DISSERTATION

submitted to the
Combined Faculties for the Natural Sciences and for Mathematics

of the
Ruperto-Carola-University of Heidelberg, Germany

for the degree of
Doctor of Natural Sciences

presented by
Dipl.-Phys. Steffen Gunther Hohmann
born in Braunschweig, Germany

Date of oral examination: 18.05.2005

Stepwise Evolutionary Training Strategies for Hardware Neural Networks

Referees: Prof. Dr. K. Meier
Prof. Dr. F. A. Hamprecht

Schrittweise evolutionäre Trainingsstrategien für neuronale Netzwerke in Hardware

Rein analoge und gemischt analog-digitale Realisierungen künstlicher neuronaler Netzwerke in Hardware entziehen sich für gewöhnlich einer exakten quantitativen Beschreibung. Die Gründe dafür sind die bei der Halbleiterherstellung unvermeidlichen Schwankungen der Bauteilparameter sowie zeitliche Fluktuationen der internen analogen Signale. Evolutionäre Algorithmen eignen sich besonders gut für das Training solcher Systeme, da sie keinerlei detaillierte Informationen über das zu optimierende System benötigen. Um die hohe Arbeitsgeschwindigkeit der neuronalen Netzwerke voll auszunutzen, werden einfache und schnelle Trainingsverfahren benötigt. Im Rahmen dieser Arbeit wurde eine spezielle schrittweise Trainingsmethode entwickelt, die es erlaubt, die synaptischen Gewichte eines gemischt analog-digitalen neuronalen Netzwerkchips unter Zuhilfenahme einfacher evolutionärer Algorithmen auf effiziente Weise zu optimieren. Die vorgestellte Trainingsstrategie wurde an neun verbreiteten standardisierten Aufgabenstellungen für Klassifikationsprobleme getestet: den *breast cancer*, *diabetes*, *heart disease*, *liver disorder*, *iris plant*, *wine*, *glass*, *E.coli* und *yeast* Datensätzen. Es zeigt sich, dass die erreichten Klassifikationsgenauigkeiten sehr gut mit denen von in Software realisierten neuronalen Netzwerken konkurrieren können. Weiterhin sind sie mit den besten Resultaten vergleichbar, die für andere Klassifikationsverfahren in der Literatur recherchiert werden konnten. Die vorgestellte Trainingsmethode begünstigt eine parallele Realisierung und eignet sich darüberhinaus gut zur Verwendung in Kombination mit einem speziell entwickelten Koprozessor, der die zeitaufwendigen genetischen Operationen in einer konfigurierbaren Logik realisiert und damit eine beschleunigte Ausführung evolutionärer Algorithmen ermöglicht. Auf diese Weise kann das entwickelte Trainingsverfahren optimal von der Geschwindigkeit neuronaler Hardware profitieren und stellt daher eine effiziente Methode dar, große neuronale Netzwerke auf dem verwendeten gemischt analog-digitalen Netzwerkchip für anspruchsvolle, praxisrelevante Klassifikationsprobleme zu trainieren.

Stepwise evolutionary training strategies for hardware neural networks

Analog and mixed-signal implementations of artificial neural networks usually lack an exact numerical model due to the unavoidable device variations introduced during manufacturing and the temporal fluctuations in the internal analog signals. Evolutionary algorithms are particularly well suited for the training of such networks since they do not require detailed knowledge of the system to be optimized. In order to make best use of the high network speed, fast and simple training approaches are required. Within the scope of this thesis, a stepwise training approach has been devised that allows for the use of simple evolutionary algorithms to efficiently optimize the synaptic weights of a fast mixed-signal neural network chip. The training strategy is tested on a set of nine well-known classification benchmarks: the breast cancer, diabetes, heart disease, liver disorder, iris plant, wine, glass, *E.coli*, and yeast data sets. The obtained classification accuracies are shown to be more than competitive to those achieved by software-implemented neural networks and are comparable to the best reported results of other classification algorithms that could be found in literature for these benchmarks. The presented training method is readily suited for a parallel implementation and is fit for use in conjunction with a specialized coprocessor architecture that speeds up evolutionary algorithms by performing the time-consuming genetic operations within a configurable logic. This way, the proposed strategy can fully benefit from the speed of the neural hardware and thus provides efficient means for the training of large networks on the used mixed-signal chip for demanding real-world classification tasks.

Meinen lieben Eltern

Contents

Introduction

All men by nature desire knowledge.

Aristotle, Metaphysics

The human brain is one of the most complex systems known to science and understanding its functional principles is a question as old as mankind. Although the brain is far from being completely understood, there is a basic comprehension of its operation on a general level. The brain can be regarded as a highly nonlinear, dynamic, and massively parallel information-processing system that consists of approximately 10^{10} processing units called neurons.

Besides its complexity, the brain is also remarkable for its outstanding capabilities. It can readily perform a large variety of difficult information-processing tasks — like the recognition of familiar faces in unfamiliar environments — much faster and with greater reliability than conventional digital computers. Most notably, however, the human brain has the ability to acquire and store additional knowledge and successfully apply it to the solution of successive tasks, a process which is commonly referred to as learning. At the same time, it contents itself with an average power consumption of only about 20 W.

These astonishing features of the brain are the motivation to build artificial systems that mimic the way in which it performs a particular task of interest. The origins of scientific research on this topic can be traced back to the first half of the 20th century. In 1943, Warren McCulloch and Walter Pitts published a simple binary neuron model that can be used to compute Boolean functions [135], and this work is considered to have founded the research area of artificial neural networks.

Since then, neural networks have established themselves as a powerful computational model and provide an interesting alternative to the Turing paradigm [211] that underlies the operation of common digital computers. It is one of the most important aspects of neural networks that they can be optimized to solve the task at hand by an iterative adaptation process denoted as training. In fact, the existence of a feasible training algorithm is a vital precondition for a neural network model to be of use in practice.

The common method to implement neural networks is to simulate their operation in software. This provides a comparably easy and flexible way of testing and evaluating different neural network approaches, but it actually compromises several initial advantages of neural systems: The principles of neural information processing feature a high degree of inherent parallelism that can only insufficiently

be exploited when the networks are simulated on sequential computers. While this deficiency can partly be compensated for by, e.g., the use of computer clusters, such an approach inevitably leads to a considerable increase in power consumption. Even only a single state-of-the-art microprocessor consumes in the order of 100 W which already exceeds the power consumption of the brain by a factor of five.

These considerations motivate to implement artificial neural networks in a dedicated parallel, low-power hardware and have eventually driven the design of the neural network chip HAGEN (Hardware AnaloG Evolvable Neural network). HAGEN has been developed within the Electronic Vision(s) group of the Kirchhoff-Institute for Physics in Heidelberg and constitutes a dedicated VLSI (Very Large Scale Integration) architecture for the realization of artifical neural networks in a mixed-signal hardware. By combining analog computing with digital signaling, the implemented network model succeeds in reconciling the aim for a low power consumption and a massively parallel network operation with the desire for an easy scalability to larger networks. The current HAGEN prototype features 256 neurons and 32 k synapses and retains an average power consumption of less than 1 W. Using contemporary CMOS (Complementary Metal Oxide Semiconductor) technologies, its underlying concepts allow for the feasible implementation of neural networks in the megasynapse realm.

Besides speed, power consumption, and scalability, the usefulness of an artificial neural network persists to be closely bound to its trainability. In the case of the HAGEN chip, the employed binary neuron model in combination with the temporal noise that inevitably affects the operation of analog circuits impede the direct application of traditional gradient-based neural network training algorithms such as backpropagation. On the other hand, due to its high reconfigurational speed and fast operation, the HAGEN chip promotes the efficient utilization of highly iterative chip-in-the-loop approaches like, e.g., evolutionary algorithms that can automatically deal with these peculiarities.

Evolutionary algorithms apply the principles of natural evolution to the solution of complex optimization problems and are widely accepted as powerful heuristic optimization procedures. They do not require detailed knowledge about the search space they are operating on and are thus particularly well suited for the training of networks on the HAGEN chip[1].

It remains that in order to take full advantage of the neural hardware during training, the algorithm itself needs to be realized efficiently such that it can keep up pace with the networks. The used hardware environment includes a dedicated coprocessor architecture that accelerates evolutionary algorithms by performing the time-consuming genetic manipulations in a configurable logic. While the coprocessor provides efficient means to speed up the training, it imposes certain restrictions on the complexity of the realizable genetic variation operators. In general, the desire for fast algorithm implementations motivates the use of simple algorithms which in turn seems to interfere with the aim of training complex neural networks on the HAGEN chip for challenging tasks. In fact, the usual way

[1] Which eventually inspired its name.

to cope with the difficulties of evolutionary neural network training is to employ more elaborate — and time-consuming — algorithms.

Within this thesis, a stepwise training strategy is investigated that allows for the application of simple and fast evolutionary algorithms to the training of large networks on the HAGEN chip for real-world classification problems. Following this approach, the training procedure is divided into independent steps: Within the individual phases, only parts of the network are trained and optimized towards solving different aspects of the whole problem.

It is demonstrated that each of the single steps can be accomplished by a simple evolutionary algorithm that can readily benefit from the functionality of the evolutionary coprocessor. Apart from that, it is a notable feature of the proposed stepwise approach that the individual training phases can be performed entirely independently. This allows for a high degree of parallelism in the training process that ideally complements a parallel hardware neural network framework.

The stepwise strategy is tested on a set of nine well-known classification benchmark tasks: the breast cancer, diabetes, heart disease, liver disorder, iris plant, wine, glass, *E.coli*, and yeast problems. It is shown that the results are more than competitive to those that are obtained with software-implemented neural networks and are comparable to the performance of the best classification algorithms that have been found for the respective benchmarks in literature.

The thesis is organized as follows. Part I provides an introduction to the basic methodology of neural networks (chapters 1 and 2) and evolutionary algorithms (chapter 3) and discusses the special precautions that are required for the successful combination of these two concepts (chapter 4). A brief overview on the field of hardware-implemented networks is included in chapter 2.

The hardware neural network framework that is used for all presented experiments is introduced in part II: Chapter 5 describes the HAGEN prototype and discusses the concepts that form the basis of its design. In order to train and interface the implemented networks, the HAGEN chip is embedded within a dedicated hardware environment which is presented in chapter 6. As an important part of the used training setup, the chapter in particular includes an introduction to the evolutionary coprocessor. In the course of this thesis, a comfortable software environment has been developed that smoothly integrates with the hardware framework and allows for the easy implementation and testing of different training approaches. This software is the topic of chapter 7.

Part III presents the conducted experiments and a discussion of the obtained results. The basic experimental setup is described in chapter 8 which also presents initial measurements that investigate the general feasibility of simple evolutionary algorithms for the training of networks on the HAGEN chip. Chapter 9 provides a detailed description and evaluation of the proposed stepwise training strategy. Among other things, the performance of the trained networks is compared to those that have previously been reported by other authors. Chapter 10, finally, discusses some additional interesting aspects of the stepwise approach. Most notably, it is demonstrated that the proposed strategy allows for a feasible parallelization of the training process within the used hardware environment.

Part I

Foundations

Chapter 1

Artificial Neural Networks

If the brain were so simple we could understand it, we would be so simple we couldn't.

Lyall Watson

Artificial neural network (ANN) is the generic term for a specific kind of computational model that is inspired by the structure and operation of biological nervous systems, particularly the human brain.

In response to sensory input from the body and its environment, it is the purpose of a nervous system to make appropriate decisions and initiate corresponding reactions that benefit the well-being and survival of the organism. A sponge does not possess any nervous system and is not capable of reacting to changes in its environment. If altered conditions inhibit further supply of nutrients, it is bound to die. A jellyfish, on the other hand, already exhibits a primitive network of neural cells that enable it to actively approach and acquire food [207].

During the course of evolution, biological nervous systems have become increasingly complex resulting in more sophisticated and powerful realizations of the decision making process and a more differentiated behavior of the respective organisms. This eventually lead to the evolution of brains which are not only remarkable for their potential to solve highly complex information-processing problems. More developed neural systems, most notably the brains of human beings, also have the ability to acquire and store additional knowledge which is then advantageously incorporated into succeeding decisions, a process which is commonly referred to as learning. Furthermore, the human brain can perform many information-processing tasks — like the recognition of familiar faces in unfamiliar environments — much faster and with greater reliability than existing computers [84]. At the same time, it retains an average power consumption of only about 20 W.

These outstanding capabilities of the brain are the motivation to build artificial systems that mimic the way in which it performs particular tasks of interest. It is important to note in this context that while the nervous systems of an earthworm or a jellyfish differ considerably from the human brain in structural and behavioral complexity, the underlying basic principles are the same. In general, it is common to all biological information-processing systems that they perform the necessary

computations in an entirely different fashion than conventional digital computers.

As artificial neural networks are more or less simplified models of organic nervous systems, it seems appropriate to start by investigating the most prominent example of biological neural networks in more detail. The following sections give a brief overview of what is known about the basic physiological, organizational and functional principles of the human brain. This introduction cannot be exhaustive, but focuses on those aspects of the brain that seem relevant for the design of artificial neural networks from an engineer's perspective.

1.1 The Human Brain

The human brain is considered as one of the most complicated structures known to science and is far from being completely understood [207]. However, there is a basic understanding of its operation at a low level. The brain is a highly nonlinear, dynamic, and massively parallel information-processing system that consists of approximately 10^{10} processing units called neurons.

1.1.1 The Neuron

Neurons are a remarkable type of cell which are specialized in generating electrical signals in response to chemical and other inputs and propagating them rapidly over large distances to other cells [44]. The brain contains various types of neurons that nevertheless share several essential features (see figure 1.1).

principal constituents

The body of the neuron is called soma and has a diameter of about 10 to 50 μm. Two types of nerve processes are attached to it: The dendrites accept inputs from other neurons via synaptic connections while the axon carries the output signals of the neuron to other cells. The intricate branched structure of the dendrites allows one neuron to receive signals from typically 10^4 other cells.

membrane potential, hyperpolarization and depolarization

A wide variety of ion channels penetrate the cell membrane of a neuron and allow ions (primarily Na^+, K^+, Ca^{2+} and Cl^-) to move in an out of the cell. These channels control the flow of ions through the membrane by opening and closing in response to voltage changes and to both internal or external signals. Ion pumps located in the cell membrane maintain a concentration gradient between the interior of the neuron and the surrounding extracellular space, leading to a corresponding difference in electrical potential. Under resting conditions, the potential inside the cell membrane is approximately -70 mV relative to the environment and the cell is said to be in a polarized state. Electric current in the form of ions flowing through the open channels can either make the membrane potential more negative — a process called hyperpolarization — or less negative and possibly even positive which is called depolarization [44].

action potential

If the membrane potential is depolarized to a given threshold, the neuron generates a so-called action potential which is a fluctuation in the electrical potential across the membrane that lasts for about 1 ms and has an amplitude of approximately 100 mV. These action potentials, also referred to as spikes, are essential for the signal transmission between neurons as they are the only form of membrane potential fluctuations that can propagate over large distances without attenuation.

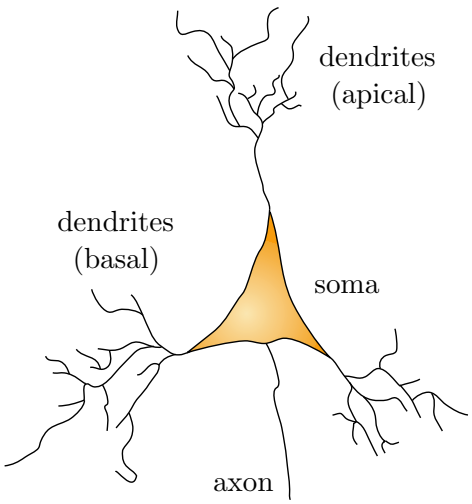


Figure 1.1: *Schematic of a cortical pyramidal neuron. These cells are the primary excitatory neurons in the cerebral cortex of the human brain. The dendrites and the axon are not shown to their full extent. While the dendrites cover an area of up to 400 μm^2 around the neuron, the axon typically expands to far greater length.*

All subthreshold potential fluctuations are severely damped over distances of less than 1 mm.

After an action potential has been evoked, a new spike cannot be emitted for a time span of a few milliseconds known as the absolute refractory period. For a longer interval of up to tens of milliseconds after the initial action potential, the generation of a new spike is more difficult. This is called the relative refractory period. *refractory period*

1.1.2 The Synapse

The axon of a neuron terminates at typically thousands of synapses that connect it to the dendrites of other cells (figure 1.2). For the majority of synapses in the brain, the coupling between the axon on the presynaptic side and the dendrite on the postsynaptic side is a chemical one. An action potential arriving via the axon on the presynaptic side causes the opening of ion channels and the resulting influx of Ca^{2+} leads to the release of neurotransmitters into the synaptic cleft. When these chemicals diffuse through the cleft and bind to receptors at the postsynaptic side, they in turn initiate the opening of ion channels in the dendrite of the signal-receiving cell. *coupling mechanism*

Depending on the nature of the ion flow, the synapse can have a depolarizing, thus excitatory, or hyperpolarizing, hence inhibitory, influence on the postsynaptic neuron. The magnitude of the resulting effect is determined by the amount of neurotransmitter that is released in response to an arriving spike which itself significantly depends on the kind and state of the synapse. While some synapses might convey arriving spikes very efficiently, causing a strong change in membrane potential within the postsynaptic neuron, others might only lead to a negligible effect. *coupling strength*

Usually, one arriving spike does not suffice to raise the membrane potential of a neuron above its threshold. However, if the neuron receives ample excitatory input within a sufficiently short time, the effects can add up and cause the neuron to fire.

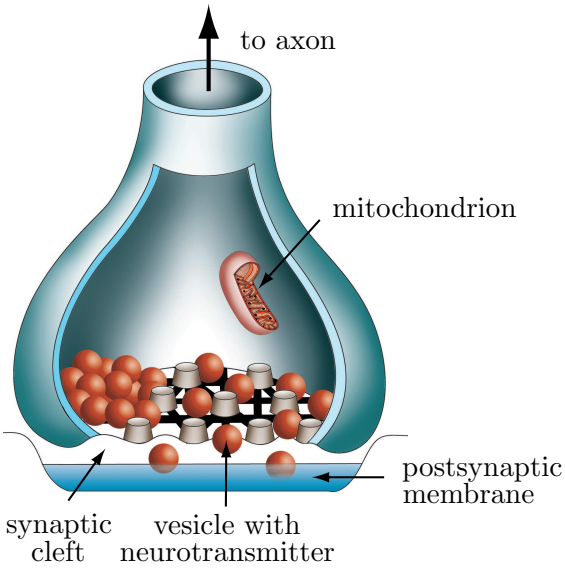


Figure 1.2: *Schematic view of a synapse. An action potential arriving via the axon causes the release of neurotransmitter into the synaptic cleft. The chemicals diffuse through the cleft and bind to receptors on the postsynaptic membrane. (figure taken from [92] with kind permission.)*

1.1.3 Neural Encoding

temporal coding

While the spikes vary only slightly in amplitude, shape and duration, the output of a neuron is in fact coded through the timing of the generated action potentials. Although the activation of a neuron follows an all-or-nothing principle, its output can be regarded as quasi-continuous if it is taken to be the number of action potentials generated within a given time interval. This spiking rate of a neuron can readily be observed and evaluated by experimenters but it is not at all clear that it is also the characteristic quantity used by neurons in the brain to encode and transmit information [69].

rate codes and spike codes

The temporal pattern of action potentials emitted by a group of neurons can in principle code information in various ways, e.g., through correlations, phase relations, or through the explicit order of the firing of specific neurons. These encoding mechanisms are commonly named spike codes in contrast to the rate based information that is accordingly referred to as rate code. The neural encoding is one of the fundamental issues of neuroscience and at present, there is no definite answer as to which are the relevant coding and encoding schemes in biological nervous systems [69] [44].

1.1.4 Learning in the Human Brain

synaptic plasticity

Within the brain, information in the form of spike trains is continuously transmitted and processed by neurons and their interconnecting synapses. The response of a neuron to the total of action potentials that arrive along its dendrites at a given time is determined by the connection strengths of the involved excitatory and inhibitory synapses. Hence, the kind of information-processing that is performed by a whole network of such neurons as well as the information that is stored within it are coded by the entire ensemble of its individual synaptic connection strengths. The efficiencies of synaptic connections are not fixed but can be

changed on different time scales by various processes and this variability is called plasticity.

Hebbian learning rule

One of the basic phenomena that is believed to underly learning and memory in the human brain is the so-called activity-dependent synaptic plasticity. In this case, all changes in the connection strength of a specific synapse only depend on the activities of its presynaptic and postsynaptic neurons. The principles of activity-dependent plasticity were first formulated in 1949 by D.O. Hebb [85]:

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

Hebb proposed this mechanism as a basis of associative learning. He suggested that it would lead to the development of "neuronal assemblies" which reflect the relationships between the input and output patterns of the involved neurons as experienced during the learning period. While the original Hebbian plasticity rule is only concerned with increases in synaptic strength, later versions have been generalized to also incorporate the depression of synaptic efficiency. More general forms regard the synaptic changes to be proportional to the correlation or covariance of the activities of the pre- and postsynaptic neurons [69] [44].

1.2 Neural Network Models

motivation

Regarding the complexity of the human brain on the one hand and its efficiency on the other, there are at least two motivations to model biological neural systems and thus to build artificial neural networks:

- to use them as a research tool for the interpretation and better understanding of neurobiological phenomena
- to use them as information-processing systems that have the potential to reproduce the efficiency, speed, adaptability, and fault tolerance of biological networks

A lot of work has been done to develop adequate network models in pursuit of both aims [44] [69] [84] [88] [118] [169]. As implied at the beginning of this chapter, this thesis deals with networks that are primarily designed for the second purpose.

neural networks for computation

It is not surprising that biologically plausible artificial networks whose dynamics are to be comparable to those of real neural systems have to include reasonably complex models of neurons and synapses [44] [69] [118]. But it turns out that powerful and adaptive information-processing systems can already be built by interconnecting simple processing units that mimic only some basic properties of biological neurons [84] [88] [169]. The term artificial neural network commonly refers to this kind of biologically inspired information-processing systems and the following sections give an overview of their basic principles, the various types and their interesting properties. This thesis largely confines itself to the discussion of what can be denoted as the classical artificial neural network approach. The more biologically motivated spiking network models will not be covered.

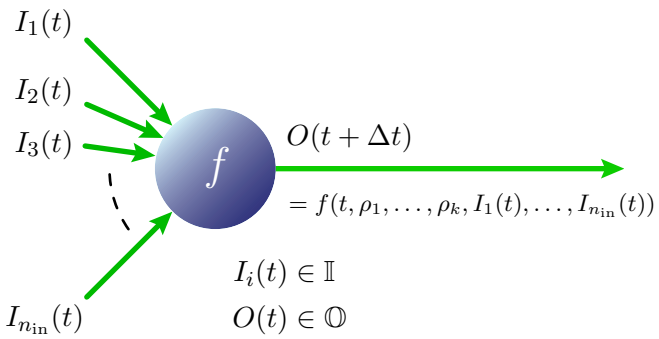


Figure 1.3: *A general neuron model: The neuron transforms its n_{in} inputs $I_i(t)$ into an output $O(t + \Delta t)$ according to a given output function f.*

1.2.1 A General Neuron Model

On an abstract level, the functionality of a neuron can be summarized as follows (see also figure 1.3):

1. At a given point in time t, a neuron receives input signals $I_i(t) \in \mathbb{I}$, $1 \leq i \leq n_{\text{in}}$ via a set of $n_{\text{in}} \in \mathbb{N}$ incoming synaptic connections.

2. Based on its input, the neuron computes its own output $O(t + \Delta t) \in \mathbb{O}$ according to a given output function $f(t, \rho_1, \ldots, \rho_k) \colon \mathbb{I}^{n_{\text{in}}} \to \mathbb{O}$:

$$O(t + \Delta t) = f(t, \rho_1, \ldots, \rho_k, I_1(t), \ldots, I_{n_{\text{in}}}(t)). \tag{1.1}$$

   The resulting output might depend on some inner parameters of the neuron ρ_i with $1 \leq i \leq n_{\text{p}}$ and $n_{\text{p}} \in \mathbb{N}_0$ as well as the time t. It is available at time $t + \Delta t$.

Starting from this basic functional description, various neuron models are conceivable. They differ in the kind of signals that can be transferred between the neurons — i.e., in the sets $\mathbb{I}$ and $\mathbb{O}$ — and the way in which the neuron determines its output as defined by the output function f. Nevertheless, important features of artificial neural networks can already be discussed against the background of this elementary definition.

1.2.2 Networks of Artificial Neurons

Neural networks are formed by interconnecting multiple neurons in a way that the output of one neuron gets connected to $n_{\text{out}} \in \mathbb{N}$ other neurons (fan-out). The latter receive the output of this neuron as one of their several inputs (fan-in). The way in which the neurons of a network are interconnected is called its architecture or topology.

input nodes, output neurons and hidden neurons

If such a network is to be used for a specific information-processing task, information must be fed into it and it must in turn be possible to read back the result of its computation. Usually, some of the neurons do not get input from other

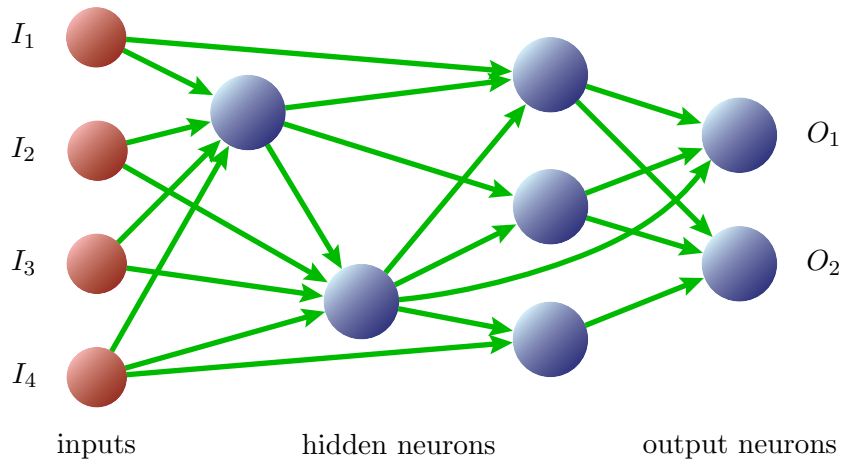


Figure 1.4: *A network of artificial neurons with $N_{\text{in}} = 4$ total inputs and $N_{\text{out}} = 2$ output neurons computes a function $F\colon \mathbb{I}^4 \to \mathbb{O}^2$.*

neurons but receive it from outside. The entirety of their input signals define the inputs of the whole network. Correspondingly, the outputs of some dedicated output neurons are regarded as the response of the system. Such a network can be visualized as a signal-flow graph with directed links and neurons at the nodes which is illustrated in figure 1.4. Neurons that do not represent part of the network output are called hidden neurons.

inputs vs. neurons

In visualizations of neural networks, the sources of external input are commonly drawn in a way similar to that of neurons and are sometimes named input neurons. This seems intuitive as different neurons can receive signals from the same external input and one neuron on the other hand usually accepts inputs from several external sources. Thus, regarding their interconnection with the network, the external input sources do behave like neurons. However, it has to be made clear that these inputs are not in fact neurons according to the above definition. First, they do not receive input via incoming connections but their state is rather set from outside. Second, they do not perform any transformation of their input but transmit their state to the receiving neurons unchanged. They can best be viewed in analogy to the receptor cells of biological nervous systems.

network function

If $N_{\text{in}} \in \mathbb{N}$ is the number of effective input arguments and $N_{\text{out}} \in \mathbb{N}$ the number of output neurons of a network, it computes a function $F\colon \mathbb{I}^{N_{\text{in}}} \to \mathbb{O}^{N_{\text{out}}}$ which is determined by the characteristics and the interconnection of the neurons.

