starting point for testing the behavior of neural networks of 3rd generation. Figure A.7 shows asmall hand-wired network of spiking neurons which implements XOR operation. Graphs in the fig.A.8 then show data measured from simulation, blue lines show membrane potential of neurons. Logical value is represented as current on input and as firing rate on network output.

Another tested simulator, **Simbrain** is designed primarily for teaching purposes, supports basic learning algorithms, is very synoptical, lightweight and is implemented in Java. However, this tool also does not support acceleration for larger-scale simulations and the

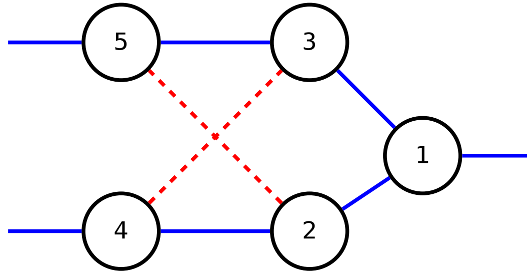


Figure A.7: *XOR implemented by network of Izhikevich's neurons. Network is composed of five Izhikevich's neurons. Neurons 4 and 5 are in input layer, 2 and 3 are hidden and neuron 1 represents output. Blue lines denote excitation connection (positive weights) and red, dashed lines denote inhibition (negative) connections.*

scripting interface is missing too.

Potentially very interesting simulator proved to be **NeMo** (Fidjeland, Roesch, et al. 2009; Thomas and Luk 2009; Wildie et al. 2009). It is simulator of networks of Izhikevich's neurons, which supports Spike-timing-dependent plasticity (STDP) learning and acceleration of simulation on GPU. The simulator is relatively lightweight and is implemented in C language. User can access to the simulation via C, C++ or Python API. If available Compute Unified Device Architecture (CUDA) capable device is found, the simulator uses acceleration, otherwise computing on CPU is used. It was shown that Nvidia CUDA GPU can deliver up to 550 million of spikes per second during the simulation (Fidjeland and Shanahan 2010), this is very suitable for running simulations in real-time. The SpikeStream, C++ graphical front-end for NeMo simulator can be also used for managing the network topologies, measured simulation data and visualization of network behavior. NeMo in combination with SpikeStream was designed to control the humanoid robot iCub (Gamez, Fidjeland, and Lazdins 2012). Their work also shows how to implement biologically plausible encoding of sensory and actuator data for robots. Example of pseudo-randomly generated network of spiking neurons is in the fig.A.9. NeMo simulator was not used because it does not support also 2nd generation of ANNs and also because of the fact that it does not support Java API.

Finally, I chose the open-source Java-based simulator with direct support of both 2nd and 3rd generations of ANNs called Nengo, this simulator is described in more detail in the Chapter 3.4.1.