Recurrent vs. Feedforward Networks

recurrent networks

A network can contain feedback loops if the output of a neuron is connected to one of its own inputs directly or via other neurons (figure 1.5 a). In this case, the temporal dimension t introduced in 1.2.1 becomes important as the computation of this neuron is now recursive: It is influenced by its own output in a preceeding time step. Consequently, the output F of the whole network in response to an

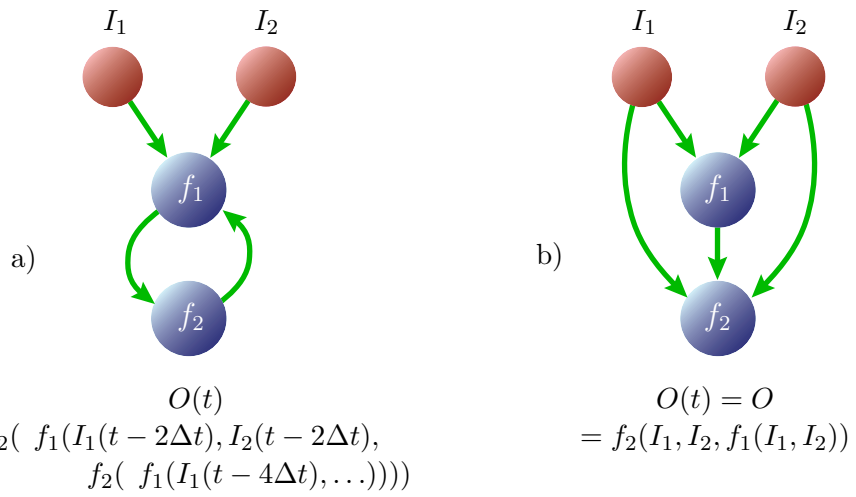


Figure 1.5: ***a)*** *A simple recurrent network: The output depends on the time t and on the initial states of the neurons.* ***b)*** *A feedforward network: The output is time independent and the temporal dimension can be omitted.*

input applied at time $t = 0$ now depends on the initial states of the neurons as well as the time t. Hence, it has to be specified at what time the computation is to be terminated or what is to be taken as the actual response of the network.

Usually, all neurons in a network need the same time Δt to compute their result and this interval can therefore be set to 1 without loss of generality. The network can then be viewed as to operate in discrete steps with $t \in \mathbb{N}_0$ being the number of computed iterations. Networks with feedback connections are commonly called recursive or recurrent networks.

feedforward networks

In contrast, networks that do not contain feedback loops are referred to as feedforward networks (figure 1.5 b)). In this case, the network output in response to an applied input is well-defined. For simplicity, it can thus be assumed that each node performs its calculation instantaneously and Δt can be set to zero without compromising the unambiguousness of the computation. Within such a network, the neurons can always be numbered in a way that a neuron with index i only receives input from neurons with indices $l < i$ and/or external input.

layered networks

Feedforward networks are often — but not necessarily — organized in layers as shown in figure 1.6. Neurons within a layer with index $k > 1$ only receive signals from neurons in layers with lower index $r < k$. Neurons in the first layer only accept external input. It is common to even further constrain the layered structure such that layer $k > 1$ only processes the output of layer $k - 1$. Connections that lead from a neuron in layer k to a neuron of a layer with index $r > k+1$ are called shortcut connections. If all possible connections between two layers are realized, i.e., if each neuron in one layer is linked to every neuron in the other layer, the two layers are said to be homogeneously connected.

The term layer is not consistently used throughout the literature which is due to the varying classifications of the inputs. As stated above, the inputs of the network are sometimes considered as neurons and thus to form a layer in their own right.

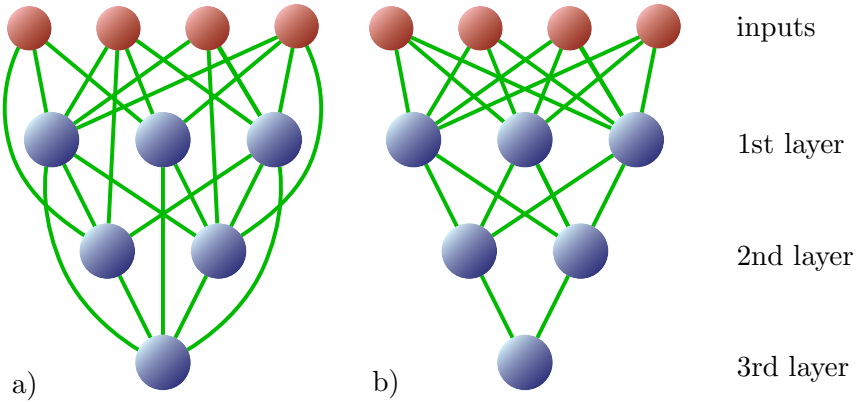


Figure 1.6: ***a)*** *Within a layered network, any neuron in layer k only accepts input from neurons in layers $r < k$ and/or external inputs. The depicted network exhibits several shortcut connections.* ***b)*** *A strictly layered, homogeneously connected network: Each neuron in layer $k > 1$ receives signals from every neuron in layer $k - 1$. No shortcut connections are present and only the neurons of the first layer are linked to the external inputs.*

counting layers

Throughout this thesis, the input fields of the network are not regarded as neurons and are consequently not seen as a separate layer. According to this definition, the networks depicted in figure 1.6 have three layers. This is equivalent to another definition that does not count the layer of neurons in a network but the layers of synaptic links that lead to them. As long as inputs are not regarded as neurons, both definitions are equivalent.

1.2.3 Important Neuron Models

If not otherwise stated, it is understood that the time Δt needed by a neuron to compute its output is the same for all neurons in the network and is either 1 or negligible for recursive and feedforward networks, respectively. During the discussion of some specific neuron models in the succeeding sections, the temporal dimension is therefore omitted.

The Neuron of McCulloch and Pitts

In 1943, Warren McCulloch and Walter Pitts published a paper that introduced a simple model of a binary neuron and which is considered to have founded the research area of artificial neural networks [135]. The output and also the individual inputs of a McCulloch-Pitts type neuron are binary, i.e., they can only be 0 or 1. The neuron can receive inputs from an arbitrary (but finite) number of other sources via synapses that can either be excitatory or inhibitory. If the neuron does not receive non-zero input via any inhibitory connection and if the sum of its excitatory inputs equals or exceeds a given bias value b, it produces an output of 1. In all other cases, the neuron responds with 0. Hence, only one inhibitory signal can compensate any amount of excitatory input which is also referred to as absolute inhibition.

output function

For a McCulloch-Pitts type neuron with threshold b, let $n_{\text{in}} \in \mathbb{N}$ be defined as in 1.2.1 and let $\mathfrak{I}_{\text{in}} \subseteq \{1, \ldots, n_{\text{in}}\}$ be a subset of indices such that an input with index $i, 1 \leq i \leq n_{\text{in}}$ of the neuron is inhibitory if $i \in \mathfrak{I}_{\text{in}}$ and excitatory otherwise. Then its output $O \in \mathbb{O} = \{0, 1\}$ in response to a set of inputs $I_i \in \mathbb{I} = \mathbb{O} = \{0, 1\}$ can be computed as follows:

$$O = f(b, \mathfrak{I}_{\text{in}}, I_1, \ldots, I_{n_{\text{in}}}) = \begin{cases} 0 & \text{if } \exists\, i \in \mathfrak{I}_{\text{in}}, I_i = 1 \\ \Theta\left(\left(\sum\limits_{\substack{1 \leq i \leq n_{\text{in}} \\ i \notin \mathfrak{I}_{\text{in}}}} I_i\right) - b\right) & \text{otherwise} \end{cases} \tag{1.2}$$

where $\Theta(x), x \in \mathbb{R}$ is the theta function defined by

$$\Theta(x) = \begin{cases} 0 & x < 0 \\ 1 & \text{otherwise} \end{cases} \qquad \forall\, x \in \mathbb{R}. \tag{1.3}$$

A network consisting of McCulloch-Pitts type neurons can be regarded as a signal-flow graph with two types of directed links, namely the inhibitory and excitatory connections. These connections, however, are not weighted: All excitatory connections have equal influence on the output of the neuron and the same applies to all inhibitory connections.

In terms of the general definition given in section 1.2.1, the bias value b of the neuron and the set of indices of its inhibitory connections $\mathfrak{I}_{\text{in}}$ can be seen as internal parameters ρ_1, ρ_2 of its output function.

computational capabilities

Associating the numerical value of 0 with the Boolean value *false* and the value 1 with *true*, it is easy to prove by construction that one single McCulloch-Pitts type neuron with two inputs can compute the Boolean functions AND, OR, NOT, NAND and NOR. The first two of them do not require any inhibitory connection. In general, it can be shown that any Boolean function $f_B \colon \{0, 1\}^n \to \{0, 1\}$ can be computed by a corresponding feedforward network of McCulloch-Pitts type neurons with two layers.

Threshold Neurons with Weighted Inputs

Like the biological original, a neuron of the McCulloch-Pitts type accepts two kinds of input: excitatory and inhibitory. It has been discussed in section 1.1.2 that beyond having an excitatory or inhibitory effect, the synaptic connections between neurons in the brain are individually weighted and do not in fact have equal influence on the postsynaptic neuron. The weighting of incoming connections can easily be incorporated by making some small modifications to the McCulloch-Pitts type neuron.

In the modified neuron model, each incoming connection with index j is assigned a synaptic weight $w_j \in \mathbb{R}$ such that an input $I_j \in \mathbb{I} = \mathbb{O} = \{0, 1\}$ arriving via this connection contributes with $w_j I_j$ to the total input of the neuron. The latter is given by the sum over all input signals and is also denoted as the network input

net of this neuron:

$$net = \sum_{j=1}^{n_{\text{in}}} w_j I_j \quad . \tag{1.4}$$

Similar to the McCulloch-Pitts type neuron, the total network input is then compared to the threshold $b \in \mathbb{R}$ and if the latter is equaled or exceeded, the neuron responds with 1 and with 0 otherwise:

$$O = f(b, w_1, \ldots, w_{n_{\text{in}}}, I_i, \ldots, I_{n_{\text{in}}}) = \Theta\left(net - b\right) = \Theta\left(\left(\sum_{j=1}^{n_{\text{in}}} w_j I_j\right) - b\right) \tag{1.5}$$

relative vs. absolute inhibition

Instead of a subset of indices $\mathfrak{I}_{\text{in}}$ that determines the inhibitory connections, the output function now contains the individual synaptic weights w_j as internal parameters. The Inhibitory influence of a single connection j can be realized by choosing an appropriate negative weight w_j. In contrast to the neuron of McCulloch and Pitts, this inhibition is not absolute. Rather, it is relative in the sense that a negatively weighted active connection does not inhibit the neuron completely but effectively increases the bias of the neuron by the absolute value of the respective weight w_j. On the other hand, absolute inhibition of a synaptic connection can be achieved by simply setting its weight to a sufficiently large negative value.

weighted vs. unweighted networks

More general, it can be shown that for every weighted network with relative inhibition where the weights and bias values of all neurons have rational values (thus $w_j, b \in \mathbb{Q}$ for all neurons) there exists an unweighted network of McCulloch-Pitts type neurons that computes the same Boolean function and vice versa [169]. It might be due to this equivalence that "McCulloch-Pitts neuron" is commonly used in the literature as a term for the threshold neuron with weighted inputs described in this section and not for the actual neuron model introduced by McCulloch and Pitts in 1943. In the following, this nomenclature shall be adopted and the term McCulloch-Pitts neuron will relate to the weighted threshold neurons introduced in this section if not explicitly stated otherwise.

the perceptron

Threshold neurons with weighted inputs were first studied by Rosenblatt in 1958 [170] [169] who named them perceptrons (see section 2.1). Hence, the threshold neuron with weighted inputs is also often referred to in the literature as simple perceptron or perceptron.

learning in unweighted networks

There is a substantial reason to prefer weighted networks of threshold neurons to their unweighted pendants: While both are equivalent in their potential to compute arbitrary finite Boolean functions, weighted networks have a critical advantage when adaption and learning are to be implemented. In biological systems, learning is realized primarily by modifying the individual strengths of the synaptic connections. This cannot be done in unweighted networks. Adaption in unweighted networks rather requires the addition or removal of single connections in combination with modifications of the bias values which is more difficult to automate and less biologically plausible.

The topic of learning in artificial neural networks, especially in the case of weighted McCulloch-Pitts neurons or perceptrons, will be discussed in more detail in sections 1.2.4, 2.1.2, 2.1.3, and 2.2.2.

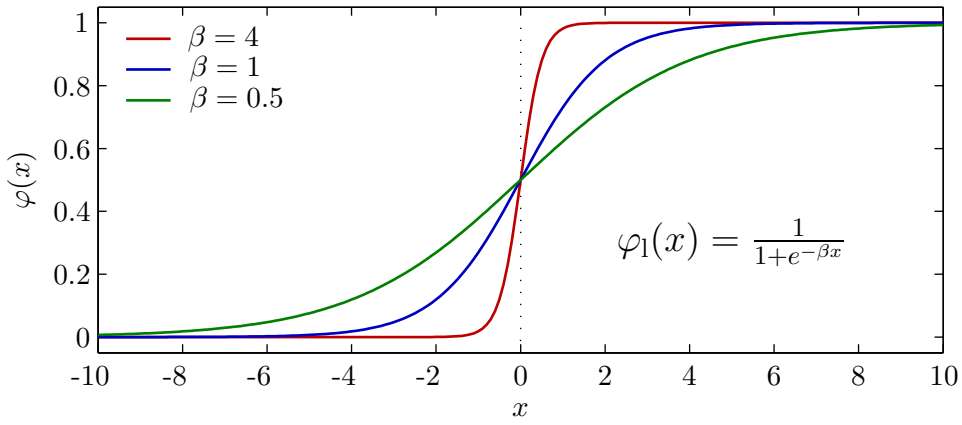


Figure 1.7: *The logistic function $\varphi_l(x)$ for different values of β. As β approaches infinity, $\varphi_l(x)$ converges to the theta function $\Theta(x)$.*

Generalizing Further: Neurons with Continuous Output

continuous output vs. temporal complexity

The binary nature of the original neuron of McCulloch and Pitts has been motivated by the "all-or-none" character of nervous activity [135]. In the meantime it has turned out that, as stated in section 1.1.3, the information which is transmitted between neurons in the brain is far more differentiated. The principles of neural encoding and decoding are in fact based on the temporal organization of the exchanged signals. Even under the simplifying assumption that the relevant quantity is the spiking rate of a neuron, this continuous signal is not adequately represented by a single binary value if the network does not exhibit sufficiently complex temporal dynamics. At least for feedforward networks, whose operation lacks any temporal component, it seems appropriate to make up for this shortcoming by other means.

activation function

Therefore, a further generalization of the neuron model introduced in the preceeding section is to allow for the input and output signals of the neuron to assume continuous values, e.g., by setting $\mathbb{I} = \mathbb{O} = [0, 1]$. In this case, the threshold function $\Theta(x)$ is replaced by a more general function $\varphi(x)$, $\varphi\colon \mathbb{R} \to [0, 1]$, called activation function. The output O of the neuron is then given by the value of its activation function $\varphi(x)$ at the point $net - b$:

$$O = f(b, w_1, \ldots, w_{n_{\text{in}}}, I_i, \ldots, I_{n_{\text{in}}}) = \varphi(net - b) = \varphi\left(\left(\sum_{j=1}^{n_{\text{in}}} w_j I_j\right) - b\right) \tag{1.6}$$

logistic function

One of the most common types of activation functions used for artificial neuron models is the s-shaped sigmoid function

$$\varphi_s(x) = \frac{1}{1 + e^{-x}} \qquad \forall\, x \in \mathbb{R} \tag{1.7}$$

or a more generalized version, the so-called logistic function

$$\varphi_l(x) = \frac{1}{1 + e^{-\beta x}} \qquad \forall\, x \in \mathbb{R} \tag{1.8}$$

with its additional slope parameter β. Figure 1.8 shows the characteristics of some logistic functions with different slope parameter β. It is to be noted that as β approaches infinity, the logistic function converges to the threshold function $\Theta(x)$. The threshold neuron in the preceeding section can thus be regarded as a special case of this generalized neuron model.

antisymmetric functions

For analytical investigations, it is often convenient to consider neurons with an output range of $\mathbb{O} = [-1, 1]$ which can, e.g., be accomplished by appropriately rescaling and shifting the hitherto discussed activation functions. In case of the threshold function $\Theta(x)$, this yields the so-called sign function $\mathrm{sgn}(x)$

$$\mathrm{sgn}(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases} \qquad \forall\, x \in \mathbb{R}. \tag{1.9}$$

Instead of modifying the logistic function $\varphi_{\mathrm{l}}(x)$, it can alternatively be replaced by the hyperbolic tangent function $\varphi_{\mathrm{th}}(x) = \tanh(x) \colon \mathbb{R} \to [-1, 1]$ that also exhibits the desired behavior.

linear neurons

All functions introduced so far map the input space $\mathbb{R}$ onto a limited interval and are therefore denoted as squashing functions. In principle, the output of a neuron does not necessarily have to be restricted to a limited range of values and it is not uncommon to regard neurons with a linear activation function

$$\varphi_{\mathrm{lin}}(x) = \frac{\beta}{4} x + \gamma \qquad \forall\, x \in \mathbb{R}. \tag{1.10}$$

The parameters β and γ are often set to 4 and 0 respectively in which case the activation function $\varphi_{\mathrm{lin}}(x)$ becomes the identity and the neuron is then referred to as a linear neuron.

Any neuron with weighted inputs and a linear activation function effectively computes a functional mapping that is multilinear in all its input arguments. Furthermore, any linear function $l \colon \mathbb{R}^m \to \mathbb{R}^n$ represented by a corresponding $(m \times n)$-matrix with elements l_{ij} can be computed by a single-layer network of linear neurons with bias $b = 0$ if each synaptic weight w_{ij} of the connection leading from input I_i, $1 \leq i \leq m$ to the output with index j, $1 \leq j \leq n$ is set to the corresponding value l_{ij}.

networks of linear neurons

Therefore, feedforward networks of linear neurons can completely be described in the terms of linear algebra. Any composition of linear combinations of linear functions yields a function that is, again, linear [53]. The consequence for neural networks is that for any feedforward network of neurons with linear activation functions and with an arbitrary number of layers there exists an equivalent network with only one layer that computes the same linear function. For this reason, linear activation functions are not well suited for the hidden neurons of a neural network. Rather, linear neurons are often used as outputs. For an output neuron, a nonlinear activation function does not yield any computational benefit but a linear network response is in some cases easier to interpret with regard to the currently investigated problem.

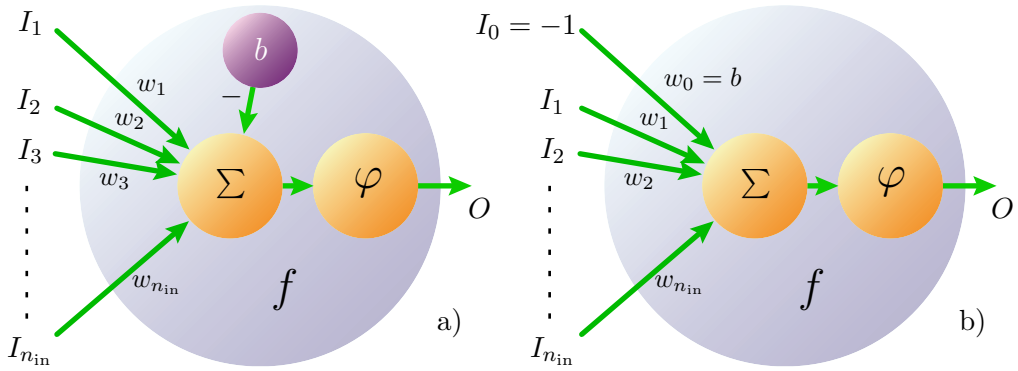


Figure 1.8: ***a)*** *The bias b of the neuron is regarded as a separate internal parameter, the role of which is distinct from those of the weights w_i and the inputs I_i.* ***b)*** *Within the alternative model, the bias is replaced by the synaptic weight $w_0 = b$ of an additional Input I_0 that is constantly set to -1.*

A Note on the Internal Bias

It could be anticipated from equation 1.6 that the bias b of the neuron plays a role which is clearly distinct from that of its inputs I_i and their corresponding weights w_i. However, the expression for the output of the neuron can be simplified by regarding its bias as an additional weight $w_0 = b$ that belongs to an input I_0 which is constantly set to -1. The bias then becomes a part of the total input *net*

$$net = \left(\sum_{j=1}^{n_{\text{in}}} w_j I_j \right) + w_0 I_0 = \sum_{j=0}^{n_{\text{in}}} w_j I_j \quad \text{with } w_0 = b,\ I_0 = -1 \tag{1.11}$$

and the output of the neuron can be written in the more homogenous form

$$O = f(w_0, \ldots, w_{n_{\text{in}}}, I_0, \ldots, I_{n_{\text{in}}}) = \varphi(net) = \varphi\left(\sum_{j=0}^{n_{\text{in}}} w_j I_j \right) \quad \text{with } I_0 = -1. \tag{1.12}$$

networks with and without bias

In other words, for each network of neurons with internal bias there exists a functionally equivalent network of neurons without bias but an additional input I_0 that is constantly set to -1. It is evident that this equally applies to all neuron models that incorporate weighted inputs.

The difference between the two approaches is illustrated in figure 1.8. It might seem purely academic to differentiate between them as they are obviously equivalent. In fact, the second form of the neuron output function f given in 1.12 proves to be advantageous for the formulation of adaptation algorithms as all adjustable internal parameters of f become synaptic weights and can thus be treated in the same fashion.

Alternative Neuron Models

Besides the types of neuron introduced in the foregoing sections, there are at least two more neuron models that, although not directly relevant for this thesis, are worth mentioning for completeness.

stochastic neurons

All preceeding neuron implementations are deterministic in that a given input signal reliably leads to a defined output. A stochastic neuron, on the other hand, assumes either one of its two possible output values (e.g., 0 and 1) with a respective probability that itself depends on the neurons total input. In other words, the activation function of the neuron is no longer regarded as to directly compute its output. Rather, it is given a probabilistic interpretation and now determines the probability $P(net) = \varphi(net)$ for the neuron to fire, i.e., to yield an output of 1:

$$O = \begin{cases} 1 & \text{with probability } P(net) = \varphi(net) \\ 0 & \text{with probability } 1 - P(net) \end{cases} \tag{1.13}$$

Most often, $\varphi(net)$ is taken to be the logistic function 1.8. The slope parameter β can then be interpreted as to control the size of the fluctuations that lead to the stochastic behavior of the neuron. In analogy to thermal fluctuations as covered by statistical physics, β is sometimes regarded as the inverse of a pseudotemperature $T = 1/\beta$. When the pseudotemperature T reaches zero, the neuron becomes a deterministic McCulloch-Pitts neuron. Stochastic neurons are primarily used within a specific type of recurrent network denoted as Boltzmann Machine [2] in honor of the physicist Ludwig Boltzmann.

radial basis function neurons

Another variation of the neuron concept are the so-called radial basis function neurons [146]. Instead of computing their total input as a weighted sum of the single inputs $I_i \in \mathbb{R}$, they take it to be the length of the difference between the input vector $\mathbf{I} = (I_1, \ldots, I_{n_{\text{in}}}) \in \mathbb{R}^{n_{\text{in}}}$ and the weight vector $\mathbf{w} = (w_1, \ldots, w_{n_{\text{in}}}) \in \mathbb{R}^{n_{\text{in}}}$

$$\begin{aligned} net &= ||\mathbf{I} - \mathbf{w}|| \\ &= \sqrt{(I_1 - w_1)^2 + \ldots + (I_{n_{\text{in}}} - w_{n_{\text{in}}})^2} \end{aligned} \tag{1.14}$$

according to the Euclidian norm $||\ .\ ||$. The activation function $\varphi(net)$ is in this case usually referred to as the basis function. A popular choice of basis function is the Gaussian

$$\varphi(net) = \exp -\frac{1}{2}\left(\frac{net}{\sigma}\right)^2 \tag{1.15}$$

with the parameter σ that determines the width of the maximum at $net = 0$.

Neurons with a localized basis function like the Gaussian are sensitive to a region of the input space that is centered around the point $\mathbf{w}$ defined by their synaptic weights. Inputs near this point cause a strong activation of the neuron, vectors that are sufficiently far away from the center of the sensitive region cause only negligible effect.

1.2.4 Modeling Adaptation

motivation

Let N_{in} and N_{out} be the respective numbers of input nodes and output neurons of a neural network with weighted connections. Once its architecture[1] and the used activation function are fixed, the output $\mathbf{O} \in \mathbb{O}^{N_{\text{out}}} =: \Omega$ of the network in response to an input vector $\mathbf{I} \in \mathbb{I}^{N_{\text{in}}} =: \Phi$ is determined by the set of weights w_i of all its internal synaptic connections[2]. The weights can thus be regarded as parameters of the function $F\colon \Phi \to \Omega$ that is computed by this network. The challenge in constructing a neural network for a specific information-processing task is to find the corresponding set of parameter values, i.e., the synaptic weights, that let it compute a given desired function F.

Learning Algorithms: An Overview

learning in artificial neural networks

In some rare cases, it might be possible to define *a priori* the appropriate values of the synaptic weights that have to be chosen for the given problem. Unfortunately, this is infeasible for most tasks of practical relevance. In fact, it is one of the primary advantages of artificial neural networks that they can iteratively adapt their weight values to suit a desired purpose. This is in direct analogy to the adaption properties of biological neural systems (see section 1.1.4) and the process by which the synaptic weights of a network are optimized with regard to its performance on a given task is therefore called learning or training.

learning algorithm: description

During the learning process, input patterns are repeatedly applied to the network and based on its response, modifications are made to its synaptic weights according to a predefined rule. These changes will then cause the network to respond in a different way during the next and successive iterations such that — in the ideal case — its performance will improve over time. A given set of well-defined weight updating rules is called a learning algorithm. Learning algorithms can be classified into two main approaches: supervised and unsupervised learning.

Supervised Learning Following the supervised approach, the output vector $\mathbf{O}$ of the network in response to an input pattern $\mathbf{I}$ is compared to the actually desired target output $\mathbf{T}$. The modifications that are applied to the weights of the network are calculated on the basis of the potential difference between $\mathbf{O}$ and $\mathbf{T}$. Supervised learning therefore requires the specification of the correct answer which is the reason why it is also called learning with a teacher.

error correction learning

The supervised training strategies can further be divided into error correction learning and reinforcement learning. In the case of error correction learning, the difference between the network response $\mathbf{O}$ and the desired output $\mathbf{T}$ is quantified according to some adequate distance measure $E(\mathbf{O}, \mathbf{T})$ which is directly incorporated into the computation of the weight updates. This measure is commonly termed error function.

[1] In case of recurrent networks, defining the maximum number of allowed iterations is regarded as part of fixing the architecture.

[2] Against the background of section 1.2.3, possible bias values can be regarded as weights and do not have to be treated separately.

reinforcement learning

During reinforcement learning, the weights are updated solely on the basis of whether the network output has been right or wrong. But it has to be noted that slightly different and more elaborate definitions of reinforcement learning can be found in literature and that it can also be seen as an unsupervised training approach [84]. The exact definition of the term reinforcement learning is not of major relevance for this thesis and it shall henceforth be used in the above sense.

Unsupervised Learning In contrast to supervised learning, unsupervised training strategies do not require a specification of the correct outputs and are thus useful for tasks where the correct outputs are not known at all. This particularly applies to cases where the goal is not to reproduce a certain output but rather to find and extract correlations within the input data. Clustering is a prominent example for this kind of problem.

biological plausibility

The Hebb-rule introduced in section 1.1.4 represents an unsupervised learning approach as the modifications to the synaptic weights of each neuron depend only on its own activity and the activities of the neurons it receives input from. The updating of the weights does not rely on any externally specified target output. In so far, unsupervised learning can be regarded as the more appropriate model for adaptation in real nervous systems. Nevertheless, while not being as biologically plausible, supervised learning proves to be of exceptional use in numerous practical applications.

The task of neural network training can be extended to incorporate the construction of network architectures and/or the choice of activation functions that are optimal for the problem in question. However, the majority of widespread learning algorithms and particularly all algorithms discussed in this and the next chapter confine themselves to optimizing the weight values of a network. Some approaches for the optimization of the network architecture during training will be discussed in 4.2.

Types of Learning Tasks

As it has already been implied in the preceeding section, the kind of problem that is to be solved by a neural network not only strongly influences the choice of its architecture and neuron model but also the used learning algorithm. Neural networks are successfully applied to a wide range of problems in a large variety of fields and it is difficult to set up a rigid and consistent categorization of the entirety of various learning tasks. But on a general level, one can at least distinguish the following prominent types: function approximation, pattern recognition, pattern association and clustering.

function approximation

For the first example, consider an arbitrary functional mapping

$$\mathbf{y} = g(\mathbf{x}) \qquad \mathbf{x} \in \Phi,\, \mathbf{y} \in \Omega \tag{1.16}$$

where the explicit function g is unknown but is represented by a discrete set $\mathbb{E}_g$ of N_e examples $(\mathbf{x}_\alpha, \mathbf{y}_\alpha)$ with $\mathbf{y}_\alpha = g(\mathbf{x}_\alpha)$ for $1 \leq \alpha \leq N_\mathrm{e}$. The aim is to train the

network such that its corresponding function F approximates the target function g as closely as required. Thus, it is desired that for a given small number $\epsilon > 0, \epsilon \in \mathbb{R}$ the network function F satisfies

$$|| F(\mathbf{x}) - g(\mathbf{x}) || < \epsilon \quad \forall \mathbf{x} \in \Phi \tag{1.17}$$

using an adequate norm $|| \ . \ ||$ defined on Ω. Considering the premises and requirements of this type of problem, it is obvious that supervised learning using the available set of examples $\mathbb{E}_g$ is the adequate training approach. In fact, equation 1.17 already implies a suitable error measure E to be used for correction learning and this will be investigated further in section 2.1.3.

pattern recognition

In pattern recognition, also known as pattern classification, it is the task of the network to correctly allocate each applied input vector $\mathbf{I} \in \Phi$ to one of a predefined set of classes $\mathbb{C} = \{C_1, \ldots, C_{N_\mathrm{c}}\}$. The various classes $C_i \subseteq \Phi$ are mutually exclusive ($C_i \bigcap C_j = \{\}$, $1 \leq i, j \leq N_\mathrm{c}, i \neq j$) and their number N_c is fixed. As in the case of function approximation, networks for pattern classification are trained in a supervised way using a set $\mathbb{E}_\mathrm{c}$ of N_e learning examples $(\mathbf{I}^\alpha, C_\alpha)$, i.e., a set of instances $\mathbf{I}^\alpha$, $1 \leq \alpha \leq N_\mathrm{e}$ for which the respective correct classes C_α with $\mathbf{I}^\alpha \in C_\alpha$ are known.

For each input vector $\mathbf{I}^\alpha$ from the set of training examples and any arbitrary class $C_k \in \mathbb{C}$, the specification of whether the former is a member of the latter — thus whether $C_k = C_\alpha$ — is clearly a binary information, i.e., either true or false. This motivates the use of a reinforcement learning strategy in the sense of the definition given in section 1.2.4.

However, the problem of pattern classification can also be extended to the prediction of the probability $P(C_k \mid \mathbf{I})$ that an instance represented by the vector $\mathbf{I} \in \Phi$ is a member of the class C_k. For each class C_k, this probability can be regarded as a functional mapping $P_k \colon \Phi \rightarrow [0, 1]$, $P_k(\mathbf{I}) = P(C_k \mid \mathbf{I})$ and it can be considered the purpose of the network to approximate this function. Seen from this angle, pattern classification is closely related to function approximation.

pattern association

In the case of pattern association, the network is taught to associate an input vector $\mathbf{I}^\alpha$ with a corresponding output pattern $\mathbf{O}^\alpha$ in the sense that when the trained network is applied an incomplete or distorted version of $\mathbf{I}^\alpha$, it nevertheless returns the learned output $\mathbf{O}^\alpha$. If $\mathbf{I}^\alpha = \mathbf{O}^\alpha$ and thus in fact $\Phi = \Omega$, the problem is also referred to as autoassociation and is named heteroassociation otherwise. While heteroassociation requires supervised learning and can be seen in analogy to the associative memory of the brain, autoassociation commonly involves unsupervised training algorithms and can, e.g., be used for filtering or noise reduction. Both are usually implemented using recurrent networks (see section 2.3.3).

data clustering

Data clustering is perhaps the most obvious example for tasks that rely on unsupervised learning strategies. The problem can be seen as that of a pattern classification task with the additional challenge that the classes C_k are not initially specified. Rather, it is the purpose of the network to extract correlation information within the input data in order to come up with an appropriate partitioning of the input space into clusters by itself. Self-organizing maps (SOMs) are a popular approach to tackle this kind of task [120].

From what has been said, it may be understood that the boundaries between the different types of learning tasks are smooth and in specific cases it might indeed not be possible to unambiguously allocate a given problem to one of them. Nevertheless, most applications of neural networks, from whatever field they might originate, fall in at least one of the above categories. In particular, it has been said that all neural networks implement a specific functional mapping $F\colon \Phi \to \Omega$. Training a network to perform a given information processing task can in this sense always be regarded as some kind of function approximation (see also section 9.3.2).

Generalization

motivation

Within this thesis, emphasis is placed on pattern classification tasks in connection with supervised learning algorithms. As stated above, these are closely related to the solution of corresponding function approximation problems. Whenever a network is trained in a supervised way using a finite set $\mathbb{E}$ of examples, it is in fact the goal of the learning process that the fully trained network will also perform correctly on hitherto unseen input vectors $\mathbf{I} \in \Phi \setminus \mathbb{E}$. The ability to use previously gained knowledge to perform well in present and future decisions under similar but different conditions is called generalization. Thus, the aim of neural network training is to produce a neural network that has good generalization ability.

definition

While this comprehension might appear trivial, it is of great consequence for the construction of neural networks and the formulation of training algorithms, especially with regard to pattern recognition tasks. This is due to the fact that improving the performance of a network on the training data does not in all cases guarantee an improvement in generalization ability but might actually result in a deterioration of the latter.

To understand this phenomenon, it is necessary to examine some explicit network architectures, including their capabilities and limitations, as well as the corresponding learning algorithms more closely. The following chapter is devoted to the important class of feedforward networks.

Chapter 2

Feedforward Neural Networks

It's a poor sort of memory that only works backward.

Lewis Carroll

Feedforward architectures that do not contain any feedback loops form an important class of artificial neural networks. A considerable variety of theoretical and experimental investigations have been performed in order to illuminate their computational capabilities and to devise efficient algorithms for their training.

The experiments and novel training strategies presented in part III of this thesis concern themselves with hardware-implemented, strictly layered, feedforward networks for pattern classification tasks. Therefore, the aim of this chapter is to lay a solid foundation for the discussion of the achieved results by summarizing what is known about the general properties of this class of networks, expecially with regard to pattern categorization problems.

Beyond that, section 2.3 gives a brief overview of some alternative network topologies and paradigms that have proven to be suitable approaches for the solution of classification problems as well. Finally, the chapter is to be concluded with a discussion of hardware-implemented neural networks and the special requirements that have to be met by suitable algorithms for their training.

2.1 Single-Layer Feedforward Networks

the perceptron

The following sections address feedforward networks that exhibit only one single layer of weighted connections. Networks of this kind were first studied by Rosenblatt in 1958 [170]. Originally, he investigated threshold neurons that received weighted input from a number of receptive neurons that were each sensitive to a different region of an artificial retina (see figure 2.1). In this initial version, the connections between the retina and the receptive neurons were deterministic and fixed, whereas the weighted connections to the threshold neurons were initialized randomly and tuned during training. Rosenblatt called this system perceptron.

the simple perceptron

In the following years, the model was refined further [22] and eventually simplified and studied in more detail by Minsky and Papert [143]. Within their model,

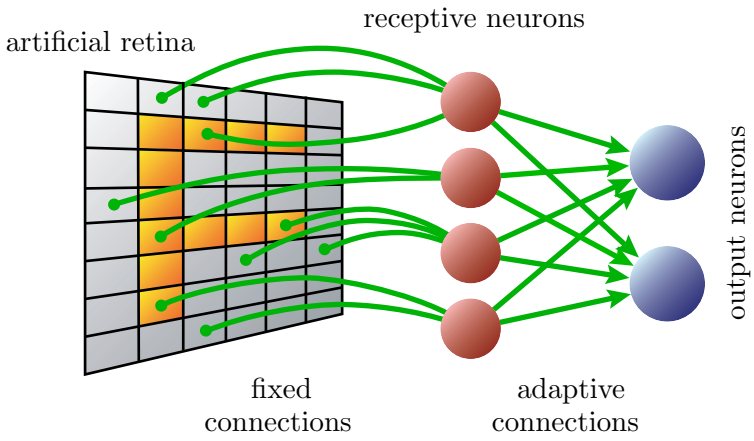


Figure 2.1: *Schematic of the original perceptron (Rosenblatt 1958 [170]): A group of receptive neurons are connected to different sets of receptive fields on an artificial retina. Their outputs are sent to threshold neurons with weighted inputs whose connection strengths are initialized randomly and adapted during training.*

the so-called simple perceptron is reduced to a single threshold neuron that receives input signals via weighted connections from a set of binary predicates.

single-layer and multi-layer perceptrons

In more recent literature, the name perceptron is commonly used for any feedforward network consisting of neurons that calculate their total input *net* through a weighted sum of their input signals I_j. If the network has only one layer, it is also referred to as a single-layer perceptron and if this single layer contains only one neuron, it is named simple perceptron.

2.1.1 Capability of the Simple Perceptron

Consider a simple perceptron, more specifically, a single threshold neuron with weighted inputs whose individual input signals $I_j \in \mathbb{I} = \mathbb{R}$ are continuous while its output $O \in \mathbb{O} = \{0, 1\}$ is limited to binary values. Figure 2.2 a) shows an example with two real inputs I_1, I_2 and a virtual third input I_0 that is constantly set to -1 in order to implement a bias $b = w_0$. The two-dimensional input space $\mathbb{R}^2$ of the neuron can unambiguously be divided into two distinct sets $\mathbb{S}_0 \bigcap \mathbb{S}_1 = \emptyset$, $\mathbb{S}_0 \bigcup \mathbb{S}_1 = \mathbb{R}^2$ which contain those pairs of input values (I_1, I_2) that cause an output of 0 and those which lead to an output of 1, respectively.

linear region boundary

For the weight values chosen in figure 2.2 a), the two resulting regions $\mathbb{S}_0$ and $\mathbb{S}_1$ are illustrated in figure 2.2 b). According to equations 1.3 and 1.12 the boundary between them is a straight line defined by

$$I_0\, w_0 + I_1\, w_1 + I_2\, w_2 = 0. \tag{2.1}$$

Using $I_0 = -1$ and $w_0 = b$, this immediately becomes

$$I_1 w_1 + I_2 w_2 = b \tag{2.2}$$

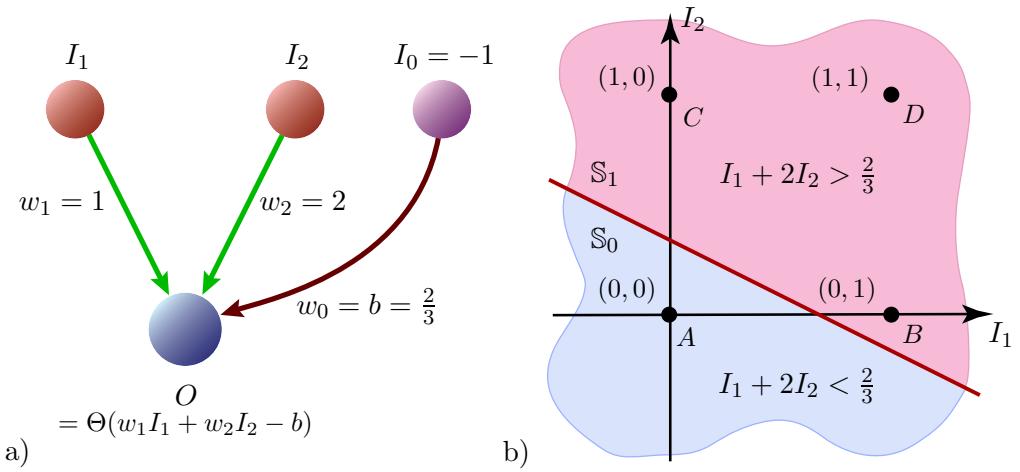


Figure 2.2: *A simple perceptron with two inputs (a) divides its two-dimensional input space into two distinct regions that are separated by a straight line (b) which is defined by $I_1 w_1 + I_2 w_2 = w_0$. Inputs that correspond to points in the lower region $\mathbb{S}_0$ lead to an output of 0, those from the upper part $\mathbb{S}_1$ (including the boundary) yield a response of 1.*

which can be transformed to the linear correlation

$$I_2 = \frac{b}{w_2} - I_1 \frac{w_1}{w_2} \tag{2.3}$$

depicted in figure 2.2 b). A little more algebra reveals that the hereby defined region boundary is perpendicular to the vector $\mathbf{w} = (w_1, w_2) \in \mathbb{R}^2$ and that its normal distance from the origin is given by

$$l = \frac{|b|}{\sqrt{w_1^2 + w_2^2}}. \tag{2.4}$$

Hence, while the orientation of the separation is solely defined by the individual values of w_1 and w_2, its position in the input plane can be determined by specifying the bias b. Through an adequate choice of these parameters, any linear partitioning of the input space $\mathbb{R}^2$ can be achieved.

hyperplane in n dimensions

This result can be generalized to an arbitrary number of n_{in} inputs, in which case a simple perceptron with weight vector $\mathbf{w} \in \mathbb{R}^{n_{\text{in}}}$ and bias b divides the input space $\mathbb{R}^{n_{\text{in}}}$ into two regions that are separated by an $(n_{\text{in}}-1)$-dimensional hyperplane $h \subset \mathbb{R}^{n_{\text{in}}}$. The normal vector $\mathbf{n}_h$ of this hyperplane is given by

$$\mathbf{n}_h = \frac{\mathbf{w}}{||\ \mathbf{w}\ ||} \tag{2.5}$$

while its normal distance from the origin l_h can be calculated as

$$l_h = \frac{|b|}{||\ \mathbf{w}\ ||}, \tag{2.6}$$

where $||\ .\ ||$ denotes the Euclidian norm.

extended weight vector

In equations 2.5 and 2.6, the bias of the simple perceptron is once more treated separately from its weights. Still, in the spirit of section 1.2.3 and figure 2.2 a), all parameters of the neuron, i.e., its weights and bias, can be regarded as components of a single extended weight vector $\widetilde{\mathbf{w}}$

$$\widetilde{\mathbf{w}} = (w_0, w_1, w_2, \ldots, w_{n_{\text{in}}}) \qquad \text{with } w_0 = b \tag{2.7}$$

and each input pattern $\mathbf{I}$ can be represented by an extended input vector

$$\widetilde{\mathbf{I}} = (-1, I_1, I_2, \ldots, I_{n_{\text{in}}}), \tag{2.8}$$

extended input vector

where I_0 has already been replaced by a constant of -1. In this representation, the perceptron again defines an n_{in}-dimensional hyperplane $\widetilde{h}$ within its extended input space $\mathbb{R}^{n_{\text{in}}+1}$, but this time, the orientation of the hyperplane is defined by the extended weight vector

$$\widetilde{\mathbf{n}}_h = \frac{\widetilde{\mathbf{w}}}{|| \widetilde{\mathbf{w}} ||} \tag{2.9}$$

which in particular includes the bias $w_0 = b$. By definition, the new hyperplane $\widetilde{h}$ expands through the origin, hence $\widetilde{l}_h$ is always 0. In turn, all extended input vectors now lie in the n_{in}-dimensional subspace $\mathbb{P}_0^{n_{\text{in}}}(-1) \subset \mathbb{R}^{n_{\text{in}}+1}$ given by

$$\mathbb{P}_0^{n_{\text{in}}}(-1) = \{(I_0, I_1, \ldots, I_{n_{\text{in}}}) \in \mathbb{R}^{n_{\text{in}}+1} \mid I_0 = -1\} \tag{2.10}$$

and thus have a minimum distance to the origin of 1.

extended input space

In this sense, the original input plane shown in figure 2.2 b) could be seen as a two-dimensional slice $\mathbb{P}_0^2(-1)$ through the extended input space $\mathbb{R}^3$ at $I_0 = -1$. Another two-dimensional plane $\widetilde{h} \subset \mathbb{R}^3$ is defined by the perceptron via equation 2.1 and its intersection with $\mathbb{P}_0^2(-1)$ results in the linear separation that is shown in figure 2.2 b). It should be repeated that although $\widetilde{h}$ must pass through the origin of $\mathbb{R}^3$, its intersection with the original input plane $\mathbb{P}_0^2(-1)$ can yield any desired linear separation of the latter if its orientation $\widetilde{\mathbf{n}}_h$ is changed accordingly.

This view of the situation might seem rather unintuitive but as stated before, it simplifies matters in so far as the hyperplane $\widetilde{h}$ in the extended input space depends on all parameters w_i of the neuron in a congenerous way. It persists, that independent of the actual treatment of its bias, the perceptron divides the extended as well as the original input space into two regions which are separated by a respective linear subspace.

Linear Separability

Two sets C_1 and C_2 of points $\mathbf{I}$ in the n_{in}-dimensional space $\mathbb{R}^{n_{\text{in}}}$ are called linearly separable, if there exist $n_{\text{in}} + 1$ real numbers $w_0, w_1, \ldots, w_{n_{\text{in}}}$ such that

$$\sum_{i=1}^{n_{\text{in}}} w_i I_i \geq w_0 \quad \forall\, (I_1, \ldots, I_{n_{\text{in}}}) \in C_1 \quad \text{and} \quad \sum_{i=1}^{n_{\text{in}}} w_i I_i < w_0 \quad \forall\, (I_1, \ldots, I_{n_{\text{in}}}) \in C_2. \tag{2.11}$$

This is equivalent to the existence of an $(n_{\text{in}}-1)$-dimensional hyperplane that separates all points in C_1 from those in C_2.

Correspondingly, a bool-valued function $f_\mathrm{b}\colon \mathbb{R}^{n_\mathrm{in}} \to [0,1]$ is denoted linearly separable if the set $B_0 = \{(I_1,\ldots,I_{n_\mathrm{in}}) \mid f_\mathrm{b}(I_1,\ldots,I_{n_\mathrm{in}}) = 0\}$ is linearly separable from the set $B_1 = \{(I_1,\ldots,I_{n_\mathrm{in}}) \mid f_\mathrm{b}(I_1,\ldots,I_{n_\mathrm{in}}) = 1\}$.

capability of the simple perceptron

It can be concluded from the results of the preceeding paragraphs that when a simple perceptron of the above definition is to be used for a pattern classification problem with two classes C_1 and C_2, it can only reliably distinguish between them if they are linearly separable. Also, if the perceptron is in fact a McCulloch-Pitts neuron that only accepts binary inputs, it can only compute linearly separable Boolean functions. It is worthwhile to investigate the consequences of this result in more detail.

Linearly Separable and Inseparable Problems

linear separable Boolean functions

Let, for a moment, the simple perceptron in figure 2.2 a) be restricted to accept only binary values $I_i \in \{0,1\}$. It then effectively becomes a McCulloch-Pitts neuron and the input space is reduced to the points A, B, C and D in figure 2.2 b) which correspond to the four possible combinations of two Boolean values. Adopting the specified weights, the network computes the Boolean function OR: Its output is 1 in all cases except if both of its inputs equal 0 (point A). It is evident that the extended weight vector $\widetilde{\mathbf{w}}$, i.e., the weights and the bias of the perceptron, could be changed such that the region boundary shown in 2.2 b) is shifted to separate points A, B and C in $\mathbb{S}_0$ (blue) from the input D in $\mathbb{S}_1$ (red). In this case, the perceptron would compute the Boolean AND. Both, AND and OR, are clearly linearly separable.

the Boolean XOR

Consider, however, the Boolean function XOR ("exclusive or") that yields a value of 1 for the points B and C while it returns 0 for the inputs A and D. Figure 2.2 b) suggests that there is no linear partitioning of the input plane that performs the desired separation. This assumption is indeed true and can easily be proven by contradiction [143] [169]. In other words, the XOR function is linearly inseparable and can therefore not be computed by a simple perceptron.

N-bit parity

The generalization of XOR to more than two inputs is the so-called N-bit parity function $f_\mathrm{p}^N\colon \{0,1\}^N \to \{0,1\}$. It accepts N binary inputs I_i, $1 \leq i \leq N$ and returns 1 only if the number of inputs with value 1 is odd:

$$f_\mathrm{p}^N(I_1,\ldots,I_N) = \begin{cases} 1 & |\{I_j \mid 1 \leq j \leq N, I_j = 1\}| \equiv 1 \ mod\ 2 \\ 0 & \text{otherwise} \end{cases} . \tag{2.12}$$

Like XOR, the N-bit parity function is linearly inseparable and is the most famous example for functions that cannot be solved by single-layer feedforward networks [143]. In general, if N is the dimension of the input space — corresponding to the number of inputs to the perceptron — there are 2^N possible input patterns and hence 2^{2^N} possible partitionings of these patterns into two classes. Each of these partitionings corresponds to one Boolean function $f_\mathrm{b}\colon \{0,1\}^N \to \{0,1\}$. The percentage of those functions that are linearly separable can be shown to converge to 0 with increasing N [20] [149] [169]. Already for $N = 3$, only 104 of the 256 possible Boolean functional mappings exhibit linear separability.

This result seems rather devastating as it suggests that the applicability of a simple perceptron is limited to only a negligible fraction of problems. The revelation of these limitations by Minsky and Papert in 1969 [143] indeed lead to a strong decrease of interest in artificial neural networks for several years.

Boolean functions vs. pattern recognition

It has to be noted that Boolean functions, while providing sound means of estimating the basic capabilities of specific neural network architectures, are only of minor interest to the practitioner who desires neural networks to generalize well on realistic problems. Reliably modeling an arbitrary Boolean function involves the learning of every single input-output mapping and the concept of generalization (see section 1.2.4) cannot reasonably be applied at all. In case of XOR and parity, the smallest possible change in the input pattern — changing one binary input from 0 to 1 or vice versa — leads to the largest possible change in the output. Most realistic pattern recognition problems show the opposite behavior, i.e., small changes in the input do in most cases lead to only minor changes in the output [20].

Therefore, the restriction to binary input values shall now be lifted again and the perceptron will be used to solve the three classification problems illustrated in figure 2.3. As it is common for classification problems, the classes are in each case represented by a finite set of examples with the instances of classes C_1 and C_2 being marked by crosses and circles, respectively. The shown class structures are simplified but capture some typical characteristics of realistic pattern recognition problems (see section 9.4).

linearly separable and inseparable classes

In figure 2.3 a), position and shape of both classes are such that they can reliably be separated by a simple perceptron. While this is not possible for the classes shown in figure 2.3 b), the given linear partitioning of the input space can at least

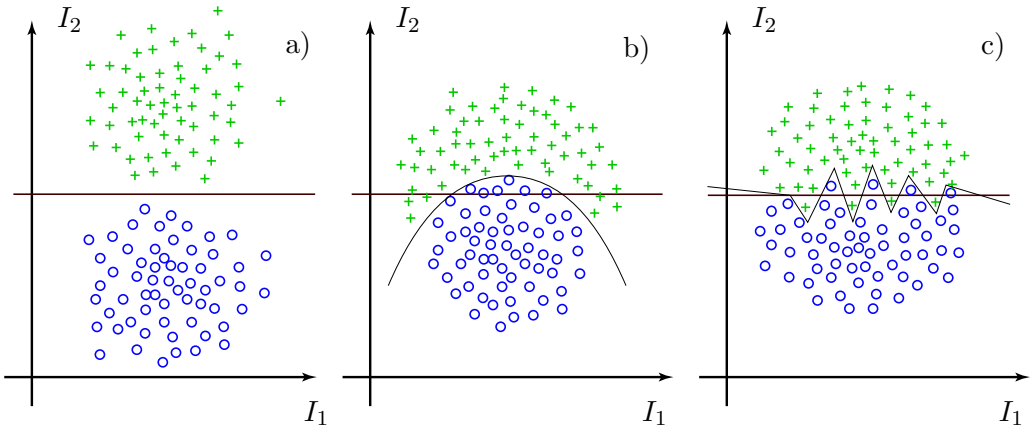


Figure 2.3: ***a)*** *The two classes are linearly separable, i.e., their members can reliably be told apart by a linear partitioning of the input plane.* ***b)*** *In this case, the thin curve yields a better classification accuracy. Still, the linear separation is a good approximation.* ***c)*** *Here, the classes partly overlap. As each class is represented by only a finite number of instances, they can completely be separated by a sufficiently complex curve (thin line) but the linear approach promises to achieve better generalization.*

be regarded as a reasonable first-order approximation of the real solution and would in this case lead to a fraction of misclassifications of only 10 %. Still, the curved thin line represents a far more suitable separation of the input space for this classification problem.

overlapping class regions

The two classes shown in figure 2.3 c) partly overlap. But as they are each represented by only a finite number of instances, there exist intricate non-linear separations of the input plane that seem to distinguish between them with 100 % accuracy. One of these possible separations is again illustrated as a thin black line. If the individual instances of the two classes are spread over the input space according to some statistical distribution, it is most likely that additional instances generated by the same underlying mechanism would not be classified correctly by the shown specialized partitioning. In that case, the superiority of the non-linear separation to the linear one would actually be an illusion caused by the use of a finite set of examples. The shown linear partitioning, although not perfectly classifying the present instances, clearly provides a more adequate separation of the classes and is likely to yield a better generalization performance.

These reflections motivate that a simple perceptron with its limited computational capabilities might nevertheless prove to be of value for the solution of pattern recognition problems if the respective class structures exhibit characteristics similar to those of the above examples. It is reasonable to wonder whether this does in fact apply to any realistic tasks. A thorough investigation of this topic will be the subject of chapter 9.

2.1.2 Training the Simple Perceptron

During the initial years after its introduction, the perceptron attracted high interest mainly because of the availability of a well-defined and effective training algorithm that, for the first time, allowed an artificial network to be used as a learning machine.

As a simple perceptron consists of only one neuron, the number of inputs to the network N_{in} is equivalent to the number of inputs to the neuron $n_{\text{in}} = N_{\text{in}}$ and the number of outputs N_{out} equals 1. For simplicity, it is assumed that the neuron exhibits no internal bias. It is recalled from section 2.1.1 that this can always be achieved by discussing the problem in terms of an extended weight vector $\widetilde{\mathbf{w}}$ and corresponding extended input vectors $\widetilde{\mathbf{I}}$ (see equations 2.7 and 2.8).

goal of the training

It has also been shown that a simple perceptron linearly divides its input space into two regions and can thus in principle solve any linearly separable classification problem with two classes C_1 and C_2. The network is to be trained for one such task using a finite training set $\mathbb{E}$ of input patterns $\widetilde{\mathbf{I}}^{\alpha}$ that are each associated with the respective desired target output T^{α}. Given this finite set of examples, it is the purpose of the training algorithm to find a linear separation (represented by a corresponding weight vector $\widetilde{\mathbf{w}}$) that yields a correct classification of all instances $\widetilde{\mathbf{I}}^{\alpha} \in \mathbb{E}$.

To keep the formulation of the training algorithm simple, it will turn out to be advantageous if one considers a threshold neuron that does not respond with 0 or 1 but with -1 or 1 instead. As stated in section 1.2.3, this transition is

straight forward — it merely involves a shifting and rescaling of the output — and effectively turns the threshold activation function $\Theta(x)$ into the sign function $\mathrm{sgn}(x)$ (equation 1.9). The learning task shall then be formulated such that input patterns $\widetilde{\mathbf{I}}$ belonging to class C_1 are desired to result in a network response of $T = 1$ and those of class C_2 are to yield an output of $T = -1$.

perceptron learning algorithm

The perceptron learning algorithm [22] starts off by initializing the weights of all connections with random values. It then proceeds by iteratively cycling through all training examples $\widetilde{\mathbf{I}}^\alpha$, presenting them to the network and comparing the respective network response O^α to the corresponding target output T^α: If $O^\alpha = T^\alpha$ the training immediately continues with the next training pattern. Otherwise, the weight vector $\widetilde{\mathbf{w}}$ is changed by a small vector $\Delta\widetilde{\mathbf{w}}^\alpha$ which is proportional to the original input vector $\widetilde{\mathbf{I}}^\alpha$. The weight change $\Delta\widetilde{\mathbf{w}}^\alpha$ shall be added to the old weights $\widetilde{\mathbf{w}}$, if the target output $T^\alpha = 1$ and be subtracted if $T^\alpha = -1$. At this point, the choice of $\{-1, 1\}$ for the possible output values pays off, as the modification of the weight vector after the application of each input pattern $\widetilde{\mathbf{I}}^\alpha$ can conveniently be formulated as

$$\widetilde{\mathbf{w}}^{\mathrm{new}} = \widetilde{\mathbf{w}}^{\mathrm{old}} + \Delta\widetilde{\mathbf{w}}^\alpha \tag{2.13}$$

with

$$\Delta\widetilde{\mathbf{w}}^\alpha = \frac{1}{2}\eta\,(T^\alpha - O^\alpha)\,\widetilde{\mathbf{I}}^\alpha \tag{2.14}$$

where the constant η is an adjustable parameter of the algorithm denoted as learning rate. This learning algorithm is an example for supervised training.