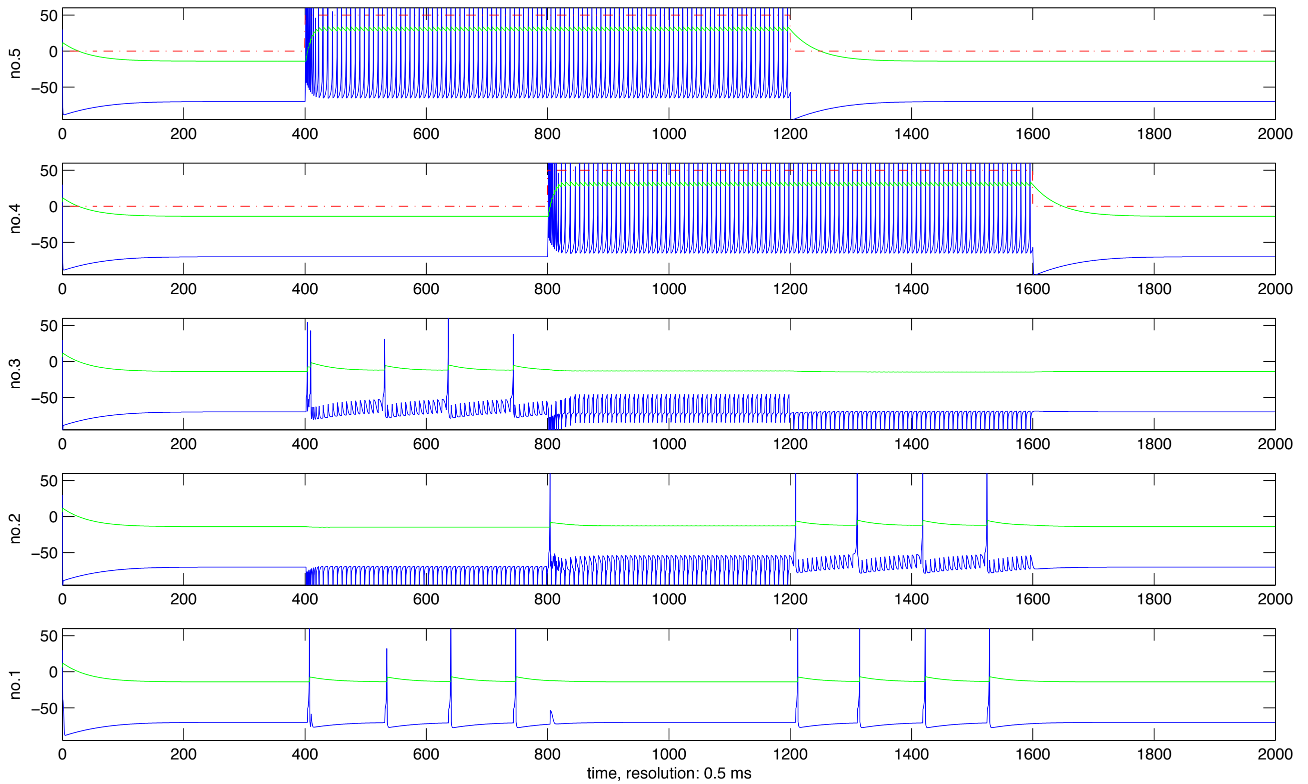


Figure A.8: *Example of simulation of XOR network in Biological Neural Network Toolbox. Red line represents input value (current), blue line is membrane potential (variable v in equations; note spikes) and green line is membrane recovery variable (denoted as u in equations). Logical value is represented as current on input and as firing rate on network output.*

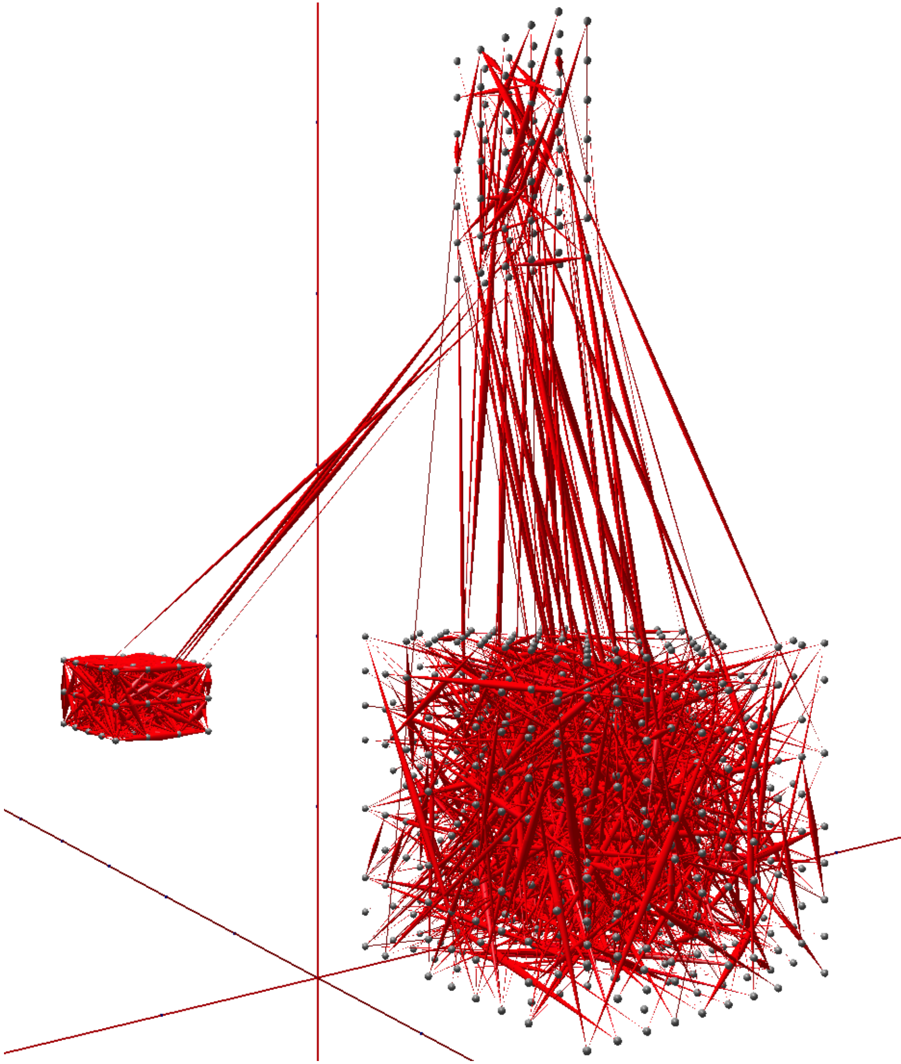


Figure A.9: *Example of real-time simulation of pseudo-randomly generated network of spiking neurons simulated by NeMo and visualized in SpikeStream. SpikeStream allows user to connect neurons in predefined patterns, store and load network topologies and store measured data in the database. Network activity can be visualized in realtime. When injecting noise signal into these neurons, network exhibited emerging synchronization of neuronal firing, as expected (Strogatz 1997).*