Discussing the Perceptron Learning Algorithm

perceptron convergence theorem

According to an important result commonly known as the perceptron convergence theorem, the perceptron learning algorithm is guaranteed to find a complete solution of a given classification task within a finite number of iterations under the condition that the problem can actually be solved by a perceptron, i.e., is linearly separable [22] [88] [143].

geometrical discussion

While the proof of this theorem shall not be reproduced here in detail, the weight updating scheme 2.14 can be motivated by some simple geometrical considerations [88] [20]. In the extended input space, the perceptron defines a hyperplane $\widetilde{h}$ that passes through the origin. Its orientation is given by the normal vector $\widetilde{\mathbf{n}}_h$ which in turn is determined by the weight vector $\widetilde{\mathbf{w}}$ by virtue of equation 2.9. According to this definition, an input vector $\widetilde{\mathbf{I}}$ results in an output of 1 if its projection onto $\widetilde{\mathbf{w}}$ is positive and yields an output of -1 if the latter is negative. Thus, if a vector $\widetilde{\mathbf{I}}^\alpha$ is incorrectly classified, its projection $\widetilde{p}_h^\alpha = \widetilde{\mathbf{n}}_h\widetilde{\mathbf{I}}^\alpha$ onto the direction of $\widetilde{\mathbf{w}}$ simply has the wrong sign which can iteratively be corrected by slightly changing the orientation of $\widetilde{\mathbf{w}}$ into the appropriate direction. If the sign of $\widetilde{p}_h^\alpha$ is false negative, $\widetilde{\mathbf{w}}$ has to be rotated towards $\widetilde{\mathbf{I}}^\alpha$, if it is false positive, $\widetilde{\mathbf{w}}$ must be changed in the opposite direction. The learning rule 2.14 is the mathematical formulation of this strategy.

training result

It is worth pointing out that the algorithm reaches a stationary weight vector $\widetilde{\mathbf{w}}$ — and can be regarded as terminated — once all training patterns are classified correctly. However, as the number of training instances is finite, the solution is not unique and multiple runs with different random initializations will most likely result in different final weight vectors $\widetilde{\mathbf{w}}$ which will nevertheless perform equally well on the training set. In particular, the result of a specific training run is not guaranteed to yield the best possible result with regard to generalization.

linearly inseparable case

If the two classes are not linearly separable at all, there will at each stage of the training be at least one input vector $\widetilde{\mathbf{I}}^{\alpha}$ that is not correctly classified. Therefore, the algorithm will in principle continue to adjust the weights infinitely, resulting in an ever increasing weight vector $\widetilde{\mathbf{w}}$. When the algorithm is eventually terminated manually or due to another suitable abortion criteria, the resulting network is not necessarily optimal in any sense, especially not in terms of generalization.

The perceptron learning rule nevertheless presents an effective training approach for single-layer networks and linearly separable problems. Unfortunately, it turned out that it cannot be transferred to multi-layer networks (section 2.2) which motivated the design and study of alternative learning algorithms.

2.1.3 Continuous Outputs and Gradient Descent

The discrete output of the simple perceptron discussed in the preceeding sections shall now be exchanged for a continuous response like generated by one of the squashing functions discussed in 1.2.3. For a start, let the output of the neuron lie in the interval $[0, 1]$ and let it be computed by the logistic function φ_{l} (equation 1.8) with a slope parameter of $\beta = 2$. If the response $O(I_1, I_2)$ of the simple network in figure 2.2 a) is then plotted as a function of its two inputs, one obtains the graph shown in figure 2.4.

linear separation

The input plane can in some sense still be regarded as linearly separated into two regions, one corresponding to an output near 1 and the other to a response close to 0, but the transition between them is now gradual and no longer discontinuous. Apart from that, all observations stated in section 2.1.1 concerning the position and orientation of the linear separation still hold. As before, the transition to the extended input space and the extended weight vector $\widetilde{\mathbf{w}}$ is straight forward.

probabilistic interpretation

Regarding a classification task with two classes C_1 and C_2, the output O^{α} of the network in response to an input $\widetilde{\mathbf{I}}^{\alpha}$ can now be interpreted as a measure of the probability $P_1^{\alpha} = P(C_1 \mid \widetilde{\mathbf{I}}^{\alpha})$ that the latter belongs to class C_1. Correspondingly, the probability $P_2^{\alpha} = 1 - P_1^{\alpha}$ that the instance is of class C_2 can be related to $1 - O^{\alpha}$. Hence, while the limitation to a linear separation remains, a neuron with continuous output can provide some additional information about its confidence regarding the classification of a specific input. When the resulting output is close to $1/2$, the input vector must lie near to the region boundary $\widetilde{h}$. If, e.g., the class structure is similar to that of figure 2.3 c), the categorization of any input close to the class separation is not as definite as if it was further away from the boundary and the output was closer to 1 or 0.

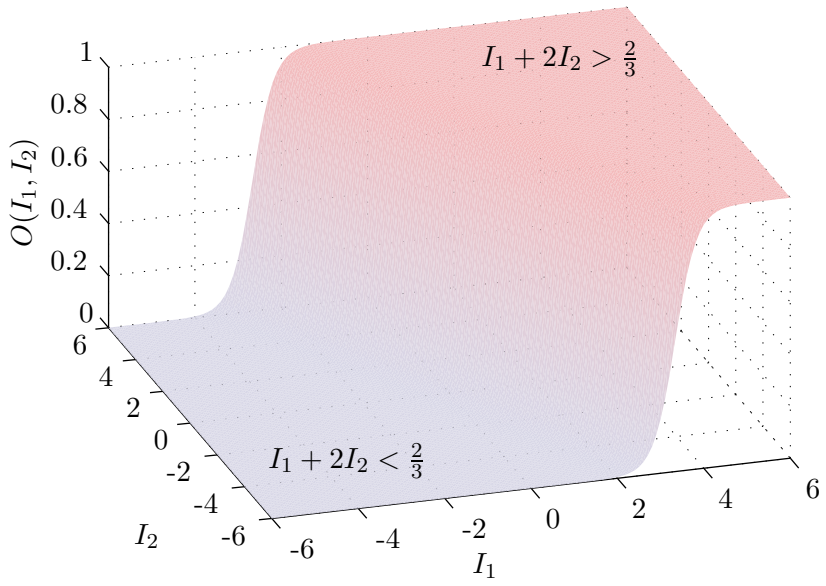


Figure 2.4: *If the output neuron of the simple perceptron in figure 2.2 a) is assigned the logistic transfer function φ_1, its output $O(I_1, I_2)$ becomes a continuous function of the inputs. The position and orientation of the linear separation between output 0 and 1 remain the same as in figure 2.2 b) but the transition becomes smooth.*

Gradient Descent Training: The Delta Rule

differentiable network output

Besides allowing for a more differentiated interpretation of the network output, a continuous neuron response as given by the logistic function φ_1 is also advantageous in so far as the neuron output function $O(\widetilde{\mathbf{w}}, \widetilde{\mathbf{I}})$ is now differentiable with respect to all input values I_i. Alternatively, given a fixed input vector $\widetilde{\mathbf{I}}^\alpha$, the response $O^\alpha(\widetilde{\mathbf{w}}) = O(\widetilde{\mathbf{w}}, \widetilde{\mathbf{I}}^\alpha)$ of the neuron can itself be regarded as a differentiable function $O^\alpha \colon \mathbb{R}^{n_{\text{in}}+1} \to \mathbb{O}$ of the synaptic weights w_i.

differentiable network error

This important insight gives rise to a new training approach. Revisiting the two-class problem of section 2.1.2, let the error E_α of the network with regard to an input pattern $\widetilde{\mathbf{I}}^\alpha \in \mathbb{E}$ be defined in terms of the actual network response $O(\widetilde{\mathbf{w}}, \widetilde{\mathbf{I}}^\alpha)$ and the respective target output $T^\alpha \in \{0, 1\}$ as follows

$$E_\alpha(\widetilde{\mathbf{w}}) = \frac{1}{2}\left(O(\widetilde{\mathbf{w}}, \widetilde{\mathbf{I}}^\alpha) - T^\alpha\right)^2. \tag{2.15}$$

Then, the overall performance of the network on the whole training set $\mathbb{E}$ can be measured by the error function

$$E(\widetilde{\mathbf{w}}) = \sum_{\mathbb{E}} E_\alpha(\widetilde{\mathbf{w}}) = \sum_{\mathbb{E}} \frac{1}{2}\left(O(\widetilde{\mathbf{w}}, \widetilde{\mathbf{I}}^\alpha) - T^\alpha\right)^2 \tag{2.16}$$

such that the smaller $E(\widetilde{\mathbf{w}})$, the better the choice of weights $\widetilde{\mathbf{w}}$ for this particular problem. The goal of training the network for a specific task could thus be identified with the minimization of $E(\widetilde{\mathbf{w}})$. As by definition, $E(\widetilde{\mathbf{w}})$ is a continuous and

differentiable function of all weights w_i, it seems appropriate to pursue this aim using common gradient descent optimization methods [20].

iterative gradient descent

Following an iterative approach similar to the one introduced in 2.1.2, the training starts once more with some random initial weights $\widetilde{\mathbf{w}}$ which are likely to yield an undesirably large value of $E(\widetilde{\mathbf{w}})$. Therefore, the weight vector is updated by moving into that direction of the weight space along which $E(\widetilde{\mathbf{w}})$ decreases most rapidly. This direction is given by the negative gradient $-\nabla_{\widetilde{w}} E$ and the weight change can be written as

$$\Delta\widetilde{\mathbf{w}} = -\eta \, \nabla_{\widetilde{w}} \, E\big|_{\widetilde{\mathbf{w}}} \tag{2.17}$$

including an adjustable learning rate η. In case of the perceptron learning algorithm, the weights are updated after the processing and evaluation of each single input pattern $\widetilde{\mathbf{I}}^{\alpha}$ and the same scheme can also be adopted here. If, furthermore, the modification of each weight w_i is regarded separately, one obtains

$$\Delta w_i^{\alpha} = -\eta \, \frac{\partial E_{\alpha}}{\partial w_i}\bigg|_{\widetilde{\mathbf{w}}} \qquad \forall \; 1 \leq i \leq n_{\text{in}}. \tag{2.18}$$

Combining equations 1.12 and 2.15, it is straight forward to compute the involved partial derivatives of E_{α} to be

$$\frac{\partial E_{\alpha}}{\partial w_i}\bigg|_{\widetilde{\mathbf{w}}} = \varphi'(net^{\alpha}) \; (O(\widetilde{\mathbf{w}}, \widetilde{\mathbf{I}}^{\alpha}) - T^{\alpha}) \; I_i^{\alpha} \qquad \forall \; 1 \leq i \leq n_{\text{in}}. \tag{2.19}$$

the delta rule

It turns out that apart from the input pattern and the corresponding actual and desired outputs, the weight change Δw_i^{α} depends also on the derivative of the activation function φ' at the point $net^{\alpha} = \widetilde{\mathbf{w}} \, \widetilde{\mathbf{I}}^{\alpha}$. It can finally be written in the form

$$\Delta w_i^{\alpha} = -\eta \, \delta^{\alpha} \, I_i^{\alpha} \qquad \forall \; 1 \leq i \leq n_{\text{in}}, \tag{2.20}$$

where δ^{α} is defined as

$$\delta^{\alpha} = \varphi'(net^{\alpha}) \; (O(\widetilde{\mathbf{w}}, \widetilde{\mathbf{I}}^{\alpha}) - T^{\alpha}) \qquad \forall \; 1 \leq i \leq n_{\text{in}}. \tag{2.21}$$

The above result was first formulated by Widrow and Hoff in 1960 [227] for the case of linear neurons ($\varphi'(net^{\alpha}) = 1$). Since then, this particular weight updating scheme and its variants are known by several names including Widrow-Hoff rule, delta rule, adaline rule or LMS (least mean square) rule.

Discussing the Delta Rule

The delta rule is designed to iteratively minimize the error $E(\widetilde{\mathbf{w}})$ of the network on the training patterns $\mathbb{E}$ and is therefore a perfect example for an error correction learning algorithm (section 1.2.4). Initially, the particular choice of the error function given by equation 2.16 is only motivated by analytical simplicity and alternative functions of the network outputs and target values are conceivable which can serve as adequate error measures as well. Using the specific form 2.16 can be motivated by some quite general assumptions concerning the underlying structure of the data [20].

linearly inseparable patterns

Unlike the perceptron learning approach, a training procedure based on the delta rule does not terminate automatically once a perfect set of weights is found. Rather, it asymptotically approaches the minimum of the error E. If, as is the case for most realistic applications, the training inputs are not linearly separable, it has been said that an exact solution $E = 0$ can generally not be found. In the case of linearly separable classes, however, it has already been suggested implicitly that the exact solution can be approximated as closely as required. More specifically, it can be achieved that $E < \epsilon$ for any given small but nonzero number $\epsilon > 0$, $\epsilon \in \mathbb{R}$.

As an example, consider once more a network with the structure and the weights as shown in figure 2.2 a) but having a logistic activation function φ_l with $\beta = 2$. It is obvious from figures 2.2 b) and 2.4 that its weights could be readjusted as to model the Boolean OR as accurately as desired. In fact, choosing $w_1 = w_2 = 4$ and $w_0 = 2$, one obtains an error E with regard to all possible four input vectors of less than $5{\times}10^{-4}$.

partial solutions

Even if the problem is not linearly separable, there might be partial solutions that correspond to minima of the error function with $E \neq 0$. Besides solving the OR problem, the above choice of the weights can be seen as one such partial solution for the Boolean XOR. Again, considering all four possible inputs, the network does in this case achieve $E \approx 1/2$: Only the input $(1, 1)$ results in a clearly incorrect output of approximately 1 instead of the desired 0.

asymptotic behavior

When an exact linear separation does not exist, the perceptron algorithm does not automatically yield a suitable near-optimal partitioning of the input space and is, on the contrary, likely to infinitely oscillate between several bad ones. Gradient based training approaches like the delta rule will under these circumstances at least converge to one of the existing partial solutions[1]. For real-world applications, the latter behavior is usually more desirable and in this respect, the gradient based training approach is clearly superior to the perceptron learning algorithm.

local minima

If, as in all hitherto discussed cases, the target values lie outside the actual range of the output function, the error $E(\widetilde{\mathbf{w}})$ has a high probability of exhibiting additional local minima besides those that represent reasonable partial solutions [88]. It is a major problem of all gradient descent approaches that they cannot reliably be prevented from getting stuck in a local minimum of this last type instead of converging to a set of weights $\widetilde{\mathbf{w}}$ that yields a perfect or at least the best possible performance.

The remaining error E of the network at the end of the learning procedure is an appropriate measure for the quality of the found solution. In particular, any problems due to unwanted local minima of the error function could be circumvented by training several networks and keeping only the one (or the ones) with the lowest error E, hoping that it might represent the actually achievable optimum. But even then, a low residual error does not guarantee the best possible solution.

choosing the learning rate η

Finally, the convergence behavior of the delta rule critically depends on the choice of the learning rate η. While the individual weight changes reasonably occur into the direction of the strongest decrease of the error function, they are discrete in nature and their size is proportional to the steepness of the error function at

[1] Assuming adequate choices for the learning rate.

the current point. The scaling factor is given by η and if both, the absolute value of the current gradient and the value of the learning rate are sufficiently large, the modification of a weight might in fact move the weight vector too far and beyond the actual minimum. Depending on the error surface, the initial weight vector and the precise value of η, this will lead to an oscillatory behavior which either considerably slows down the convergence towards the minimum or even causes the weights to reliably diverge from the optimal weight vector.

small learning rates

Too small a learning rate, on the other hand, naturally impedes a fast convergence as well. As the slope of the logistic activation function $\varphi_{\mathrm{l}}(x)$ is arbitrary close to 0 for any input arguments that are sufficiently far away from the switching point $x = 0$, the derivatives of the error function with respect to the weights tend to be relatively small within large domains of the weight space. Using a small learning rate η, it might take the algorithm many iterations to get out of these flat regions of the error surface. If the algorithm is only allowed a fixed number of training steps before it is terminated — as is common in most realistic setups — the impact of these plateaus on the training performance is equivalent to the effects caused by local minima.

In practice, the delta rule is known to approach the optimal weight vector measurably faster than the perceptron learning algorithm but choosing the optimal learning rate often remains a matter of trial and error.

2.1.4 Generalization to Multiple Outputs

multi-class problems

A network with only one output neuron cannot satisfactorily be applied to classification tasks with more than two classes and in such cases, the network is therefore required to exhibit multiple outputs. It is common to allocate one output neuron of the network to each individual class of the currently investigated problem. In this setup, the desired response of the network after the application of an input vector $\widetilde{\mathbf{I}}^{\alpha}$ is to distinctly activate the specific output neuron that is associated with the respective correct class C_{α}.

Consider a classification problem with N_{out} classes and let the network exhibit N_{out} output neurons that are labeled such that the output with index k is associated with class C_k. If the input pattern $\widetilde{\mathbf{I}}^{\alpha}$ belongs to class C_{α}, the target value T_k^{α} for the kth neuron is reasonably set to 1 if $k = \alpha$ and to 0 otherwise.

Thus, regarding the coding of the network response and the appropriate assigning of target values, the transition to networks with multiple outputs is straight forward. However, as each of the individual output neurons of a one-layer perceptron continues to perform just one single linear separation of the input, it remains to be analyzed what kind of partitionings of the input set a network with multiple outputs is able to perform.

Representational Capability

Figure 2.5 a) schematically visualizes the input space of an exemplary network with two input nodes and three output neurons. The weights of the network shall be chosen such that the neurons with indices $k = 1, 2, 3$ perform the shown

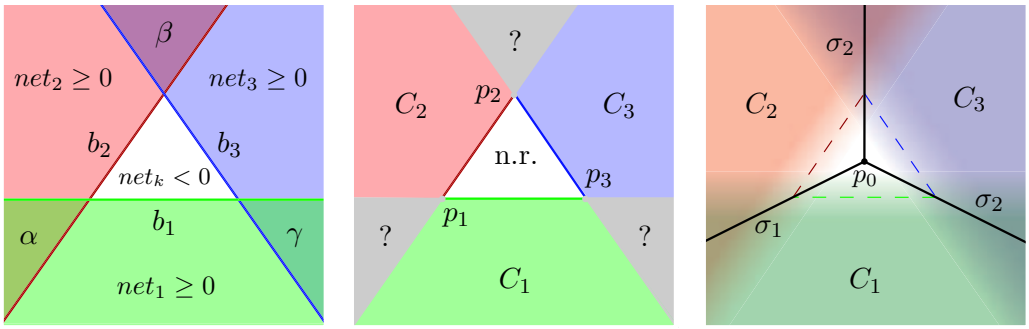
corresponding linear separations b_1, b_2 and b_3. The colored areas mark those regions of the plane for which the total inputs net_k of the respective neurons exceed 0. These regions partly overlap, i.e., input vectors from the areas α, β and γ give rise to a positive total input for two neurons at a time. On the other hand, points from the white triangle in the center do not result in positive input to any of the three neurons.

binary output

Using threshold activation functions, the network output in reply to an input $\widetilde{\mathbf{I}}$ can only be unambiguously interpreted if not more than one output neuron responds with 1. This is illustrated in figure 2.5 b): All points of the former regions α, β and γ cause two output neurons to be activated simultaneously and cannot be clearly classified. In accordance to the definition of the theta function (equation 1.3), these areas include their respective boundaries, particularly, the points p_1, p_2 and p_3 which are given by the intersections of the original linear separations b_1, b_2 and b_3.

ambiguous responses

While all inputs originating from the white area in the center result in no response at all, the parts marked with C_1, C_2 and C_3 yield an unambiguous output of the network. Any input vectors from these regions including their respective boundaries to the central area lead to the activation of exactly one output neuron which can be regarded as the definite classification prediction of the network. As figure 2.5 b) suggests, a large fraction of the input space is lost to points that cannot be classified at all. This is likely to cause problems if the different classes lie close to each other in the input space, in which case the response of the network is bound to be useless for a large number of instances.



a) b) c)

Figure 2.5: ***a)*** *Three neurons with two input nodes perform three linear separations b_1, b_2 and b_3 of the input plane. The colored areas denote those regions, where the respective total input net of the neurons exceeds 0. While the areas α, β and γ deliver positive input to two neurons simultaneously, no neuron receives a positive input from the white central region.* ***b)*** *Using a threshold activation function, the regions formerly denoted as α, β and γ result in an ambiguous output. The central region yields no response at all and only the areas marked with C_1, C_2 and C_3 can unambiguously be associated with their respective classes.* ***c)*** *A continuous output of the neurons allows a for a better separation of the input plane: Only the edges σ_1, σ_2 and σ_3 as well as the point p_0 lead to ambiguity. The hereby performed partitioning is slightly different from the one in b) and in particular, all points in the central region cause a nonzero network response.*

continuous output

If the neurons incorporate continuous activation functions, it is sufficient for the respective correct neuron to yield the highest response among all outputs. The remaining neurons are not required to exhibit an activation close to 0 as long as the one with the highest output is regarded as the classification response of the network. This leads to a partitioning of the input plane as illustrated in figure 2.5 c) which is clearly distinct from the one in figure 2.5 b). As before, the areas marked with C_1, C_2 and C_3 represent those parts of the input space that can unambiguously be allocated to the corresponding classes. But the separation between two regions C_i and C_k is now defined by the points that cause the respective outputs O_i, O_k to return the same value $O_i(\widetilde{\mathbf{I}}) = O_k(\widetilde{\mathbf{I}})$. The hereby defined boundaries turn out to be straight lines (σ_1, σ_2 and σ_3) that meet in the point p_0 which is unique in resulting in an equal response of all outputs.

improved response

It is to be emphasized that this new partitioning is different from the original separation performed by b_1, b_2 and b_3. Most notably, there are no points left in the input space that yield a network response of exactly 0. Even inputs from the central part (except p_0 and those on σ_1, σ_2 and σ_3) can each be assigned to one of the classes. In practice, it is common to define a threshold value that has to be exceeded by a neuron output in order to be accepted as a trustworthy classification. Fixing a threshold value of 1/2 would again declare the points of the central triangle (framed by the dashed lines) as not belonging to any class.

convex decision regions

A comparison between figures 2.5 b) and 2.5 c) reveals that a differentiated neuron output notably decreases the amount of unclassifiable inputs but it is also evident that in both cases, the pairwise separations between the different categories persist to be linear. Moreover, it can be anticipated from both figures and can also be proven mathematically that for monotonic activation functions, the individual classification regions are always connected and convex. That is, for two points $\widetilde{\mathbf{I}}_a^\alpha$ and $\widetilde{\mathbf{I}}_b^\alpha$ being assigned to class C_α, all points $\widetilde{\mathbf{I}}_\mu$ that lie on the straight connection between them ($\widetilde{\mathbf{I}}_\mu = \widetilde{\mathbf{I}}_a^\alpha + \mu(\widetilde{\mathbf{I}}_b^\alpha - \widetilde{\mathbf{I}}_a^\alpha)$, $0 < \mu \leq 1$) are inevitably assigned to class C_k, too [20].

Going back to the classification problem depicted in figure 2.3 b), it has to be concluded as a consequence of the above result that this specific task cannot accurately be solved by a single-layer network even if it has one output neuron reserved for each class: The instances of the upper class in figure 2.3 b) form a region that is not convex but concave.

motivation for multiple layers

The limitation to connected and convex classification regions as well as the frequent ambiguities caused by the undifferentiated output of threshold neurons can be overcome by assigning multiple neurons to each class C_k. If the outputs of these neurons are appropriately combined, they can perform a potentially more differentiated and flexible decision concerning the allocation of a given input $\widetilde{\mathbf{I}}$ to the class C_k. This is in fact the main motivation to build feedforward networks with more than one layer which will be the topic of section 2.2. The subject of single-layer networks with multiple outputs shall be closed by discussing the necessary modifications to the training approaches.

Training Single-Layer Networks with Multiple Outputs

multiple outputs are independent

Regarding the formulation of learning algorithms, the generalization to single-layer networks with multiple outputs poses no further problem. This is due to the fact that the operation of each single output neuron is in no way influenced by the response of the others. In fact, a single-layer perceptron with N_{out} outputs is fully equivalent to N_{out} separate networks with one layer and one output each, as long as the kth network is trained using only the respective kth components T_k^α of the target vectors $\mathbf{T}^\alpha$.

perceptron learning

Let the weights of the network be labeled such that the weight w_{ik} is associated with the connection leading from the ith input node $0 \leq i \leq N_{\text{in}}$ to the output neuron with index k, $1 \leq k \leq N_{\text{out}}$. The weights of the network then form an $(N_{\text{in}} \times N_{\text{out}})$-matrix with its rows being the extended weight vectors $\widetilde{\mathbf{w}}_k$ of the single output neurons. In this notation, the perceptron learning rule 2.14 can readily be generalized to multiple outputs by setting

$$\Delta w_{ik}^\alpha = \frac{1}{2}\eta\,(T_k^\alpha - O_k^\alpha)\,\widetilde{I}_i^\alpha. \tag{2.22}$$

gradient descent

Correspondingly, if net_k denotes the total input of the kth output neuron, the delta rule defined by equations 2.20 and 2.21 immediately becomes

$$\Delta w_{ik}^\alpha = -\eta\,\delta_k^\alpha\,\widetilde{I}_i^\alpha \qquad \forall\ 1 \leq i \leq N_{\text{in}},\ 1 \leq k \leq N_{\text{out}} \tag{2.23}$$

where δ_k^α is now defined as

$$\delta_k^\alpha = \varphi'(net_k^\alpha)\ (O_k(\widetilde{\mathbf{w}}_k, \widetilde{\mathbf{I}}^\alpha) - T_k^\alpha) \qquad \forall\ 1 \leq i \leq N_{\text{in}},\ 1 \leq k \leq N_{\text{out}}. \tag{2.24}$$

2.2 Multi-Layer Feedforward Networks

Single-layer feedforward neural networks suffer from several limitations concerning the range of functions that they can represent and it has already been indicated that these restrictions can be overcome by networks with multiple layers. The following sections address strictly layered networks without shortcut connections as already introduced in section 1.2.2 and illustrated by figure 1.6 b).

In addition to the input signals I_i and the outputs O_k, let the return value of the hidden neuron with index h in layer l be denoted with $H_h^{(l)}$, $1 \leq h \leq N^{(l)}$, where $N^{(l)}$ is the number of units in this layer. Furthermore, let the weight $w_{ih}^{(l)}$ be associated with the synaptic connection leading from the ith node in layer $l-1$ to the unit h in layer l. For $l = 1$, the sources of these connections are the input nodes of the network I_i while for $l \geq 2$, they are the units of some hidden layer $l-1$.

implementing the bias values

The bias values $b_h^{(l)}$ of neurons in any layer $l \geq 2$ can be implemented by connections $w_{0h}^{(l)} = b_h^{(l)}$ originating from an additional hidden unit $H_0^{(l-1)}$ in the preceeding layer whose output is constantly set to -1. This is similar to the implementation of bias values for neurons in the first layer via an additional virtual network input I_0. The bias values of hidden units in higher layers could also be realized by directly

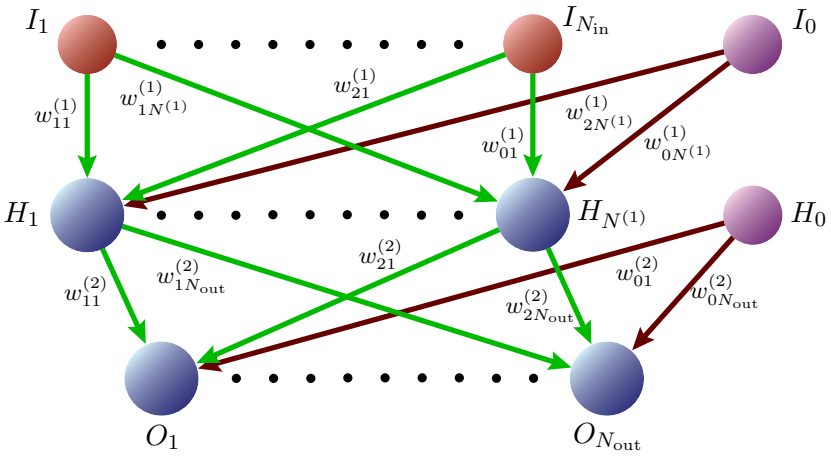


Figure 2.6: *An exemplary network with N_{in} input nodes, N_{out} output neurons and $N^{(1)}$ units in a single hidden layer. The weight $w_{ij}^{(l)}$ connects the ith node in layer $l-1$ with the jth neuron in layer l. The internal bias of the neurons is implemented by additional connections originating from virtual nodes (I_0 and $H_0^{(1)}$) in the respective preceeding layer.*

connecting these neurons to I_0, but this would not conform to the desired strictly layered structure. Regardless of how the additional input signals are implemented, the dimensionality of the input space $N_{\text{in}}^{(l)}$ for any unit in layer $l \geq 2$ — and thus the length of its extended weight vector $\widetilde{\mathbf{w}}_h^{(l)}$ — is given by the number of nodes in the preceeding layer $N^{(l-1)}$ incremented by 1.

an exemplary two-layer network

As an example, consider the network shown in figure 2.6 which exhibits N_{in} inputs, $N_{\text{out}} = N^{(2)}$ outputs and $N^{(1)}$ neurons in its single hidden layer. Adopting the above notation, the output function of this network can be written as

$$O_k(\widetilde{\mathbf{I}}) = \widetilde{\varphi}\left(\sum_{h=1}^{N^{(1)}} w_{kh}^{(2)}\,\varphi\left(\sum_{i=0}^{N_{\text{in}}} w_{hi}^{(1)} I_i\right) + w_{k0}^{(2)}\,H_0^{(1)}\right) \quad \forall\, 1 \leq k \leq N_{\text{out}}, H_0^{(1)} = I_0 = -1, \tag{2.25}$$

where φ and $\widetilde{\varphi}$ are the respective activation functions of the hidden and the output neurons. Note that these functions are not required to be the same and for analytical investigations, it is quite common to use hidden units with squashing activation functions but to choose a linear response for the output neurons.

2.2.1 Computational Capabilities

Boolean functions

It has already been stated earlier that a two-layer feedforward network of the type of neuron originally introduced by McCulloch and Pitts can compute any arbitrary Boolean function $f_{\text{B}}\colon \{0,1\}^n \to \{0,1\}$. This equally applies to networks of weighted threshold neurons. Figure 2.7 shows an example that, provided its inputs are constrained to assume binary values, effectively computes the Boolean XOR. Networks of two layers are in particular not restricted to linearly separable problems.

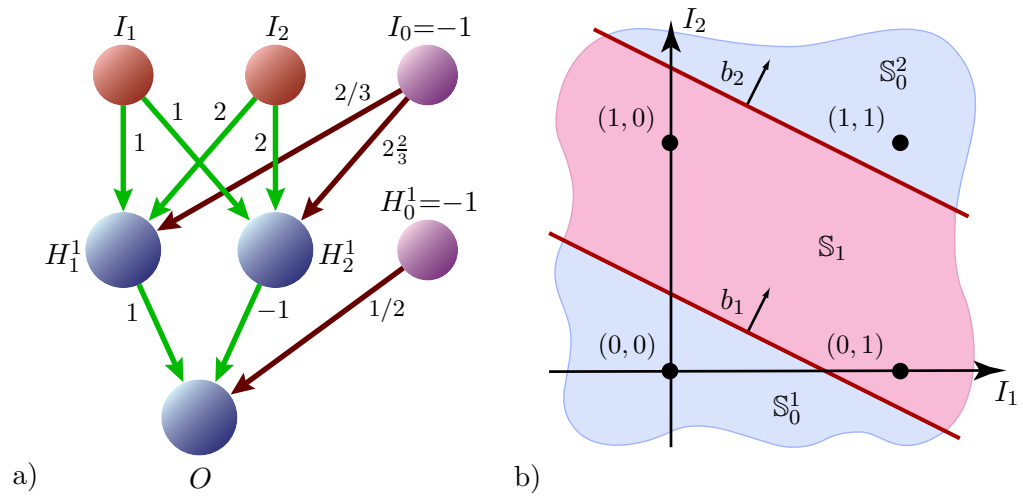


Figure 2.7: *The two hidden neurons H_1^1 and H_2^1 of the simple two-layer network shown in* ***a)*** *create the respective linear separations b_1 and b_2 as depicted in* ***b)****. The small black arrows mark those sides of the boundaries that cause the corresponding neuron to exhibit an output of 1. Hidden unit H_1^1 actually corresponds to the simple perceptron shown in figure 2.2 a). Its output is 0 for all points in region $\mathbb{S}_0^1$ but assumes 1 on the domains $\mathbb{S}_1$ and $\mathbb{S}_0^2$ while the bias of neuron H_2^1 has been modified to let it responds with 1 only for input vectors in $\mathbb{S}_0^2$. The output neuron O combines the states of both hidden units according to (H_1^1 AND (NOT H_2^1)) resulting in a value of 1 only for the points of $\mathbb{S}_1$ (red). When being applied binary input values, the network readily computes the Boolean XOR.*

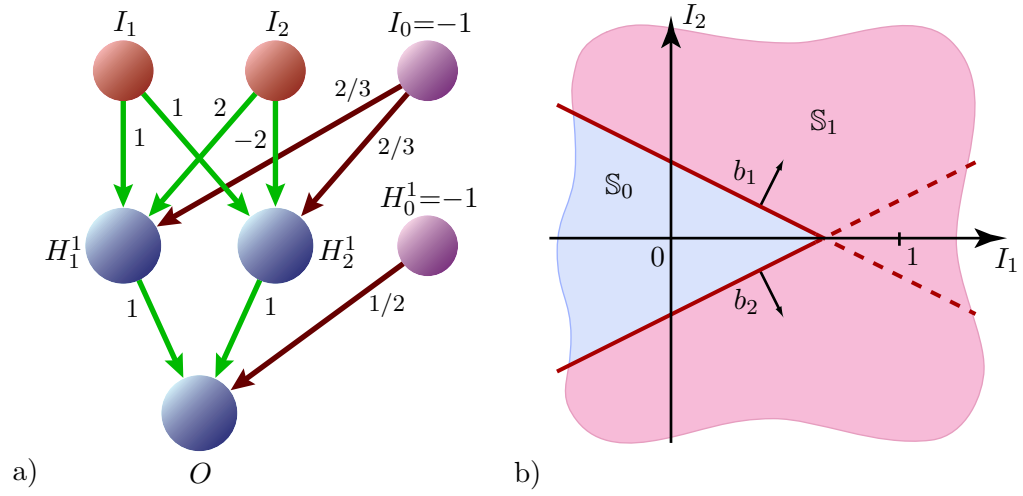


Figure 2.8: *The two-layer network shown in* ***a)*** *gives rise to a partitioning of the input plane that is illustrated in* ***b)****. As in figure 2.7, the linear separations defined by the hidden units H_1^1 and H_2^1 are given by the boundaries b_1 and b_2 and once again, H_1^1 performs the same computation as the simple perceptron in figure 2.2 a). Neuron H_2^1 differs from H_1^1 only in the sign of the weight $w_{22}^{(1)}$. The output neuron O gets activated once at least one of the hidden units assumes an output value of 1 (H_1^1 OR H_2^1), thus, the network response is 0 only for points in $\mathbb{S}_0$ (blue) and is 1 for all inputs in $\mathbb{S}_1$ (red). The resulting partitioning is clearly not convex.*

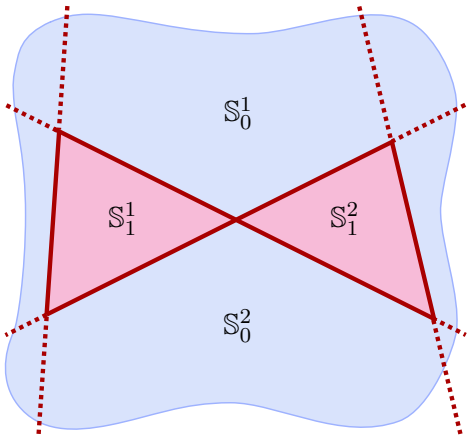


Figure 2.9: *If the network output is required to be 0 for any points of the regions $\mathbb{S}_0^1$ and $\mathbb{S}_0^2$ but to assume a value of 1 on the domains $\mathbb{S}_1^1$ and $\mathbb{S}_1^2$, this peculiar separation cannot be achieved by a network of threshold neurons having only one hidden layer [23].*

concave decision regions

The investigations in section 2.1.4 revealed the limitation of single-layer networks to convex decision regions. There is no such general restriction for two-layer networks as can be inferred from figure 2.8 which presents a simple example for a network with a non-convex classification domain. This observation could give rise to the impression that two-layer networks of threshold neurons can implement any desired decision boundary which is, in fact, not the case. The separation given in figure 2.9 cannot be realized by a network with two layers [23]. But it turns out that to create arbitrary, disjoint and non-convex classification regions, three layers of threshold neurons are generally sufficient [127].

finite classification problems

Moreover, it can also be proven that under relatively mild conditions[2], a network with one output and $\lceil n/d \rceil$ neurons in a single hidden layer can correctly separate n points in d dimensions into two arbitrarily defined classes[3] [11]. In other words, any classification problem with two classes that is defined by a finite set $\mathbb{E}$ of instances can completely be solved by a two-layer network of McCulloch-Pitts neurons as long as the number $N^{(1)}$ of its hidden units is large enough.

function approximation

If the output neuron O is allowed to exhibit a linear activation function, it has been shown by various authors that a network with one hidden layer and a sufficiently large number of internal neurons can approximate any continuous function $f : \mathbb{S} \rightarrow \mathbb{R}$ on a compact subset $\mathbb{S} \in \mathbb{R}^{N_{\text{in}}}$ to arbitrary accuracy [23] [100]. More precisely, the network can be constructed such that the associated network function F obeys

$$\epsilon > \mathcal{E}^f = || F - f ||_\infty^{\mathbb{S}} \tag{2.26}$$

for any small but nonzero number $\epsilon \in \mathbb{R}$, $\epsilon > 0$ and $|| \cdot ||_\infty^{\mathbb{S}}$ denoting the uniform norm on $\mathbb{S}$

$$|| f(x) ||_\infty^{\mathbb{S}} = \sup_{x \in \mathbb{S}} | f(x) | . \tag{2.27}$$

For the propositions of this theorem to hold, the activation function of the hidden units can be any continuous or discrete squashing function (see section 1.2.3). In this sense, two-layer neural networks can be regarded as universal function

[2] The n points are required to be in general position, i.e., any subset of d or fewer vectors has to be linearly independent.

[3] $\lceil n/d \rceil$ denotes the smallest integer value greater than or equal to n/d.

approximators and therefore form an important class of networks for practical applications.

sigmoidal units

Note that the value set V of any continuous real function f on a compact set $\mathbb{S} \in \mathbb{R}^{N_{\text{in}}}$ is itself compact [59]. Through shifting and rescaling, f can be modified to yield a corresponding function f' with a value set V' that obeys $V' \subset [0, 1]$. On the other hand, for $x \approx 0$ and/or $\beta \gg 1$, the logistic function $\varphi_{\text{l}}(x)$ becomes approximately linear. The above result can thus easily be transferred to networks that consist entirely of sigmoidal units.

open questions

It has to be noted that most proofs for the universal approximation potential of two-layer networks are neither concerned about the number of hidden units that is necessary to achieve a desired accuracy nor about whether networks with more layers might be able to obtain the same result with a smaller number of neurons and synaptic connections. Also, they do not provide an answer to the question of how the network can actually be trained to perform the desired approximation. Nevertheless, these issues are undoubtedly of critical relevance for practical applications.

network sizing

trainability

Regarding the residual error $\mathcal{E}_N^f$ that remains when a network with N hidden units in a single hidden layer optimally approximates a given function f, the former can be shown to decrease as $\mathcal{O}(1/\sqrt{N})$ when N grows [113]. Assuming the existence of an efficient training algorithm that is capable of optimizing a network with N hidden units to yield the minimum achievable error E (equation 2.16) on a finite training set $\mathbb{E}_{\text{t}}$, it can be estimated how N has to grow with the number of training inputs $N_e = \mid \mathbb{E}_{\text{t}} \mid$ in order to let the residual error $\mathcal{E}_N^f$ approximate 0 [219]. It is to be emphasized that in contrast to the error E of the network on the training set, the value $\mathcal{E}_N^f$ refers to the remaining difference[4] between the network output F and the original function f on their whole domain $\mathbb{S}$ (see equation 2.27). Thus, feedforward networks can not only approximate any continuous function to arbitrary accuracy but can in principle also be taught to do so using a limited number of training examples which is a promising observation especially with regard to their generalization abilities.

Admittedly, this last theorem assumes two vital conditions that are both not trivially fulfilled. First, it requires the network size to be adjusted appropriately and second, it depends on the existence of the aforementioned learning algorithm that is able to reliably yield the global minimum of the network error E on the training set. It has already been shown for single-layer networks that the formulation of such a training algorithm is at best difficult and the following sections shall therefore address the important issue of training feedforward neural networks with more than one layer.

2.2.2 Training Multi-Layer Networks

the internal feature space

Each output neuron of a multi-layer feedforward network with L layers can be considered as a simple perceptron that receives its input signals from the $N^{(L-1)}$ units of the last hidden layer and as such, it can only perform a linear partitioning of its input space $\Psi \subseteq \mathbb{R}^{N^{(L-1)}}$. Seen from this angle, it is the purpose of the

[4] according to the L^2 norm [60]

earlier layers to map the original input vectors $\widetilde{\mathbf{I}}^{\alpha} \in \Phi \subseteq \mathbb{R}^{N_{\text{in}}}$ onto corresponding internal representations $\widetilde{\mathbf{R}}^{\alpha} \in \Psi$ in such a way that within the feature space Ψ, the desired classification can be performed by simple linear separations. Following this view, the operation of the network is divided into two distinct steps: First, the nonlinear mapping of a given input onto its internal representation and second, the classification of this pattern according to a linear separation of the intermediate feature space.

The Credit Assignment Problem

training the hidden nodes

If it is ensured that the preceeding layers provide an adequate internal representation in which a given set of training examples $(\widetilde{\mathbf{I}}^{\alpha}, \mathbf{T}^{\alpha})$ is linearly separable (or at least approximately so), the weights of the last layer could be trained using either the perceptron algorithm or the gradient based approach introduced in 2.1.2 and 2.1.3. Unfortunately, training the hidden neurons to perform the required preprocessing cannot be achieved via the same learning algorithms. While they can equally be regarded as being like simple perceptrons, there are no target values available for the hidden units that could be assigned to their outputs. Thus, if the network produces an incorrect output in response to a given input $\widetilde{\mathbf{I}}^{\alpha}$, there is, initially, no way of determining which group of hidden units is responsible for this error. Consequently, there is no apparent procedure for calculating appropriate changes of the synaptic weights. This is a fundamental issue that is commonly denoted as the credit assignment problem.

Once the operation of the system is divided into the two distinct stages described in the preceeding paragraphs, it is not mandatory to let the first step be performed by a feedforward network. Various alternative approaches have been proposed that circumvent the credit assignment problem by using different types of networks for the implementation of the initial nonlinear mapping to the feature space. These approaches will briefly be introduced in section 2.3. Regarding multilayer feedforward networks, there does in fact exist a relatively simple solution to this problem if the output of the neurons is continuous.

Error Backpropagation

differentiable network function

As long as the activation functions of all neurons are differentiable with respect to the synaptic weights, it is evident by virtue of the chain rule [61] [59] that the same also applies to the output of a complete feedforward network regardless of the number of layers. For a network with one hidden layer, this can directly be inferred from equation 2.25. Therefore, the sum-of-squares error of the network on a given set of training examples as defined by equation 2.16 becomes a differentiable function of the weights as well.

Similar to the case of single-layer networks, the derivatives with respect to the individual synaptic weights $w_{ij}^{(l)}$ can be used to minimize the network error function E by employing appropriate methods like gradient descent. The procedure by which the necessary derivatives can be evaluated for feedforward networks with multiple layers is known as error backpropagation or simply backpropagation [175].

differentiable network error

In analogy to the iterative approaches described earlier, consider the network error E_α obtained on a single training example $(\widetilde{\mathbf{I}}^\alpha, \mathbf{T}^\alpha)$ as given by equation 2.15 and let the network incorporate L layers of neurons. Using the common chain rule, the derivative of E_α with respect to any synaptic weight $w_{ij}^{(l)}$ within the network can be calculated as

$$\frac{\partial E_\alpha}{\partial w_{jk}^{(l)}} = \frac{\partial E_\alpha}{\partial\, net_k^{(l)}} \cdot \frac{\partial\, net_k^{(l)}}{\partial w_{jk}^{(l)}} \tag{2.28}$$

for all index values $1 \leq l \leq L$, $1 \leq k \leq N^{(l)}$ and $1 \leq j \leq N^{(l-1)}$ with $N^{(0)} = N_{\text{in}}$ and $N^{(L)} = N_{\text{out}}$. Here, $net_k^{(l)}$ stands for the total input to the kth neuron of the lth layer in the presence of the network input $\widetilde{\mathbf{I}}^\alpha$. It is convenient to denote the first term on the right hand side of 2.28 as $\delta_k^{(l)}$, i.e.,

$$\delta_k^{(l)} := \frac{\partial E_\alpha}{\partial\, net_k^{(l)}}. \tag{2.29}$$

In pursuit of a simple notation, it is understood that the regarded error E is calculated on the basis of only a single training pattern and the index α shall in the following be omitted. The second term in equation 2.28 can then readily be written as

$$\frac{\partial\, net_k^{(l)}}{\partial w_{jk}^{(l)}} = H_j^{(l-1)}\Big(\widetilde{\mathbf{w}}_j^{(l-1)}, \widetilde{\mathbf{I}}\Big) = H_j^{(l-1)} \tag{2.30}$$

and the substitution of 2.29 and 2.30 into 2.28 yields the general result

$$\frac{\partial E}{\partial w_{jk}^{(l)}} = \delta_k^{(l)} \cdot H_j^{(L-1)}. \tag{2.31}$$

output neurons

For synaptic connections that lead to one of the output neurons in the last layer L, the resulting $\delta_k^{(L)}$'s turn out to be similar to the δ_k's that are obtained in the case of a single-layer network

$$\begin{aligned} \delta_k^{(L)} &= \varphi'(net_k^{(L)}) \ \Big(O_k\Big(\widetilde{\mathbf{w}}_k^{(L)}, \widetilde{\mathbf{I}}\Big) - T_k\Big) \qquad \forall\, 1 \leq k \leq N_{\text{out}} = N^{(L)} \\ &= \varphi'(net_k^{(L)}) \ (O_k - T_k)\,. \end{aligned} \tag{2.32}$$

Therefore, the respective derivatives of the error function E equal those used for the delta rule (section 2.1.3) with the network input I_j being replaced by the output of the hidden unit $H_j^{(L-1)}$

$$\frac{\partial E}{\partial w_{jk}^{(L)}} = \varphi'(net_k^{(L)}) \ (O_k - T_k) \ H_j^{(L-1)}. \tag{2.33}$$

hidden neurons

In the case of a connection that does not lead to a network output but rather to the jth neuron of some hidden layer $1 \leq l < L$, the chain rule has to be applied once more. Thereby, equation 2.29 is reformulated to become

$$\delta_j^{(l)} = \frac{\partial E}{\partial\, net_j^{(l)}} = \sum_{k=1}^{N^{(l+1)}} \frac{\partial E}{\partial\, net_k^{(l+1)}} \cdot \frac{\partial\, net_k^{(l+1)}}{\partial\, net_j^{(l)}} \qquad \forall\, 1 \leq j \leq N^{(l)} \tag{2.34}$$

where it has been used that any change in the total input of the considered neuron does only affect the error E by effectively changing the total inputs of the other neurons that it is connected to.

backpropagation of the error

The first factor in each addend of the above sum is in fact the $\delta_k^{(l+1)}$ of the corresponding receiving neuron in the next layer. The calculation of the second factor is straight forward and if the activation functions of all neurons in the network are assumed to be the same, equation 2.34 can be rewritten as

$$\delta_j^{(l)} = \varphi'(net_j^{(l)}) \sum_{k=1}^{N^{(l+1)}} w_{jk}^{(l+1)}\, \delta_k^{(l+1)}. \tag{2.35}$$

In other words, the value of $\delta_j^{(l)}$ for a specific hidden unit can be obtained by propagating the δ's of neurons in higher layers backwards through the network. For the output neurons, the values $\delta_k^{(L)}$ are explicitly given by equation 2.33. Therefore, all values $\delta_j^{(l)}$ and correspondingly all possible partial derivatives of the error function with respect to arbitrary synaptic weights can be calculated by recursively applying equation 2.35.

Note that although the considerations so far assumed a network with a strictly layered structure where the activation functions of all neurons are the same, the above result can easily be generalized to any feedforward architecture and/or networks that incorporate different activation functions for different neurons [20].

advantages of backpropagation

Calculating the partial derivatives of the error function via backpropagation has two main advantages that are worth being pointed out explicitly. First, the backpropagation scheme is local in the sense that calculating $\delta_j^{(l)}$ for a specific connection only requires the knowledge of quantities that are available at the two end points of this synaptic link. Hence, all connections within one layer can be evaluated independently which allows for a potentially fast parallel implementation. Second, computing the partial derivatives of the error function 2.15 directly would in the presence of W weights take $\mathcal{O}(W^2)$ operations. Using equations 2.31, 2.32 and 2.35, all derivatives can be calculated in only $\mathcal{O}(W)$ operations.

Training with Backpropagation

error backpropagation and gradient descent

On the basis of the network error E after the application of an input pattern $\widetilde{\mathbf{I}}$ and equipped with its partial derivatives with respect to each weight $w_{ij}^{(l)}$, the gradient descent training strategy introduced in section 2.1.3 can be applied to multi-layer feedforward networks according to

$$\Delta w_{ik}^{(l)} = -\eta\, \delta_k^{(l)}\, H_i^{(l-1)}\left(\widetilde{\mathbf{I}}, \widetilde{\mathbf{w}}_i^{(l-1)}\right) \qquad \forall\, 1 \leq i \leq N^{(l-1)},\, 1 \leq k \leq N^{(L)} \tag{2.36}$$

where the values $\delta_k^{(l)}$ are obtained via error backpropagation.

Often, the term backpropagation refers to this specific training approach that combines the calculation of the required partial derivatives using the rules derived in the preceeding paragraphs with the gradient descent learning rule discussed in section 2.1.3. This convention shall be adopted in the following. Initially, the two parts are independent and the error-backpropagation scheme can also be combined with other optimization strategies that rely on the derivatives of the error function [20] [88].

Various modifications are conceivable to the backpropagation training strategy that potentially benefit its ability to yield the minimum possible error of the network on a given set of training examples. For example, it is straight forward to replace the sum-of-squares error function by alternative differentiable error measures.

learning with momentum

Furthermore, it has already been discussed in section 2.1.3 that the choice of learning rate η has a strong influence on the convergence behavior and speed of gradient descent. To avoid divergent oscillations, η is preferably kept small which in turn slows down the speed of the training. One way of dealing with this problem is the addition of a so-called momentum term in equation 2.36 that introduces a contribution from the preceeding learning step to each new weight change in the form of

$$\Delta w_{ik}^{(l)}(t+1) = -\eta\, \delta_k^{(l)}\, H_i^{(l-1)} + \nu \Delta w_{ik}^{(l)}(t) \tag{2.37}$$

where t refers to the number of training iterations [176]. The momentum parameter ν is bound to lie in $[0, 1]$ and is commonly chosen to be 0.9. It turns out that in the presence of a momentum term, the learning rate can safely be set to larger values without running the risk of provoking divergent oscillations [218].

learning rate adaption

Another approach to avoid the difficulties in defining appropriate learning parameters η and ν is to let them be adjusted automatically during the training process [109] [32].

local minima

While all these modifications promote a more reliable convergence of the training algorithm towards a minimum of the error function, they provide no guarantee of the latter being the global minimum instead of a possible local one. For a multilayer network, the error surface as a function of the individual weight values $w_{ik}^{(l)}$ can be considerably complex and compared to single-layer networks, the number of local minima that correspond to suboptimal solutions of a given learning problem is most likely to be increased even further. Regarding the difficulties that arise due to the existence of these local minima, the backpropagation algorithm and its different variants inherit all the problems that already affect the delta rule.

In spite of this, multilayer feedforward networks in combination with error backpropagation training algorithms constituted the working-horse of neural network research for many years and have successfully been employed in various applications ever since [123] [187].

2.3 A Short Overview of Alternative Network Models

While the remainder of this thesis primarily addresses implementations and applications of the feedforward networks discussed so far, the following sections shall give a brief summary of some important other types. These network models also prove to be of considerable value not only for scientific research but also for realistic pattern recognition applications. In so far, they can be seen as competing approaches to ordinary feedforward networks.

An exhaustive discussion of the various known types of artificial neural networks that would do justice to their interesting properties, capabilities and applications exceeds the scope of this thesis. The succeeding sections will examine alternative approaches only to an extent that is sufficient to allow for a reasonable comparison with classical feedforward networks and the training strategies presented in later chapters. Some important neural network models like Boltzmann machines [2] and self-organizing maps [120] will not be covered. For a detailed investigation of these topics and a more exhaustive study of artificial neural network theory in general, the interested reader is referred to the available literature [20] [84] [88] [169].

2.3.1 The Feature Space Revisited: Support Vector Machines

Returning to the concept of the intermediate feature space Ψ brought forward in section 2.2.2, it has been said before that the corresponding nonlinear mapping $r\colon \Phi \to \Psi$ of the original input vectors $\mathbf{I} \in \Phi$ does not necessarily have to be performed by a layered feedforward network of the hitherto discussed type.

radial basis function networks

A single hidden layer of N localized radial basis function neurons (see section 1.2.3) implements a mapping $\mathbf{r}\colon \Phi \to \mathbb{R}^N$ that is essentially different from those performed by ordinary neurons. Based on this mapping, a succeeding layer of simple binary or continuous-valued neurons can perform complex separations of the original input space. For some applications, this strategy proves to be an advantageous alternative to the partitionings provided by common feedforward networks [146]. Systems of this kind are referred to as radial basis function networks. Both, classical two-layer feedforward networks and radial basis function networks can be summarized under the generalized framework of support vector machines that attracted increasing interest during the last years [29] [40] [84] [112].

support vector machines

The methodology of support vector machines includes an efficient procedure for determining the optimal hyperplane that separates two classes of points $\mathbf{r}_\alpha$ in some space Ψ. The considered points can (but do not have to) be the nonlinear projections $\mathbf{r}(\mathbf{I}_\alpha) \in \Psi$ of some original vectors $\mathbf{I}_\alpha \in \Phi$. If the two sets are linearly separable in Ψ, the found hyperplane is optimal in the sense that it retains the highest possible normal distance from all vectors $\mathbf{r}_\alpha$. For linearly inseparable classes, it at least yields the minimum achievable classification error.

This optimal hyperplane is found by minimizing a Lagrangian that contains the vectors $\mathbf{r}_\alpha$ only in the form of inner products $\mathbf{r}_\alpha \cdot \mathbf{r}_\beta = \mathbf{r}(\mathbf{I}_\alpha) \cdot \mathbf{r}(\mathbf{I}_\beta)$. With regard to the original input vectors $\mathbf{I}_\alpha$, this important feature allows to reformulate the inner product in Ψ as a symmetric Kernel function

$$K(\mathbf{I}_\alpha, \mathbf{I}_\beta) = \mathbf{r}(\mathbf{I}_\alpha) \cdot \mathbf{r}(\mathbf{I}_\beta) \tag{2.38}$$

on Φ. The task of separating the two classes in the original input space can thus be solved by finding an optimal hyperplane in an intermediate space without having to consider this feature space itself in explicit form. The used non-linear mapping $\mathbf{r}$ and the feature space Ψ become manifest within the formulation of the solution only in the form of the Kernel function K. In particular, this allows to use feature spaces with very high or even infinite dimensions.

high-dimensional feature spaces

The desire for very high-dimensional intermediate spaces and hence the motivation for support vector machines arises from a general observation made by Cover in 1965 [41]: The non-linear projections $\mathbf{r}(\mathbf{I}_\alpha) \in \Psi$ of patterns I_α from some N_{in}-dimensional input space Φ exhibit an increasing probability to be linearly separable in the target space Ψ when the dimensionality of the latter grows. For general sets of input patterns, both, the nonlinearity of the mapping and the increased dimensionality of the intermediate feature space are necessary preconditions to ensure an improved linear separability [41].

2.3.2 Hierarchical Approaches and the Neocognitron

basic idea

Hierarchical networks like the neocognitron by K. Fukushima [63] [64] and related approaches [136] [205] divide the solution of a complex classification problem into iterative nonlinear mappings between successive feature spaces. Each layer of neurons combines localized patterns in the output of the preceeding layer to higher-level features that are in turn the inputs of the next level. Viewed in terms of its own input, each layer performs relatively simple classifications. However, with regard to the original input space, the hereby constructed features grow in abstractness and complexity with each hierarchical level. This way, a neuron in the final output layer can exhibit a differentiated and robust sensitivity to the presence or absence of complex features in the originally applied input patterns.

specifying the features

Among other things, the primary difference between the neocognitron approach and ordinary feedforward models is that the features to be extracted by the individual layers are defined externally. Thus, the layers can in principle be trained independently using specialized supervised training algorithms [63] [64]. The success of this training concept critically depends on the selection of adequate features which is usually straight forward for the first layer but poses a substantial challenge in the cases of higher hierarchical levels. Variants of this approach therefore aim to construct a single set of features that can be used in all layers [52] [205] or to find appropriate features through unsupervised learning [63] [64] [136].

Hierarchical networks are commonly applied to image classification tasks like hand-written digit classification [63] [64], object detection in realistic images [205] or character recognition [52].

2.3.3 The Hopfield Network

In the year 1982, the field of artificial neural networks received fresh impulse by a major contribution from the physicist J. Hopfield [98] who introduced a special form of recurrent network architecture. Within a so-called Hopfield network, any neuron $1 \leq i \leq N$ is connected to every other unit $j \neq i$ apart from itself and

the synaptic weights are chosen to be symmetric, i.e., $w_{ij} = w_{ji}$, $w_{ii} = 0$ for $1 \leq i, j \leq N$. Originally, the neurons are fixed to be weighted threshold units with possible outputs of -1 and 1 but the concept has later been extended to neurons with continuous output as well [99].

symmetric weight condition

As a consequence of its highly recurrent architecture, the Hopfield network exhibits distinct temporal dynamics. When the output states O_i of the neurons are set to some initial states $O_i(0) = I_i$ at time $t = 0$, each neuron output $O_i(t)$ will in general be a function of time given by

temporal dynamics

$$O_i(t + \Delta t) = sgn\left(\sum_{i \neq j} w_{ij}\, O_j(t)\right). \tag{2.39}$$

In practice, it is common to implement this updating scheme by randomly selecting a new single neuron in each time step t and recalculating its activation due to the current states of the remaining units to obtain its new output $O_i(t + 1)$. The state of the whole network $\mathbf{O}(t) = (O_1(t), \ldots, O_N(t))$ is a discrete trajectory in the configuration space $\{-1, 1\}^N$. The question arises whether the network ever reaches a stationary point $\mathbf{O}^{\mathrm{s}}$, a so-called attractor, such that after some time t_{s}, the network state obeys $\mathbf{O}(t) = \mathbf{O}^{\mathrm{s}}\ \forall\, t > t_{\mathrm{s}}$.

The remarkable novelty in Hopfield's theoretical investigation of this question is the introduction of an energy function H

energy function

$$H(t) = -\frac{1}{2}\sum_{ij} w_{ij}\, O_i(t)\, O_j(t). \tag{2.40}$$

If the network state evolves according to the update rule given by equation 2.39, it is the central property of the energy function 2.40 that in each time step, the energy H will either decrease or remain constant. Under the condition of symmetric weights, the network is furthermore guaranteed to eventually reach a minimum of the energy function and hence a stationary state after a finite time [98].

For this reason, Hopfield networks are attractive practical models of associative memory. Based on the energy function 2.40, it is possible to derive simple rules for the adequate choice of weights w_{ij} that yield a network whose attractors are given by a predefined set of desired output patterns $\mathbf{T}_\alpha$ [88]. The network can then be regarded to have learned these patterns: If it is applied an external input $\mathbf{I}$ by setting the initial states of all neurons to $O_i(0) = I_i$, $1 \leq i \leq N$, the network output $\mathbf{O}$ will after some finite time t_s settle to the specific attractor $\mathbf{T}_\alpha$ that resembles the original input most. In other words, Hopfield networks are adequate systems to perform autoassociation tasks as introduced in section 1.2.4 and are also often denoted as autoassociative networks.

Hopfield networks as associative memory

Moreover, the formulation of the energy function H underlines Hopfield's statement that the introduced network architecture is "isomorphic with an Ising model", a model from statistical physics that describes ensembles of interacting two-state systems [54] [98]. This important observation readily allowed a vast repertoire of physical theory to be applied to neural network research.

relation to statistical physics

2.3.4 Computing Without Stable States

unsymmetric weights

Once the symmetric weight condition of the Hopfield model is abandoned, the temporal dynamics of the network become more complicated. Being released from an initial configuration $\mathbf{O}^0$, the network is no longer guaranteed to reach a stable attractor. Experiments reveal that, started from different random states $\mathbf{O}^0$, it will in most cases reach a stationary point, and will in the remaining trials at least settle into limiting behaviors and wander around in only a small region of the configuration space [98]. These domains are centered around one particular minimum of the energy function.

At first sight, it might not seem desirable to have a neural network exhibit rich temporal dynamics. To be usable for autoassociation tasks, a network is required to converge to stable states within as short a time as possible. On the other hand, it can be argued that the brain represents a highly recurrent system with complicated temporal behavior and nevertheless performs demanding computational tasks on time scales of 100–150 ms that are not much longer than those defined by the operation of its constituents, i.e., the neurons and synapses [208].

liquid computing and echo states

Motivated by this observation, Mass et al. [129] and Jaeger [110] independently conceived a way of performing complex computations using neural systems without stable states. The basic idea is to use highly recurrent neural networks that can receive external input streams via dedicated input nodes. Such a system could, e.g., be implemented by a Hopfield network with some input sources I_j being connected to the neurons O_i. In any case, the purpose of the network is to act as a dynamical non-linear system whose current response $\mathbf{O}(t)$ is influenced by the present and past inputs $\mathbf{I}(t')$, $t' \leq t$.

A simple readout unit that only processes the current state of the dynamic network $\mathbf{O}(t)$ can then be trained to accomplish non-trivial classification tasks on the original input patterns $\mathbf{I}(t)$. This readout can be as simple as a linear perceptron which permits the use of common optimization techniques like those described in sections 2.1.2 and 2.1.3.

It is a striking observation that different linear classifier systems can successfully be trained for different computational tasks while processing the output of the very same randomly connected recurrent network, as long as the latter exhibits sufficiently complex temporal dynamics [129] [110]. The originally used network models are more elaborate and closer to biology than the Hopfield networks described above but it has recently been shown that the temporal characteristics of a recurrent network of threshold units suffice to allow a simple adjacent linear classifier to perform non-trivial classification tasks [18] [184].

2.4 Hardware Neural Networks

motivation

It is one of the key features of artificial neural networks that their operational principles feature a high degree of inherent parallelism. In a feedforward network, for example, all neurons of a given layer can compute their outputs simultaneously and independently as soon as the outputs of all neurons in the former layer are available. Software implementations on ordinary sequential computers can only

insufficiently exploit this parallelism. Therefore, hardware realizations have been a topic of investigation almost from the outset of artificial neural network research.

organization

The following sections are intended to provide a brief overview on the field of hardware implemented neural networks. Past approaches, present trends, and challenges to the development of dedicated hardware realizations are reviewed on a general level, and the discussion will be concluded by an investigation of suitable training approaches. These reflections serve as the basis for the more concrete considerations that drove the design of the actual neural network chip used for the experiments presented in this work. An exhaustive description of this chip and the concepts that underly its design will be given in chapter 5.

2.4.1 Historical Overview

SNARC

The first dedicated neural network hardware was introduced as early as 1951 by Minsky and Edmonds [142]. The SNARC (Stochastic Neural-Analog Reinforcement Computer) was an electro-mechanical implementation and contained 40 neurons. Later, in parallel to computer simulations [171], Rosenblatt and co-workers developed the MARK I, a hardware realization of the perceptron (section 2.1) that already included automatic learning [83]. It comprised of a grid of 20×20 photoreceptors that could each be connected to 40 of the 512 so-called associator units. The outputs of these units were conveyed to up to 8 threshold neurons (compare figure 2.1). Although the MARK I was not particularly faster than software simulations of the perceptron [22], it was expected that for larger network architectures, the parallelized operation would result in a much higher performance gain.

MARK I

ADALINE

When they originally published the delta rule in 1960 (section 2.1.3), Widrow and Hoff also presented the ADALINE (Adaptive Linear Element), a hardware-implemented threshold unit with 16 variable weights [227]. The weight adjustment during training had to be performed by a human operator. In the succeeding years, the concept was extended to the MADALINE — the multiple ADALINE — in which several single ADALINEs were connected to a network. The MADALINE also included automatic electronic learning.

MADALINE

MARK I–IV

The increasing speed of general-purpose microprocessors let software simulations of neural networks become more and more feasible as well. Software packages like the *MARK I* and *MARK II*[5] [86] fueled the desire for faster simulation speeds and this eventually drove the development of general purpose neural network accelerators (e.g., the *MARK III* and *MARK IV*) [86].

neural networks in CMOS VLSI

In the late eighties, improvements in the fabrication process in conjunction with more advanced design tools promoted the utilization of VLSI (Very Large Scale Integration) technologies for the parallel implementation of neural networks on CMOS (Complementary Metal Oxide Semiconductor) substrates. In 1989, it was shown by Mead [137] that neural systems can successfully be realized by directly exploiting the physical features of the CMOS substrate instead of implementing them on the basis of digital components. Since then, a large variety of analog

[5]The confusing resemblance with the MARK I perceptron by Rosenblatt is usually resolved by setting the names of the software packages in a slanted font.

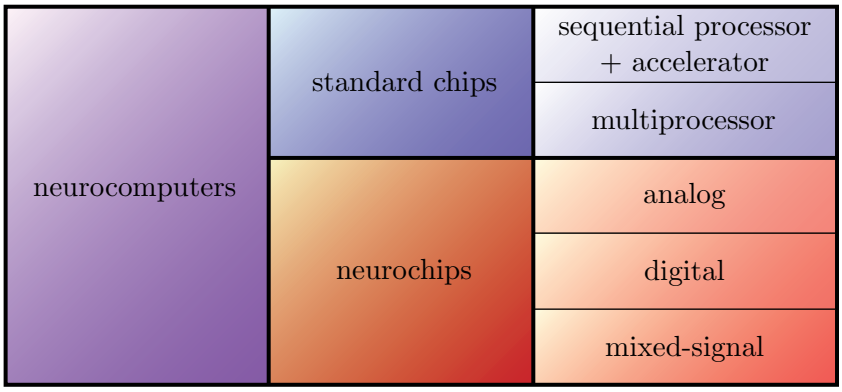


Figure 2.10: *A simple but practical categorization of neural hardware according to Heemskerk [87]. Regarding the last column, the potentially achievable performance of the different approaches is deemed to increase from top to bottom [15] [87].*

and digital neural hardware has been and is still developed by semiconductor companies and in university laboratories [15] [35] [87] [97] [126] [212].

2.4.2 A Categorization of Neural Hardware

neurocomputers and neurochips

In order to categorize the large variety of different neural network hardware approaches, a simple but practical scheme has been proposed by Heemskerk [87]. First of all, it strictly differentiates between neurocomputers and neurochips (see figure 2.10). While reconfigurability, training, and interfacing are regarded as an essential part of the former, they are not necessarily included in the latter. In so far, the term neurocomputer is seen to embrace all kinds of stand-alone hardware neural networks systems, and these can either be built from specialized neurochips or standard chips.

accelerators and distributed systems

digital, analog, and mixed-signal solutions

Neurocomputers which contain standard chips can further be categorized into systems made of a single (sequential) processor plus a dedicated accelerator unit on the one side and systems that are distributed on multiple processors on the other. Besides purely digital or analog approaches, the field of dedicated neurochips also embraces mixed-signal implementations that combine analog with digital computing.

It has repeatedly been argued that regarding the last column of figure 2.10, the potentially achievable performance of the different approaches is expected to increase from top to bottom [15] [87].

2.4.3 Performance Criteria

connections per second

The computational expense of calculating a neural network's response is for the largest part determined by the evaluation of the single synaptic connections. Therefore, the connection is regarded to be the natural basic unit of neural computation [97], and a practical measure to quantify the performance of a neural

network hardware implementation is provided by the achievable number of evaluated connections per second CPS:

$$\text{CPS} := \text{connections/s}. \tag{2.41}$$

complexity of a connection

For the types of networks discussed in the preceeding sections, the calculation of a connection boils down to the multiplication of an input signal I with the associated synaptic weight w in order to yield the corresponding contribution to the total input *net* of the respective neuron. The complexity of this operation clearly depends on the resolutions of both, the input signal and the synaptic weight. This applies to digital implementations, where the adders and multipliers need to be scaled to the precision of the operands, as well as to analog circuits the complexity of which is bound to grow with an increasing desired accuracy. Therefore, a more suitable performance measure is given by the number of so-called connection primitives per second CPPS

$$\text{CPPS} := b_I \cdot b_w \cdot \text{connections/s} \tag{2.42}$$

where b_I and b_w are the respective resolutions of input signals and weight values measured in bit.

connection updates

In the course of the training process, the weight values of a neural network are repeatedly subject to modifications. Similar to the above measures that specify the performance of the network during the evaluation phase, the number of available connection updates per second CUPS

$$\text{CUPS} := (\text{connection updates})/\text{s} \tag{2.43}$$

allows to quantify the reconfigurability of a hardware neural network during training. A high CUPS value is of particular advantage when highly iterative chip-in-the-loop training approaches are to be applied (see section 2.4.5).

2.4.4 Challenges and Present Trends

dedicated hardware vs. software

The continuing increase in the available computing power of conventional computers — provided by an enhanced device integration [147] and a growing operational speed — in connection with the flexibility of software solutions considerably challenge the development of dedicated neural hardware. It can be argued that the potential performance advantage of parallel hardware implementations might not be required at all [183], particularly, as the necessary data pre- and post-processing stages cannot benefit from this parallelization.

analog and mixed-signal systems: drawbacks

Hybrid systems that combine analog with digital computation and which are apprehended to yield the highest performance gain come at the cost of a substantially increased engineering expense. This is mainly due to the poorly automated design process. Furthermore, in order to efficiently exploit the benefits of a parallel realization in CMOS VLSI, a given neural network hardware is preferably designed to optimally suit a specific model. This naturally impedes the simultaneous investigation of multiple different neural network concepts. Regarding the diversity

of existing and competing network paradigms (see section 2.3), investments to a comparably inflexible neural hardware seem risky.

Therefore, the commercial interest in hardware implemented neural networks noticeably decreased during the nineties. The competition with conventional general-purpose microprocessors mostly affected the research on digital neural systems [152]. Nevertheless, besides dedicated implementations for specific applications [48], realizations of neural networks in configurable logic, such as Field Programmable Gate Arrays (FPGAs), lately received new attention [158] [168].

analog and mixed-signal solutions: motivation

It remains that although software simulations and digital hardware solutions are potentially superior to analog or mixed-signal implementations in terms of flexibility, design times, and costs, they eventually suffer from a higher power-consumption and increased silicon area requirements. Aiming for the realization of large neural networks whose complexity and performance are to be comparable to those of biological systems (see section 1.1), only dedicated analog or mixed-signal implementations have the potential to combine the necessary speed, area-efficiency, and low power-consumption. Several such approaches have been proposed [16] [122] [137] [215] [216].

2.4.5 Training Hardware Neural Networks

Beyond speed, efficiency, and scalability, the usefulness of a hardware neural network is ultimately determined by its trainability. Unlike digital implementations, analog systems generally suffer from inevitable device mismatches and fluctuations in the analog signals. This considerably impedes the application of training algorithms that rely on detailed information about the characteristics of all neurons and synapses like the exact individual transfer functions or the current weights. Regarding equations 2.23, 2.24, 2.35, and 2.36, it has to be concluded that this ultimately limits the feasibility of all conventional gradient-based training approaches.

training on unreliable substrates

off-chip training

So-called off-chip learning algorithms try to circumvent this problem by optimizing the synaptic weights according to simplified models of the network implementation. Consequently, the hardware is not in fact involved in the learning process: The resulting weight values need not to be transferred to the hardware before the training has been completed. Although this approach readily allows to utilize the traditional and well-proven algorithms presented in the preceeding sections, it is limited in so far as it can only insufficiently deal with the peculiarities and deficiencies of the actual neuron and synapse implementations.

More promising approaches are on-chip or chip-in-the-loop algorithms that cope with noise and imperfections of the devices by implementing and evaluating the network directly on the dedicated substrate [144] [212]. For on-chip learning, the procedure to calculate the necessary weight updates has to be implemented in hardware and, as implied by the name, has to be located directly on the ASIC.

on-chip training

chip-in-the-loop training

In a chip-in-the-loop setup, the training algorithm is realized externally and can, e.g., be executed as software on an ordinary computer (see figure 2.11). To be suitable for this kind of training approach, a neural network ASIC needs to provide means for the external specification of all variable network parameters like the weight values and/or the synaptic connectivity. During training, slightly modified

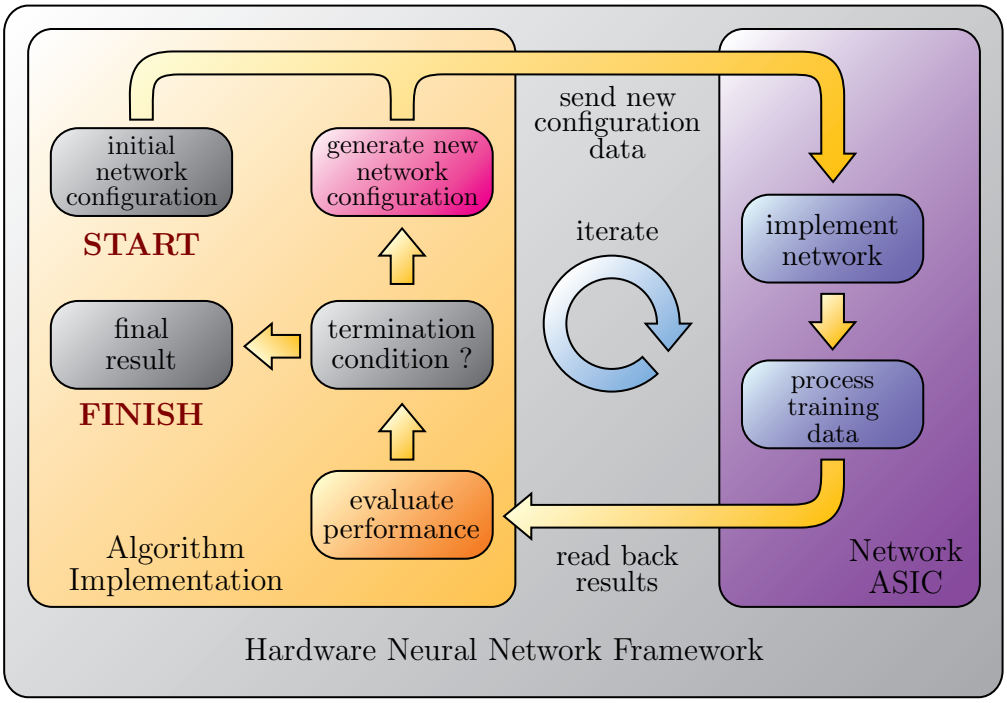


Figure 2.11: *The general scheme of a chip-in-the-loop algorithm. First, the current network configuration is sent to the neural hardware for evaluation. Then, on the basis of the resulting network response to the training data, the algorithm applies adequate modifications to the network that in the ideal case lead to an improved performance during the next evaluation. The whole process is iterated until a suitable termination condition is fulfilled, e.g., the network achieves the desired performance. Since the algorithm itself is not included on the used network ASIC but is implemented separately, e.g., as software on a computer or within a configurable logic, large amounts of configuration and output data have to be transferred repeatedly. Therefore, the algorithm implementation and the network chip need to be embedded within a suitable hardware neural network framework that provides efficient means of communication (see chapter 6).*

variants of the network are iteratively implemented on the chip. After they have been presented the desired input patterns, the resulting output is read back by the algorithm for evaluation. Hence, while such a setup provides a considerably larger flexibility compared to on-chip solutions, it also leads to massive data transfer between the neural hardware and the algorithm implementation. In order to retain high training speeds, the algorithm implementation and the network chip therefore need to be embedded within a suitable hardware neural network framework that provides an appropriate infrastructure to support fast data transfer (see chapter 6).

training a black-box system

Furthermore, if the training process is to be made independent of a necessarily idealized and imperfect network model, the generation of beneficial modifications to the synaptic weights is restricted to be based on the evaluation of the network response. Optimization algorithms that do not rely on any model of the optimized

system but are capable of finding improved versions merely by investigating the performance of one or more given candidate solutions are commonly denoted as model-free or black-box approaches (see also section 3.3.3).

evolutionary algorithms

Against the background of what has been said above, it can be reasoned that such model-free algorithms promise to be the most adequate approach for the training of analog or mixed-signal hardware neural networks. A prominent example for this kind of optimization procedure are the so-called evolutionary algorithms that mimic the principles of natural evolution and which shall be introduced in more detail in the following chapter.

Chapter 3

Evolutionary Algorithms

I have called this principle, by which each slight variation, if useful, is preserved, by the term Natural Selection.

Charles Darwin, On the Origin of Species

The field of evolutionary computation embraces a variety of algorithmic approaches that are inspired by the mechanisms of natural evolution. Against the background of the previous chapters that described how the principles of neural information processing are successfully utilized in artificial systems, it is not surprising that natural evolution has been considered as a paradigm for the construction of powerful optimization algorithms. Over the course of time, evolution has created an enormous diversity of life-forms that are each remarkably well adapted to their respective ecological niche. Evolutionary algorithms aim to reproduce this efficiency by transferring the underlying principles of biological evolution to common optimization problems. Like in the case of artificial neural networks, the discussion of evolutionary algorithms shall be preluded by a short investigation of the biological original.

3.1 Natural Evolution

the origins of life

The origins of the planet that we inhibit date back to about 4.6 billion years ago. Within the first 500 million years of its existence, the earth developed a solid shell, the lithosphere, and a surrounding layer of gases called the atmosphere. Approximately 4 billion years ago, the atmosphere cooled down to below 100°C [66] [201], and it is believed that very simple life-forms initially appeared several 100 million years later.

From the very beginning, life in all its variants has influenced and changed both the surface as well as the atmosphere of this planet to a far-reaching extent. Among the most primordial groups of life-forms are the archeabacteria and eubacteria that produce oxygen by photosynthesis and thereby gradually converted the atmosphere over a period of approximately 3 billion years. About 350 million years ago, the concentration of oxygen settled to its present value. Among other things, this distinct environmental change is assumed to have played the central

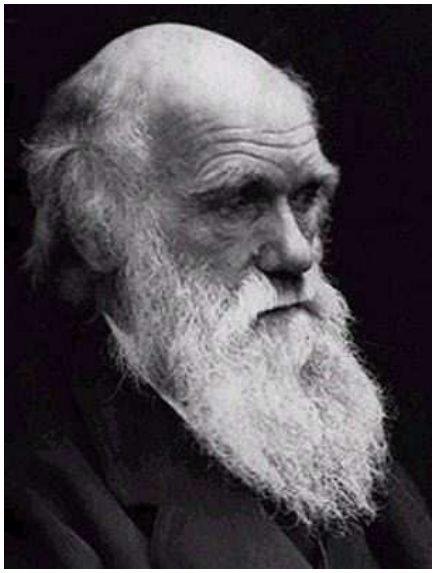


Figure 3.1: *Charles Darwin (1809–1882) was the first to explain the evolution of biological species by natural selection. The publication of his famous work* "On the Origin of Species" *[42] in November 1859 caused public reactions that ranged from fascination to disgust. Today, complemented by molecular genetics, Darwin's theory constitutes the foundation of evolutionary biology [66] [201].*

role in promoting the evolution of multicellular organisms which in turn resulted in an exploding increase in the diversity of life-forms [66] [201].

biological diversity

Today, biologists describe approximately 2 million different species of animals and plants that nevertheless represent definitely less than 10 % of the species that ever existed during life's history. Probably, this fraction is even below 1 % [201]. In other words, there can be assumed to have appeared (and for the largest part vanished) in the order of a billion species on this planet within a period of around 3.5 billion years. Hence, on average — although this is admittedly a somewhat simplifying calculation — a new species developed every few years.

3.1.1 The Principles of Darwinian Evolution

In 1859, Charles Darwin proposed a theory to explain the astonishing biological diversity and its underlying mechanisms that in its central aspects still holds today [42]. Within Darwin's model of natural evolution, an essential role is accredited to selection. Selection naturally arises whenever living organisms with their basic instinct to reproduce compete for vital resources in an environment that can only sustain a limited number of them. Given these constraints, the individual organisms that are best adapted to the environmental conditions will outrival their competitors and are more likely to survive and to produce offspring. This principle by which selection favors individuals with higher fitness, i.e., a greater ability to survive and to reproduce, is also known as survival of the fittest.

selection

variation

Besides selection, the second main force behind evolution is variation. Each organism exhibits characteristic physical and behavioral features that affect its fitness and which are commonly called the phenotypic traits. During the lifetime of an individual, its specific combination of traits is tested for their beneficial influence on the individuals ability to survive and to produce offspring such that if advantageous, these traits will be passed to the next generation. Otherwise, the individual is likely to die without reproducing and its specific combination of char-

acteristics will be discarded. Small occasional random variations of the relevant features, so-called mutations, appear during reproduction from one generation to the other. This leads to the occurrence of slightly modified compositions of traits in the offspring. Once again, these new featural combinations get evaluated and are themselves put to the test of natural selection with the useful variations being spread by reproduction and the unfavorable ones dying out. As this process iterates, the population of organisms gradually changes and evolves to be comprises of individuals that are more and more adapted to the specific demands of their environment.

individuals and selection

It is worth emphasizing the different roles of the individual on the one hand and the population on the other. Individuals are said to be the unit of selection. While their specific characteristics determine how well they perform in passing these very traits to succeeding generations, they are not in fact being modified directly. Rather, the continuing selection process in combination with occasional variations leads to continuous changes in the population and cause it to slowly converge to individuals whose features optimally suit the environmental conditions. Hence, the population is to be regarded as the unit of evolution.

populations and evolution

3.1.2 Evolution on the Genetic Level

genotypes and phenotypes

While the Darwinian principles describe evolution from a macroscopic point of view, molecular genetics provide insight into its underlying mechanisms on a microscopic scale. Each living organism can be viewed as a duality of its phenotype that represents the entirety of its physiological, morphological and behavioral traits, and its genotype that encodes the combination of these features on a molecular level. This encoding is complete in the sense that the genotype contains all information that is necessary to build the phenotype.

genomes and genes

The genetic information about an organism is summarized within its genome which itself consists of single genes. The gene is the basic functional unit of genetic encoding particularly with regard to heredity. In natural organisms, the mapping between individual phenotypic traits and the corresponding encoding genes is not one-to-one. A single gene might affect multiple phenotypic traits, a phenomenon called pleitropy, and one specific characteristic of the individual may in turn be influenced by more than one gene which is denoted as polygeny.

chromosomes

The genetic material of an organism, i.e., the complete set of its genes, is organized in several chromosomes that contain the single genes in a linear arrangement. The number of chromosomes varies from species to species and while human genomes include 23 of them, e.g., many birds exhibit in the order of 40 [201]. Higher life-forms include two copies of each chromosome in most of their cells. These cells as well as the respective organisms are called diploid. Cells that participate in the reproduction process, i.e., sperms and egg cells, the so-called gametes, contain only one version of each chromosome and are denoted as haploid.

variation during reproduction

All phenotypic variations are caused by modifications on the genotypic level, more specifically, by the mutation of single genes and the recombination of genetic sequences during sexual reproduction. Recombination occurs in the course of fertilization where the haploid sperm cell merges with the haploid egg cell to

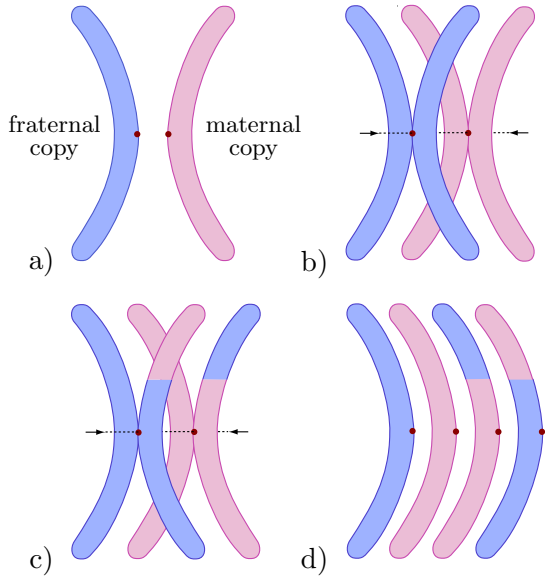


Figure 3.2: *Schematical illustration of the biological process of meiosis.* ***a)*** *The starting point is given by the two corresponding chromosomes inhered from the mother (maternal copy) and the father (fraternal copy).* ***b)*** *First, each chromosome is doubled and the four resulting strands are aligned.* ***c)*** *In the second step (crossing-over), genetic material is swapped between one fraternal and one maternal chromosome.* ***d)*** *The resulting chromosomes suffice for four haploid cells.*

form a diploid cell called the zygote. Hence, the zygote contains two copies of each chromosome, one inherited from the mother and one from the father. The succeeding development of the new organism does not change the genetic information and therefore all its cells contain copies of the same set of chromosomes found in the original zygote.

meiosis

Nevertheless, further mixing of genetic code takes place during meiosis [50] [66] [201], a special cell-division process by which haploid gametes are formed from diploid cells (see figure 3.2). Meiosis starts with the physical alignment of the two corresponding chromosomes inherited from the mother and the father. Both chromosomes are then doubled to yield a total of four aligned chromosomal strands called chromatides. In the last step, known as crossing-over, one maternal and one fraternal copy are broken up at the same random point and exchange parts. The resulting chromosomes are used to form four haploid cells, two of which contain unchanged genetic material from either the mother or the father. The remaining two cells include the respective gene sequences in new and complementary combinations.

mutations

The reproduction process by which chromosomes are copied during meiosis is subject to infrequent errors. Thus, in addition to the possible recombination of genetic material, the four resulting haploid gametes will contain random modifications of the original genetic information caused by various types of mutations [66] [201].

It is an important observation that the information-flow between genotype and phenotype is unidirectional and that variations occur exclusively on the genotypic level. As indicated above, any changes of the phenotypic features during the lifetime of an individual, e.g., through learning or external influence, do not affect the genetic code stored in its cells and are thus not passed to the offspring via genetic inheritance. Hence, it can be said that the the two essential mechanisms of evolution, i.e., selection and variation, are each confined to operate on a different

level. While selection acts solely on the phenotype, all variation processes affect only the genotype.

3.1.3 Speciation

inhomogeneous populations

So far, it has indirectly been assumed that the regarded population consists of individuals from one single species. Thus, all individuals are biologically compatible in that any two of them can have offspring (subject to the natural sexual constraints). A given environment will contain individuals from various species. Seen from the perspective of one population, the presence of other species that compete for the same resources can be regarded as a special characteristic of the environment that contributes an additional aspect to the selective pressure on the individuals.

allopatric speciation

In this sense, natural selection can not only be seen to act on single organisms but also on whole species. In nature, this has lead to the adaption of species to different available ecological niches. New species can develop when individuals of one species are separated into two populations that do no longer exchange genetic material among them. Most often, this separation is caused by geographical isolation (allopatric speciation), and the members of the two populations have to survive in different environments, each with its special demands and constraints. The interaction of variation and selection will therefore favor different types of phenotypic traits within the two populations. Eventually, the differences between both groups will have grown to be such that they are to be regarded as distinct species [201].

3.2 Evolutionary Algorithms: An Overview

population based optimization

Evolutionary algorithms apply the principles of natural evolution to the solution of optimization problems. Instead of modifying a single instance of the system to be optimized, they process a whole group, a so-called population, of candidate solutions that are now referred to as individuals. The performance of each individual is measured by a fitness function which is usually (but not necessarily) chosen in a way such that the better the performance of the candidate solution, the higher the fitness value.

fitness function

An evolutionary algorithm starts with the generation of an initial population of candidate solutions and the evaluation of their fitness values. Based on the fitness, a selection procedure then determines those individuals of the population that are allowed to produce offspring in the form of new candidate solutions. The reproduction itself is accomplished by applying appropriate variation operators to the original individuals.

parent selection

recombination and mutation

Two types of variation operators can be distinguished: While recombination acts on two or more individuals to yield one or more new candidate solutions, mutation operators are applied to single individuals, resulting in one offspring. These operators do not act on the individuals themselves but on suitable representations, i.e., special data structures that contain all parameters of a candidate solution that are to be optimized by the algorithm.

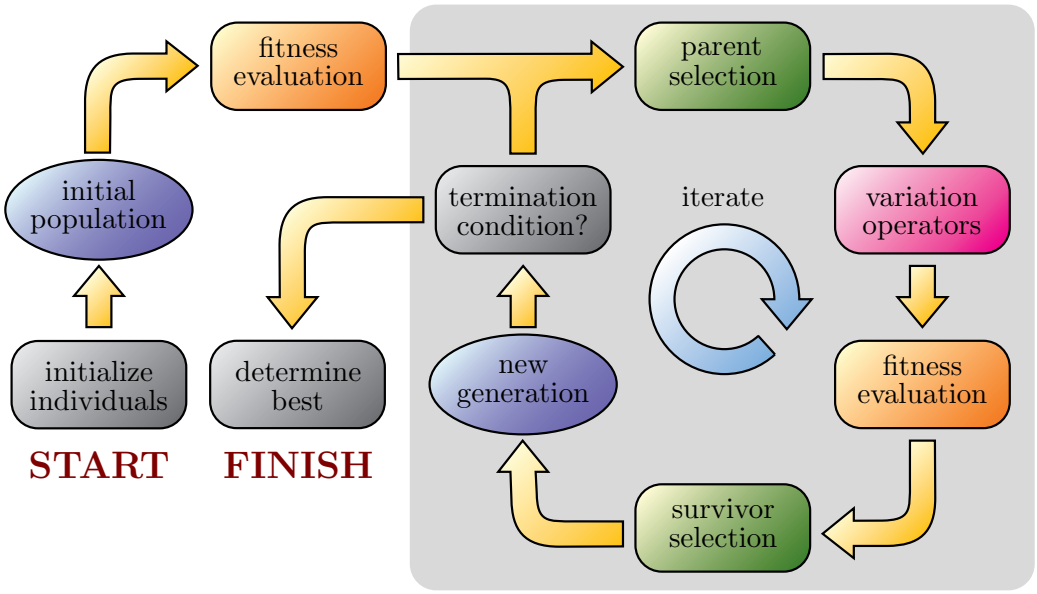


Figure 3.3: *The general scheme of evolutionary computation. Starting from an initial population, the individuals are repeatedly subject to variation operators and selection procedures until a given termination condition is fulfilled. Typically, the result is taken to be the best individual in the final generation. The main aspects of an evolutionary algorithm are its fitness function, selection schemes, and the used genetic representation in connection with the used variation operators.*

survivor selection

Most commonly, the size of the population is kept constant and the resulting offspring competes with the old individuals for places in the next generation. Like the parent selection process, the assigning of individuals to slots in the succeeding generation, also called survivor selection, can be based on the fitness. However, it can also be determined by the age of the individuals. Both selection procedures can be more or less stochastic or completely deterministic.

Similar to natural evolution, evolutionary algorithms rely on the two essential principles of variation and selection that are each represented by the recombination and mutation operators and by the parent and survivor selection procedures, respectively. When the above process is iterated, the individuals in the population will improve in their ability to solve the given problem reflected by an increasing average fitness value of the population. This is to be viewed in analogy to the evolutionary adaption of biological organisms to optimally suit their environmental conditions.

The algorithm proceeds until a given termination condition is fulfilled, e.g., one individual in the population represents a solution with sufficient quality (high fitness) or a fixed computational limit is exceeded. Figure 3.3 illustrates this general scheme of evolutionary algorithms.

historical overview

Historically, the idea of evolutionary computation can be traced back as far as to the forties of the past century [57] and since the 1960s, four main streams of evolutionary algorithm implementations have emerged: evolutionary programming [56],

genetic algorithms [46] [71], evolution strategies [9] and, more recently, genetic programming [121]. These approaches differ mainly in the types of optimization problems they are commonly applied to as well as in the specific implementations of the single components shown in figure 3.3.

main fields

While evolution strategies and genetic algorithms are typically employed for parameter optimization problems, evolutionary programming has initially been developed to simulate evolution as a learning process with the aim of generating artificial intelligence. Genetic programming, finally, is to be positioned in the field of machine learning. Apart from that, the distinction between these approaches as well as their respective traditional conventions and notations merely have historical origins. Today, they are all seen as subareas of the whole field of evolutionary computation [140] [50]. In the spirit of this generalized view, the succeeding sections shall discuss the individual parts of an evolutionary algorithm in more detail but on a generic level. Explicit realizations will be introduced in section 3.4.

3.2.1 The Main Constituents of an Evolutionary Algorithm

The following main components have to be specified in order to completely define a particular evolutionary algorithm:

- The *representation*, also denoted as the *genetic coding*, defines the data structure that contains the information about all parameters of a candidate solution that are to be optimized by the algorithm.
- The *variation operators* determine the way in which recombination and mutation are implemented on the basis of the above representation.
- The *fitness function* provides a measure for the quality of an individual. Its value is the actual quantity that is optimized by the algorithm.
- The *parent selection* and *survivor selection* procedures operate on the whole population in that they transform the old generation of individuals to a new one.

The Representation

An evolutionary algorithm is desired to eventually yield a system that represents the solution to a given problem. In terms of the simulated evolution, this system and any candidate solutions evaluated during the search process can be regarded as the phenotypes. In contrast, the objects that are directly processed by the algorithm are the corresponding genotypes. Each genotype, henceforth also called genome, is an instance of a data structure that summarizes all variable parameters of the respective candidate solution. The organization of this data structure as well as the respective mapping from the phenotype to the genotype are both commonly denoted as the (genetic) representation.

example: network weights

As an example, consider the case of a neural network with a fixed architecture possessing n synapses. Here, each phenotype is completely defined by specifying all synaptic weight values w_i, $1 \leq i \leq n$. Thus, a simple choice of genotype is

a vector $\mathbf{g} = (g_1, \ldots, g_n) \in \mathbb{R}^n$ of real numbers that each represent one weight $w_i = g_i$. However, another valid representation would be to code all weight values in binary form with a resolution r and arrange them to a linear binary string $s \in \{0,1\}^{(r \cdot n)}$.

genotype space

The entirety of all different genomes that can possibly be realized by a given representation forms the genotype space $\mathbb{G}$. In principle, this space is distinct from the phenotype space $\mathbb{P}$ that consists of all candidate solutions to the investigated problem. For the above examples, the mapping between them is straight forward. Depending on the problem and the chosen representation, the relation between the genotype space and the phenotype space can be relatively complex [117] [121].

It is important to note that an evolutionary algorithm performs its search in the genotype space. Hence, when designing the representation for a given optimization task, it has to be ensured that the desired solution can be defined by a valid genome of the chosen kind, i.e., that there exists a point in the genotype space that codes an acceptable solution to the problem. Since for most realistic applications, this point will not be known in advance, the selection of a suitable representation often has to be based on reasonable assumptions about the general nature of the solution.

The Variation Operators

During evolution, new individuals are created from the old ones by applying adequate variation operators to the respective genotypes. In other words, in search of the optimal solution, the algorithm progresses through the genotype space by virtue of these operators. It is crucial for the success of the optimization process that the used genetic representation in connection with the corresponding variation operators suit the investigated problem. More precisely, both are to be chosen such that starting from a random initial population of points in the genotype space and iteratively applying recombination and mutation, the optimal solution can be expected to be reached within a finite number of steps with a sufficiently large probability. Again, this would in principle require the exact knowledge of the solution. In practice, the feasibility of a set of variation operators for a given task can only approximately be estimated in advance.

traversing the genotype space

Recombination Recombination operators are also called crossover operators. They merge the information stored in the genotypes of two individuals, denoted as the parents, to yield one or two new genomes, named the offspring. This is usually achieved by randomly breaking up each of the original genomes into two or more parts and exchanging some of the resulting segments among them.

biological motivation

The use of a crossover operator is inspired by the crossing-over that appears during meiosis in biological systems[1] (see section 3.1.2). While its purpose is to recombine advantageous features of the parents to form potentially improved new individuals, the application of crossover is not guaranteed to be beneficial. The quality of a specific implementation of the recombination process can be measured

practicability

[1]Note however, that while the crossover operator acts during reproduction, the biological crossing-over occurs earlier during the development of gametes.

by the average probability by which it produces individuals that have equal or better fitness than their parents.

crossover and genetic coding

For a crossover operator to maximize this probability, its design has to account for the structure of the chosen representation and vice versa. In particular, it is the crossover operator that defines the smallest functional unit of the genome as being the minimum part that is not broken up during the segmentation step. In analogy to the biological original, this basic unit is called a gene and the different values that it can assume are referred to as alleles.

Regarding the two examples above, the gene would in the first case be one real number g_i and the vectors of the two parents would only be cut apart between any two components g_i and g_{i+1}. In the second encoding, a gene could correspond to a single binary digit of the string s. Note that in the first example, the mixing process merely recombines the weights of the two parent networks. The second approach, on the other hand, can produce new weight values by cutting apart and recombining sequences of the original strings that in fact formed a translational unit by coding one weight. Such a behavior might be beneficial, but due to the binary nature of the encoding, the outcome of this recombination process is highly unpredictable.

While this already provides a first insight into the close relation between the recombination operator and the chosen genetic representation, a thorough discussion of this topic shall be deferred to sections 3.4.4 and 3.5.

Mutation A Mutation operator acts on single genotypes by applying a series of stochastic modifications. Often, it affects each gene separately by randomly changing the respective allele to a new value. Mutation operators that act on whole groups of genes simultaneously are conceivable as well but in any case, the changes are desired to be stochastic and unbiased and are applied with a given low probability. Returning to the examples of the real valued vector and the binary string, a simple mutation operator could in the first case replace single components with new random numbers and in the second case randomly flip single bits.

single gene mutation

mutation and diversity

The presence of mutations is vital for the performance of an evolutionary algorithm since they support the genetic diversity within the population. This way, they balance the effects of selection that tend to reduce this diversity by repeatedly discarding the genetic material of individuals with low fitness. Viewed in terms of the search space, mutations ensure that this space is connected. For theoretical convergence proofs of evolutionary algorithms, it is often required that any allele can be mutated into any other with nonzero probability [173] [174].

adjusting mutation

On the other hand, too frequent or too violent modifications of the genetic information in the population run the risk of impeding evolutionary progress by destroying advantageous combinations of alleles. The correct adjustment of the mutation rate in connection with the appropriate selective pressure is one of the main issues when tuning the parameters of an evolutionary algorithm.

The Fitness Function

The fitness function has to be chosen as to provide a fair measure for the quality of the candidate solutions that are evaluated during the search process. Since the selection procedures described below are based primarily on the fitness values of the individuals, the evolutionary algorithm can be seen as to optimize the fitness. At the same time, the desired result of the search is a system that solves the given problem.

smooth fitness measures

In this sense, it is evident that the fitness function has to be designed such that it assumes its maximum value on the optimal solution(s)[2]. Furthermore, it is preferred to be as smooth as possible, i.e., to reward even small improvements in the quality of an individual with a measurable increase in fitness. If two or more individuals can exhibit the same fitness value although one is actually closer to the desired solution than the others, the efficiency of the selection procedure is compromised.

Finally, it is understood that a worse individual should never be assigned a higher fitness value than a better one. But it turns out that depending on the task at hand, it is not necessarily straight-forward to formulate a simple fitness function that is guaranteed to do so (see section 8.3.3).

The Selection Schemes

parent selection

Given a population of individuals with known fitness values, it is the purpose of the parent selection scheme to chose some members of the population that will be used to create offspring: New candidate solutions that are potentially accepted for the next generation. Usually, this is selection is probabilistic and individuals with higher fitness stand a better chance to be selected than those with lower fitness.

survivor selection

After the desired number of offspring have been created and their fitness values have been evaluated, the survivor selection process determines those individuals among the old and the new candidate solutions that are to form the new generation. Unlike the parent selection scheme, survivor selection can in addition to the fitness values of the individuals also consider their age, such that offspring are preferred over the parents.

selection in different fields

In practice, many implementations of evolutionary algorithms confine themselves to only use either of the two selection processes. For example, genetic algorithms traditionally apply only parent selection and fill the new population completely with offspring[3] [46] [71]. Evolution strategies chose the parents randomly and regardless of their fitness. Also, they build the new generation from the offspring (and in some case the old population) in an entirely deterministic way [9] [50].

adjusting selection

Selection is the pushing force behind the improvement of the individuals in the population and as such, it has to favor good individuals over inferior ones. On the

[2] It is straight forward to reformulate all statements of this section to suit a fitness function that monotonically decreases with the quality of the individual. Obviously, this function is to be minimized rather than maximized.

[3] In fact, this can also be seen as a completely deterministic survivor selection that discards all individuals with an age greater than one generation.

other hand, the exploration of new solutions depends on the diversity within the population. Even bad individuals might partially comprise of allelic combinations that are beneficial for the solution of the problem. Thus, in order to maintain a sufficient genetic variance, the selection process is best not made too rigorous. A brief overview of various selection schemes is given in section 3.4.1.

3.3 General Features of Evolutionary Algorithms

The preceeding section has outlined the basic methodology of evolutionary computation. Before several specific implementations of the main components will be introduced in 3.4, this section shall summarize some general properties of evolutionary algorithms that already emerge as a direct consequence of their underlying principles. Potential difficulties of the evolutionary approach will be discussed and extensions to the basic concepts of evolutionary optimization will be introduced that can compensate for some of these deficiencies.

general characterization

As a general characterization, evolutionary algorithms can be sorted into the category of generate-and-test approaches [50] but are distinct among the algorithms of this family under several aspects. First, they are population based. Second, they are stochastic and third, they often rely on the recombination of candidate solutions to form new ones. Especially the first two features have important consequences for the progression of the search process.

3.3.1 Evolutionary Algorithms as Global optimizers

The use of a population of individuals effectively enables an evolutionary algorithm to probe the search space at multiple different points simultaneously. Although initially, the candidate solutions are randomly spread across the whole space, later generations will by virtue of the selection process and the variation operators be clustered within regions of high fitness. It is a common observation that the population will eventually converge towards a single located domain and slowly approximate the optimal solution within this area. These two distinct phases of the evolution process are often denoted as exploration — finding regions of high fitness in the search space — and exploitation, i.e., the concentration of the search in the vicinity of the best encountered solutions [50].

exploration and exploitation

local maxima and diversity

The exploration phase greatly benefits from the diversity in the population. Maintaining a whole group of different candidate solutions potentially increases the probability of the algorithm to locate the global maximum of the fitness function. Hence, it reduces the risk of getting trapped in local maxima like it is a common problem of gradient ascent approaches (section 2.1.3) or even other stochastic search algorithms (see section 4.3.1).

On the other hand, once the algorithm has reached the exploitation phase, too strong variance interferes with an efficient approximation of the best solution in the current neighborhood of the population. Therefore, evolutionary optimization is often discussed in terms of a trade-off between exploration and exploitation.

genetic drift

The eventual convergence of the whole population around one optimum is a phenomenon commonly known as genetic drift. In principle, it is a desirable

feature that the population will after several generations be clustered in regions of high fitness. However, in the case of multimodal tasks that exhibit more than one optimum of the fitness function and where only one of these maxima is a global one, the genetic drift can cause problems. If the diversity of the population is lost too early during the search process, the algorithm might get stuck in local optima and thus miss the global optimum. This unfavorable behavior is usually referred to as premature convergence.

premature convergence

Against the background of these considerations, the optimal adjustment of the selection pressure and variance parameters of an evolutionary algorithm can be identified with the problem of balancing the efficiencies of the exploration and exploitation phases. Ultimately, the chosen parameters are desired to avoid premature convergence and at the same time allow for an efficient approximation of the global optimum during the exploitation phase.

3.3.2 A Modular View of Evolutionary Algorithms

Since the fitness function has to quantify the ability of an individual to solve the given problem, it is clearly specific to the currently investigated task. But even if its formulation might be difficult (see section 8.3.3), the fitness function depends only on the desired behavior of the optimized system or on specific requirements to the latter. In other words, the fitness function is defined on the phenotype space. As such, it is independent of the used genetic representation and the corresponding variation operators.

autonomy of the fitness function

In turn, the explicit form of the genotype and the applied genetic operators can be chosen without regard to the details of the fitness evaluation. Rather, the progress of the evolution depends on how far the coding and the operators pay tribute to the structural and organizational features of the system to be optimized.

autonomy of the genetic representation and operators

The selection process, finally, operates solely on the basis of the fitness of the individuals. It is neither concerned about how these values have been obtained or about the way by which new offspring is created. In so far, the selection scheme constitutes the part of an evolutionary algorithm that is least problem specific.

autonomy of the selection schemes

From this point of view, the three main components of an evolutionary algorithm, i.e., the fitness calculation, the selection scheme and the genetic representation in connection with the variation operators, are inherently independent. The detailed realization of each part can be chosen without necessarily affecting the implementation of the others. Each component can easily be replaced by another version in order to test the feasibility of different algorithm setups for a given task.

3.3.3 Evolutionary Algorithms as Model-Free Heuristics

The above considerations already indicate that specifics of the investigated problem become manifest within the formulation of a corresponding evolutionary algorithm primarily in the form of the fitness function and possibly the design of the genetic coding. The former is undoubtedly peculiar to the particular task at hand. The latter is preferably chosen as to account for the structural characteristics of the optimized system in order to increase the probability of the recombination

operator to produce improved offspring (see also sections 3.5 and 4.1.2).

However, many implementations of evolutionary algorithms, like evolution strategies, do not use any recombination operator and rely solely on mutation and selection [9]. Also, recombination can contribute to the variance in the population and thereby benefit the exploration of the search space even if it is not optimized towards producing good offspring. Finally, the knowledge of how to best arrange the genetic representation and to implement the recombination process is often simply not available. This does not prevent the evolutionary approach — with or without recombination — from being successfully applied to a multitude of optimization problems (e.g., [19] [31]).

independence of the task

Thus, aside from an adequate problem-dependent fitness function and a specification of the number and range of the variable parameters, no further information about the optimized system is required. Neither does an evolutionary algorithm rely on any model of how the candidate solutions actually work, nor does it depend on any derivatives of the performance measure with respect to the parameters. Therefore, evolutionary algorithms are often denoted as model-free, or black-box, approaches. As such they can easily be applied to a wide range of optimization tasks, even to those for which no other optimization strategy can readily be formulated. In fact, it is the property of being model-free that constitutes the main motivation for using evolutionary algorithms to train hardware neural networks (see 2.4.5). The evolution of artificial neural networks will be the topic of chapter 4 which will also discuss some alternative model-free optimization algorithms.

evolving hardware neural networks

Heuristics and the No Free Lunch Theorem

Despite the potential problem of premature convergence, the principles of evolutionary computation prove to be of great value for the solution of optimization problems in practice. Nevertheless, there are some issues concerning evolutionary algorithms that shall not remain unmentioned.

evolutionary computation as a heuristic

The evolutionary approach is based on simple and transparent ideas that are inspired by natural evolution. Regarding the efficiency of the biological original, evolutionary algorithms are expected to provide feasible means for the solution of optimization problems. But it is important to note that apart from some selected artificial tasks and specific implementations of the algorithm [174], it cannot be proven that an evolutionary algorithm is guaranteed to find a suitable solution of sufficient quality within a reasonable time [50] [102]. For this reason, the evolutionary approach is often referred to as a heuristic. The term heuristic is not unambiguously defined but can be understood as to denote a technique for the design of efficient optimization algorithms "for which no one is able to guarantee at once the efficiency and the quality of the computed feasible solutions" (taken from Hromkovič, 2004 [102]).

heuristics and robustness

It is a common property of heuristics to be robust in the sense that they can be applied to a large variety of optimization tasks, even if these problems differ significantly in their combinatorial structures. In the 1980s, it has been anticipated that evolutionary algorithms outperform other techniques on a wide range of problems [71] and that they are only beaten by dedicated algorithms that are

the No Free Lunch theorem

specially designed for particular tasks. Recent studies, however, lead to a correction of this view. In simple terms, the famous No Free Lunch theorem (NFL) by Wolpert and Macready [228] states that when averaged over the space of all possible optimization problems, all nonrevisiting, model-free algorithms will show the same performance. The term nonrevisiting refers to the demand that the algorithm does not generate and test the same point in the search space twice. This is not initially a property of evolutionary algorithms but can easily be incorporated.

The conclusions drawn from this theorem are twofold. First, if any algorithm performs better than others on a set of problems, it pays for this by performing worse on other tasks. Second, for a given problem, it is possible to break the limitations of the NFL by abandoning the model-free nature of the algorithm, hence, by incorporating problem specific knowledge.

practicability of evolutionary algorithms

It can be argued that by designing an appropriate genetic coding and corresponding operators, task specific information is automatically incorporated into the algorithm. Furthermore, evolutionary algorithms have repeatedly demonstrated their ability to find good solutions for many problems that could otherwise not easily be solved. In practice, this is most often sufficient, even if the found system cannot be proven to represent the achievable optimum or if other algorithms can theoretically perform better. In so far, being a heuristic and as such being theoretically limited by the NFL does not necessarily pose severe restrictions to the applicability of evolutionary algorithms.

3.3.4 Extensions to the Basic Concept

Several mechanisms are present in nature that support the exploration of different environmental niches and thus lead to the development of distinct species. When tackling multi-modal problems with evolutionary algorithms, it is equally desirable to explore various different regions of the search space in order to avoid premature convergence towards a suboptimal solution. It is reasonable to assume that the efficiency of the exploration phase could be improved if the basic concept of evolutionary computation is extended by some of the mechanisms that also promote speciation in nature. This has in fact been done in various ways.

island model algorithms

- In the so-called island model algorithms (also coarse-grain parallel or simply parallel evolutionary algorithms), multiple separated populations of individuals are evolved in parallel. After a fixed number of generations denoted as an epoch, several individuals from each population are exchanged with the neighboring populations (migration) [38] [39] [128] [161] [172] [204]. The topological arrangement of the populations is usually chosen as to suit the architecture of the parallel system on which the algorithm is implemented. Most often, it is a ring, torus or hypercube.

diffusion model algorithms

- Diffusion model algorithms (also fine-grain parallel, distributed or cellular evolutionary algorithms) divide the population into smaller, partly overlapping subpopulations (demes) that are distributed in an imaginary algorithmic space. Selection and mating only occur within the subpopulations [72] [103] [132] [150] [220] [222]. A simple exemplary realization of this

concept is to distribute all individuals on a (toroidal) grid such that each individual can only be recombined — and has to compete with — individuals on neighboring grid points.

- The above approaches both aim to simulate restrictions on the mating of individuals like they arise from geographical separation in nature. Another idea is to introduce an appropriate distance metric in either the genotype or phenotype space that quantifies some kind of relationship or similarity between individuals. Those pairs of candidate solutions with a sufficiently small mutual distance are then regarded as belonging to the same species and are allowed to have offspring. Otherwise, the individuals are considered as biologically incompatible and cannot be recombined. Various types of distance metric have been introduced [45] [194]. The distance can be based on features of the individuals (resp. their genotypes), but the membership to a given species can also be assigned randomly and made subject to recombination and mutation [196].

distance metrics and mating constrictions

- Instead of using the distance between individuals to determine appropriate restrictions on potential mating pairs, the similarity concept can alternatively be utilized to bias the survivor selection process. In fitness sharing [70], the fitness of each individual is normalized by the number of other candidate solutions that fall within a specified metric distance. This effectively reduces the fitness for whole groups of individuals that are too similar and — since the selection process favors candidate solutions with high fitness — can be assumed to support the diversity within the population.

distance metrics and biased selection

fitness sharing

  The crowding algorithm [46] [131] incorporates a special replacement selection scheme that favors offspring to preferably replace those original members of the population to which it exhibits a close similarity. Thereby, initial subpopulations that inhibit different niches of the search space are likely to be preserved. However, their size does not depend on their fitness as it is the case for fitness sharing.

crowding

All of the above concepts have proven to constitute fruitful extensions to the basic evolutionary algorithm framework. Nevertheless, they also introduce additional parameters that require reasonable adjustment. A detailed investigation of these approaches, their benefits and problems cannot be provided here. The interested reader is referred to either the original publications or the overview given in [50]. Instead, it shall be discussed in how far the introduction of the above concepts compromises the independence of the main components of evolutionary algorithms that has been claimed in section 3.3.2.

Do Extensions Challenge Modularity ?

The first two of the above extensions primarily involve a specific organization of the population. Within the parallel populations of the island model or the subpopulations of the diffusion algorithm, the applied selection schemes can be equal to those used in common evolutionary algorithms. Moreover, the genetic

representation, the variance operators, and the fitness function are not affected by the adoption of these population models. In so far, mating restrictions that arise due to the simulation of geographical separation merely represent a special addition to the parent selection process and the population management.

autonomy of the distance metric

Similar to the case of the fitness function, any selection process that takes into account the distance between two individuals needs not to be concerned about how this quantity is obtained. Likewise, the formulation of the fitness function is not at all affected by the presence of a similarity measure. Calculating the similarity between two individuals can therefore be seen as an additional component of the evolutionary algorithm that is independent of the fitness function and the selection scheme.

distance metrics and genetic codings

The distance metrics introduced by the last two approaches of the above list are either based on features of the individuals or on special tags that are encoded within their genomes. Hence in both cases, the similarity between two individuals can be inferred from their genotypes, and the reasonable choice of a distance metric is closely linked with the used genetic coding. Designing a genetic representation in connection with corresponding variance operators could thus be extended to also incorporate the definition of an adequate similarity measure. Instead of being considered as a new independent component, the similarity function can then be regarded as an additional aspect of the genetic representation.

preserved modularity

In summary, it is concluded that the discussed extensions give rise to additional demands on the implementations of the main components of an evolutionary algorithm. Apart from that, it persists that the detailed realization of each component is encapsulated from the remaining parts by simple and well-defined interfaces: The genetic representation has to provide adequate variation operators and possibly a reasonable distance measure. The fitness function has to assign a fitness value to each individual that quantifies its performance on the given task. On the basis of the fitness and the similarity between individuals, the selection scheme finally has to form mating pairs and select candidate solutions for the next generation. It is worth repeating that as long as all components meet their respective requirements, their detailed realizations remain unaffected by the implementations of the others. This important feature will turn out to be of considerable advantage for the design of flexible evolutionary algorithm software frameworks (see section 7.4).

3.4 Evolutionary Algorithm Implementations

This section introduces the most widespread implementations of the different evolutionary algorithm components. The only exception is the fitness function: Surveying common realizations of the latter would be infeasible and of only limited use as it is largely specific to the particular task at hand. Furthermore, the presented implementations of the genetic coding and the corresponding variation operators are confined to those suited for parameter optimization problems. Representations and operators that are typical for the field of genetic programming are deliberately excluded since this area is not directly related to the work presented in this thesis.

For an exhaustive overview of evolutionary algorithm implementations that also accounts for genetic programming see [50].

3.4.1 Selection Schemes

The differentiated survival of individuals based on their fitness is one of the cornerstones of evolutionary computation. Two basic models can be distinguished of how this competitive element of population management is realized. According to the generational approach, each iteration starts with a population of μ individuals from which a fixed number of parents is selected to form a mating pool. Note that depending on the applied selection scheme, this mating pool can contain multiple copies of the same individual. In any case, its members are used to create μ offspring that completely replace the old individuals and form the next generation. Implementations that follow such a complete replacement scheme are commonly denoted as generational evolutionary algorithms.

generational algorithms

In contrast, steady-state algorithms only change parts of the population in each step by substituting $\lambda < \mu$ individuals with a corresponding number of offspring. The fraction of the population that is replaced during one iteration is called the generational gap and is given by λ/μ.

steady-state algorithms

While both approaches are equally often encountered in classical genetic algorithms, the fields of genetic programming and evolution strategies usually follow the generational scheme and the area of evolutionary programming traditionally applies the steady-state model. Nevertheless, both strategies agree in that the corresponding parent and/or survivor selection procedures operate on the basis of an individual's fitness. In the following, it shall be assumed that the fitness increases with the quality of the candidate solution and is always nonnegative.

Fitness Proportional Selection

The selection scheme originally applied in classical genetic algorithms is the so-called fitness proportionate selection [71]. It is a probabilistic procedure in so far as each individual is assigned a nonzero probability to be chosen. Given the fitness f_i of the individual with index i, its selection probability p_i is computed according to

mechanism

$$p_i = \frac{f_i}{\sum_{j=1}^{\mu} f_j}, \quad \forall\, 1 \leq i \leq \mu \tag{3.1}$$

where μ is the population size. Thus, p_i depends on the absolute fitness value of the individual compared to the total cumulated fitness of the population. As required, the hereby defined probabilities readily obey

$$\sum_{i=1}^{\mu} p_i = 1. \tag{3.2}$$

Fitness proportional selection suffers from several problems. First, individuals with outstanding fitness tend to take over the whole population very quickly which

challenges

leads to premature convergence. Second, in later generations of an evolution, the fitness values of all population members usually have become large and are likely to be close together. Hence, there will only be marginal differences between the selection probabilities of the single individuals and thus hardly any selection pressure left. Third, the same phenomenon occurs when the fitness function is transposed, i.e., is modified by the addition of a sufficiently large value. The last two issues are closely related and are consequences of the fact that the selection probabilities are calculated on the basis of the absolute fitness values.

Ranking Selection

The drawbacks of fitness proportionate selection can be overcome if the selection probability of an individual is not based on the absolute value of its fitness but rather on its rank in the current generation [10] [77] [221]. Let the μ individuals in the population be ordered such that the one with the the highest fitness is assigned rank μ and the one with the lowest fitness is allocated to rank 1. In linear ranking, the selection probability is a linear function of the rank i given by

linear ranking

$$p_i = \frac{1}{\mu}\left(s^- + (s^+ - s^-)\frac{i-1}{\mu-1}\right), \quad \forall\, 1 \leq i \leq \mu \tag{3.3}$$

where s^-/μ and s^+/μ are the respective probabilities for the worst and the best individual to be selected [21]. In order to fulfill the normalization condition 3.2, it is required that $s^- \geq 0$ and $s^+ = 2 - s^-$. This poses a rigid restriction on the selection strength, i.e., the difference in selection probability between the worst and the best individual.

exponential ranking

Exponential ranking selection [21] does not suffer from these limitations. Here, the selection probabilities are calculated on the basis of the rank i according to

$$p_i = \frac{c^{\mu-i}}{\sum_{j=1}^{\mu} c^{\mu-j}}, \quad \forall\, 1 \leq i \leq \mu \tag{3.4}$$

using the adjustable parameter $c \in (0, 1)$. If c approaches 1, the selection probabilities of the individuals all converge to a common value and the selective pressure weakens. For c close to 0, the exponential character of equation 3.4 becomes more pronounced and individuals with higher ranks are more and more favored over those at lower ranks.

truncation selection

A special form of ranking selection is the so-called truncation selection that fixes a threshold T and only assigns a nonzero selection probability to the best T individuals in the population. While worse individuals cannot be selected at all, the first T members usually stand the same chance $1/T$ to be chosen. The deterministic variant of this approach simply defines the best λ candidate solutions to represent the outcome of the selection procedure.

Tournament Selection

mechanism

Instead of calculating and sampling a probability distribution defined on the fitness values of the whole population, tournament selection solely relies on an ordering relation that can rank any two individuals. The selection is performed in two steps. First, τ members of the population are chosen in a completely random manner. This intermediate group is referred to as the tournament. Second, the individual with the highest fitness among the τ chosen competitors is taken to be the final selection. Hence, while the composition of the tournament is entirely random, the choice of the winner is deterministic.

adjustable selection pressure

In order to select a pool of λ individuals, λ independent tournament selections have to be performed. Although the effective selection probability for each individual depends on its rank, the tournament scheme can be realized without sorting the entire population. Furthermore, the selective pressure can conveniently be adjusted by varying the tournament size τ. With increasing τ, the selection probability for individuals with low fitness decreases.

The selection pressure also depends on whether the constituents of the tournament are chosen from the initial population with or without replacement, i.e., whether an individual can appear in the tournament more than once. Especially the worst individual of the population only exhibits a non-zero probability to win a tournament, if it actually has to compete solely with copies of itself.

It can be shown that a tournament selection with tournament size 2 is effectively equivalent to a linear ranking selection [21]. Since it is considerably simple and easy to implement — and due to the direct control of the selective pressure via the tournament size τ — this selection scheme is one of the most widely used for evolutionary algorithms.

Survivor Selection Specialties

In principle, the selection procedures discussed so far are all equally well suited for both, parent and survivor selection. Traditionally, however, they originate from the field of genetic algorithms where they are only applied to the parent selection process. It has been said before that it is common in this kind of algorithm to replace the whole initial generation by its offspring. In a steady-state algorithm, on the other hand, the number of offspring λ is lower than the size of the population μ. Those individuals that will have to make room for the offspring can be selected according to any of the above schemes if it is based not on the fitness or rank of an individual but rather on the respective inverse.

deterministic replacement

A popular choice for this kind of replacement selection is the deterministic variant of the truncation selection, i.e., the worst λ members of the old population are chosen to be substituted by the offspring. But even if the replacement of the old individuals is stochastic, it is most often desirable to prevent the best member of the former population from being discarded. It is therefore common to exclude the best individual from the replacement selection process. This is referred to as elitism and can be generalized to a preservation of the best n_{el} individuals.

elitism

An alternative realization of the survivor selection process is to apply one of the

introduced selection schemes unmodified but to the whole group of the μ parents *plus* the λ offspring.

3.4.2 Genetic Representations

codings for parameter optimization tasks

As stated at the beginning of this section, it will be assumed in the following that the task at hand is a parameter optimization problem. In general, the parameters can be binary (Boolean), nominal (i.e., they can assume one of a finite set of possible values), integer-valued, or continuous (e.g., real numbers). If the genes directly translate to characteristics of the phenotype, the coding is called a direct encoding. In an indirect encoding, the genes rather correspond to the parameters of some kind of building rule according to which a phenotype is constructed[4]. The two examples for the coding of neural network weights given in section 3.2.1 represent direct encoding schemes. Some indirect encoding methods for neural networks will be introduced in section 4.2.2.

representations on digital systems

Evolutionary algorithms are commonly implemented on digital computers. This suggests at least three basic ways in which parameters can conveniently be coded within a genome: In a binary representation, by integer values or in the form of floating point variables (which shall for simplicity be regarded as equivalent to real numbers). While in a digital system, all of the above representations are ultimately stored in binary form at the lowest level, the approaches rather differ in what is regarded as a complete gene, i.e., an indivisible basic constituent of the abstract genome (see section 3.2.1).

Binary Representation

The binary coding, besides obviously being an adequate choice for Boolean attributes, can also be used for nominal, integer-valued and quasi-continuous parameters. Historically, genetic algorithms have often incorporated a binary representation regardless of the investigated problem and the nature of its free variables. In its simplest variant, the single genes are arranged to form a linear bit string that represents the entire genome.

binary coded numerical attributes

When numbers are coded through binary n-bit integers, the single bits are assigned different significances. This gives rise to the well-known phenomenon that the probability to change a given value x to any other value y by a random flipping of bits is not necessarily equal for all possible y's. For example, changing a 15 (01111) into a 14 (01110) requires the modification of only one bit. But in order to yield a 16 (10000), it is in fact necessary to flip every single position. When evolving nominal, integer or real-valued parameters, this peculiarity of the canonical binary coding can cause an unwanted and disadvantageous bias to the variation operators, particularly mutation.

known problems

This issue can be circumvented by applying other, more appropriate binary coding schemes [50]. In general, however, it is not advisable to use a binary

[4]Note that by using an adequate indirect encoding, it can be possible to map specific problems to a simple parameter optimization task that would normally exhibit a more complicated combinatorial structure

representation to code nominal, integer-valued or real parameters at all. Instead, a corresponding integer or floating-point encoding should be chosen.

Integer and Floating Point Representations

The implementation of integer or floating-point representations is straight forward. The single genes of the genome are arranged to form a corresponding vector of integers or real numbers. As each value represents a complete gene, its integrity under crossover is warranted. Furthermore, mutation operators will treat the numbers as a whole and not just operate on the single bits of their representation at a lower level (see section 3.4.3).

The range of possible alleles will be limited in practice. In integer representations, a gene can usually assume one of a finite set of values $\{L, L+1, \ldots, U-1, U\}$ lying between a lower bound L and an upper bound U. Correspondingly, a floating-point gene is confined to a closed and connected interval $[L, U]$. Note that in the latter case, the number of possible allele values in principle remains infinite and is only limited by the precision of floating-point numbers on the used digital system.

3.4.3 Mutation Operators

uniform and nonuniform mutation

In the case of parameter optimization tasks, the most widespread mutation operators all affect individual genes separately. The old value of the gene is either replaced by a new one that is in no way correlated with the original, or it is modified by a small amount. The first procedure is called random resetting or uniform mutation, the second approach is sometimes denoted as nonuniform or creep mutation [50].

mutation rate

It is common to define a mutation probability ρ_{m} for each applied operator such that it is individually decided for each gene whether it is to be modified or not. In this scenario, the eventual number of genes that are changed is not decided in advance but depends on the generated random numbers. Alternatively, if n_{g} is the number of genes in the genome, $\rho_{\mathrm{m}} \cdot n_{\mathrm{g}}$ can be interpreted as the fixed number of genes to be mutated. An according set of genes is then picked randomly.

For most practical applications, the mutation rate is in the order of several percent and thus $\rho_{\mathrm{m}} \ll 1$. Therefore, the required number of random values for the second strategy is considerably lower than for the first approach. This can be advantageous when the generation of random numbers is costly and the size n_{g} of the genome is large.

Uniform Mutation

binary codings

In binary representations, random resetting assumes the form of simple bit-flipping, i.e., the affected gene changes its value from 1 to 0 or vice versa. This is the predominant mutation operator used in classical genetic algorithms with binary codings.

integer codings

For integer-coded genes, the original allele is replaced by a new value randomly chosen from the entire set of allowed numbers. The choice is uniform in that the possible values are each assigned the same probability to be selected. Thus,

since the number of permissible alleles is finite, the gene stands a non-zero chance to remain unchanged even if it is mutated. Especially for attributes with only few alternative allele values, this can have measurable influence on the effective mutation rate and it might be necessary to ensure that the new value is in fact different from the original.

real-valued codings

In the case of floating-point genes, the new value is chosen uniformly and randomly from the allowed interval $[L, U]$. Given the usual precisions of floating-point numbers on common digital systems, the probability to reproduce the original gene value is negligible.

Nonuniform Mutation

In contrast to random resetting, creep mutation does not generate a new random allele but rather adds a small value r to the original gene that is randomly sampled from a given nonuniform distribution $\rho(r)$. This distribution is generally symmetric around zero and more likely to yield small numbers than large ones. The most popular choice is a Gaussian or normal distribution with zero mean and a fixed standard deviation σ_r

Gaussian mutation

$$\rho_{\sigma_r}(r) = \frac{1}{\sqrt{2\pi}\sigma_r} \exp -\frac{1}{2}\left(\frac{r}{\sigma_r}\right)^2. \tag{3.5}$$

Using a Gaussian distribution, it is ensured that approximately two third of the random numbers lie within one standard deviation around zero. In addition to the mutation rate ρ_m, the average effect of mutation can thus be controlled by varying σ_r.

Although the Gaussian 3.5 is defined on the whole set $\mathbb{R}$, the results of the random changes are reasonably bound to lie between L and U. If the gene value after mutation exceeds the allowed range, it is common practice to set it to the respective boundary value L or U. For integer representations, the outcome of the mutation either has to be rounded accordingly or the normal distribution itself has to be discretized. Furthermore, it is evident that creep mutation cannot sensibly be applied to binary values.

Mutation Parameters

automatic parameter adaption

The algorithm parameters that control the effect of mutation — the probability ρ_m and the width σ_r — are not necessarily fixed globally. It is in fact a distinct feature of evolution strategies, that the standard deviation of the applied Gaussian mutation is often set individually for each gene. Furthermore, either the common value σ_r or, potentially, the individual widths σ_r^i are coded within the genomes and are subject to variation operators themselves. This allows to automatically adapt the mutation parameters during evolution [9] [50].

It has been argued that an adaption of the evolution parameters is generally to be favored above fixing them in advance [8] [43] [50] [91]. This claim is supported by the intuitive assumption that during the individual stages of the search process, i.e., exploration and exploitation, different choices for the mutation parameters are deemed to be optimal.

automatic adaption vs. fixed parameters

Preceeding investigations had focused on finding a fixed set of parameters for specific algorithms that are best suited for a given set of tasks [46] [75] [89]. Against the background of the No Free Lunch theorem (section 3.3.3), it is understood that no set of evolution parameters can be found which is ideal for all tasks. On the other hand, practical experience suggests that a more than satisfactory performance can often be obtained even without a thorough optimization of the mutation parameters or a sophisticated self-adaption process (see section 9.4.5).

3.4.4 Recombination Operators

One of the features that distinguishes evolutionary algorithms from other optimization procedures is the recombination of characteristics of two or more candidate solutions to form new ones. This mixing of information is achieved by applying appropriate recombination operators. For parameter optimization tasks, the genomes of the parents are each given by a set of values, i.e., a vector $\mathbf{g}$ of bits, integers or floating-point numbers g_i, $1 \leq i \leq n_\mathrm{g}$. The genomes of two individuals $\mathbf{g}^a$ and $\mathbf{g}^b$ are most commonly combined by swapping single components g_i among the respective vectors[5].

One-Point Crossover

The oldest and simplest form of an exchange operator is one-point crossover [46] [71] and the term crossover has in fact (illegitimately) become a widespread synonym for recombination. Given two parental gene vectors $\mathbf{g}^a$ and $\mathbf{g}^b$ with dimension n_g, one-point crossover starts by generating a uniformly distributed random number $r_\mathrm{c} \in \{0, \ldots, n_\mathrm{g} - 1\}$. It then defines the two offspring $\mathbf{h}^a$ and $\mathbf{h}^b$ according to

$$h_i^a = \begin{cases} g_i^a & i \leq r_\mathrm{c} \\ g_i^b & i > r_\mathrm{c} \end{cases} \quad \text{and} \quad h_i^b = \begin{cases} g_i^b & i \leq r_\mathrm{c} \\ g_i^a & i > r_\mathrm{c} \end{cases} \quad \forall\, 1 \leq i \leq n_\mathrm{g}. \tag{3.6}$$

In other words, the vectors are cut apart behind position r_c and the ends are exchanged between them. As seen in figure 3.4 a), this procedure results in two individuals that represent complementary combinations of the genes of their parents.

Multi-Point Crossover

One-point crossover can readily be generalized to n-point crossover where the original genomes are randomly partitioned into $n + 1$ parts. This requires the generation of n random cut points r_i, $1 \leq i \leq n$ with $r_{i-1} < r_i \; \forall\, 2 \leq i \leq n$. Offspring is created by taking alternating segments from the two parents. For the case $n = 2$, this is illustrated in figure 3.4 b). Again, two new individuals are created.

positional bias

n-point crossover (with the special case $n = 1$) has a tendency to preserve combinations of genes that lie close to each other in the genome. This is commonly

[5] While this approach is typical for genetic algorithms, evolution strategies often employ different forms of recombination [9] [50] or no recombination at all.

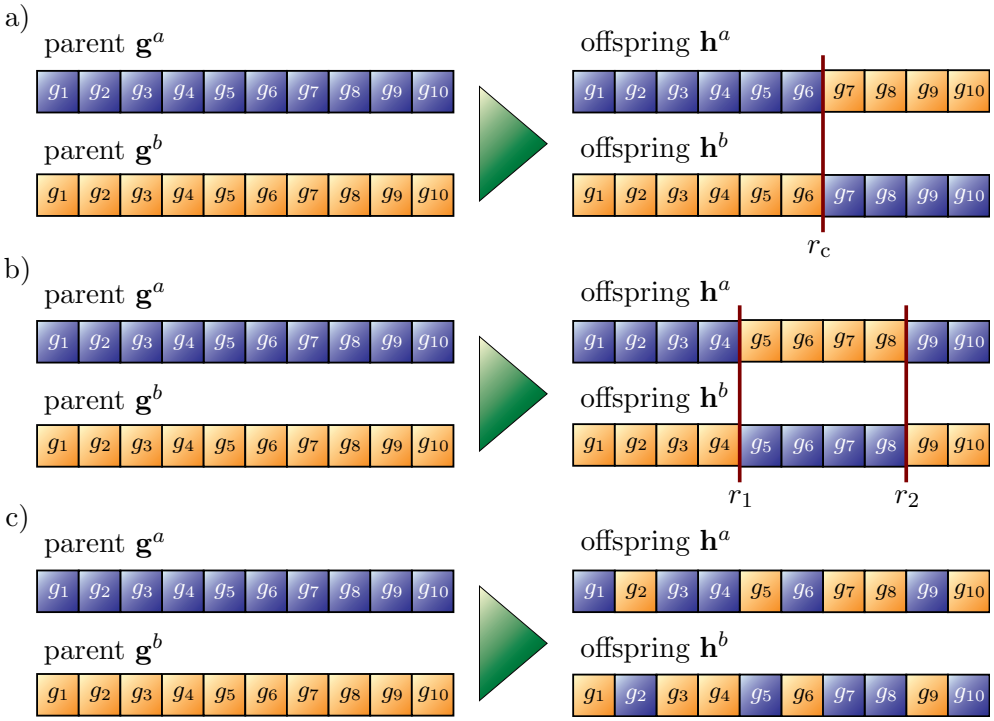


Figure 3.4: ***a)*** *The one-point crossover operator splits the two involved genomes at the random cut point r_c (6 in this example) and swaps the tails between them.* ***b)*** *Two-Point crossover generates two random cut points r_1 and r_2 (here, 4 and 8, respectively) and exchanges the enclosed segments.* ***c)*** *In uniform crossover, it is decided for each gene separately, from which parent it is taken.*

denoted as positional bias [51]. Whenever there are known dependencies between the optimized parameters, positional bias can in fact be exploited as to benefit the efficiency of the recombination process. If genes with strong interrelations are arranged next to each other, n-point crossover of two individuals with high fitness is likely to preserve their advantageous combinations of alleles and recombine them to yield improved candidate solutions.

practical implications

This comprehension lead to the formulation of the schema theorem discussed in section 3.5. Moreover, it constitutes a helpful directive for the design of adequate representations in practice: When using recombination operators with positional bias, the genes are best grouped within the genome according to their functional relationships.

Uniform Crossover

In uniform crossover, it is decided separately for each gene from which parent it is chosen, usually with equal probabilities. A second offspring can then be created by using the inverse combination of genes as shown in figure 3.4 c). Within the field of evolution strategies, this operator is also referred to as discrete recombination.

distributional bias

The uniform crossover operator does not exhibit any positional bias. On the other hand, it tends to yield offspring that contains the genetic material of both parents in an approximately equal amount. Such a behavior is known as distributional bias [51].

selecting adequate crossover schemes

Bearing in mind the propositions of the No Free Lunch theorem brought forward in section 3.3.3, it can be concluded that if no knowledge is available about the dependencies between the optimized parameters — and their arrangement within the genome is thus random — there is no *a priori* reason to favor any of the above crossover operators over the others. Empirical and theoretical studies suggest that uniform crossover is superior to the one or two-point operators on several artificial problems [197] [203]. But it should not be underestimated that the number of necessary random values is far greater in the uniform case than for n-point crossover with moderate n. Furthermore, when the genes can be grouped as to account for relationships between the parameters, recombination operators with positional bias might generally be the better choice.

A Note on the Organization of the Genome

partitioning into chromosomes

So far, it has been assumed that all genes of a candidate solution are arranged within one single vector that constitutes the entire genome of the individual. Like in biological systems, the genetic material of a genome can be partitioned into multiple chromosomes, each being a string or vector of genes as considered above.

Any Recombination operators are then applied to each parental pair of chromosomes separately. As in the case of operators with positional bias, this scheme is considered useful if the distribution of genes between the chromosomes reflects the organizational characteristics of the optimized system.

swapping chromosomes

Moreover, a segmentation into chromosomes suggests a new form of crossover where whole chromosomes are swapped between two mating individuals with a given probability. Whether they are exchanged or not is individually decided for each chromosomal pair. Finally, the actually applied recombination operator can be different for each chromosome. The respective choice might depend on the presence or absence of functional relationships between the contained genes.

Another Note on Mixed Representations

Returning to the subject of genetic representations discussed in section 3.4.2, it is to be expected that a given optimization task incorporates a variety of parameters of different kinds. Rather than trying to find a common representation to be applied to all attributes, it seems reasonable to code each parameter in the way that best suits its characteristics. The resulting genomes will contain genes of different types and each gene can assume alleles g_i of its individual set G_i, $g_i \in G_i$.

mixed representations and variation

As long as all genomes in a population are structurally equivalent, a mixed representation does not necessarily give rise to further complications. The presented mutation operators all operate on single genes. Like in case of the used encoding, the applied mutation operator can readily be chosen for each gene separately. Furthermore, all discussed recombinational models are independent of the

specific nature of the single genes. Hence, all of the above exchange operators can immediately be applied to any mixed representation without further modification.

3.5 Theoretical Analysis: The Schema Theorem

objectives

A considerable amount of theoretical work has been undertaken to model and analyze the behavior of evolutionary algorithms. Ultimately, these efforts aim for a prediction of the performance of a given algorithm on a specific task. If successful, this would allow for the selection of the most suitable algorithm — and the most adequate choice of parameters — for a desired optimization problem in advance.

methodologic variety

A variety of theoretical techniques have been employed in pursuit of this goal such as, e.g., the analysis with Markov Chains [173] [174], the dynamical systems approach [217] [214], and the application of methods that originate from statistical mechanics [164] [165]. Although these investigations provide valuable insights into several different aspects of how evolutionary algorithms work, they are in the majority of cases concerned with comparably simple binary genetic algorithms on simplified artificial problems.

limitations

Evolutionary algorithms are complex dynamic systems involving numerous random factors and it can be argued that the ultimate goal of precisely predicting the behavior of an arbitrary algorithm on a realistic task might not ever be achievable [50]. Nevertheless, the reported results at least provide useful hints for the construction and tuning of efficient evolutionary algorithms in practice.

the schema theorem

A thorough examination of these various theoretical approaches and their results cannot be given here. Instead, the following sections will confine themselves with the discussion of an early result found by Holland which is commonly known as the schema theorem [96]. Holland's analysis has not only been of vital importance for the development of genetic algorithms in general but will also prove to be useful when investigating the feasibility of evolutionary algorithms for the training of neural networks (chapter 4).

3.5.1 Schemata

definition

A schema is simply an affine subspace in the search space. The agreed notation for a schema uses the "don't care" symbol #, such that in a 5-dimensional genotype space $\mathbb{G}$ consisting of genome vectors $\mathbf{g} = (g_1, g_2, g_3, g_4, g_5)$, the exemplary schema $H = (1, \#, \#, \#, 1)$ denotes the affine subspace in $\mathbb{G}$ which is defined by all points that exhibit ones in the first and last component. All vectors that meet this requirement are called examples or instances of this schema. This allows to define the fitness of a schema to be the mean fitness of all its instances.

defining length and order

Two measures characterize a schema. First, its order is defined as the number of its specified positions, i.e., those positions that do not show the # sign. Second, the defining length is taken to be the number of crossover points between the outermost fixed genes. The schema defined above has order 2 and defining length 4, whereas the schema $(1, \#, 0, \#, 1)$ also has defining length 4 but exhibits an order of 3.

the standard genetic algorithm

Holland's initial analysis is concerned with the so-called standard genetic algorithm (SGA) that uses a binary representation, fitness proportional parent selection, one-point crossover, a single-gene bit-flipping mutation operator, and a generational replacement scheme. In this case, a genome $\mathbf{g}$ is given by a simple binary string and the above schema — which is now conveniently written as $1\#\#\#1$ — includes $2^3 = 8$ instances.

3.5.2 The Processing of Schemata

implicit parallelism

Holland showed that a given string of length n_g is an example of 2^{n_g} schemata. Although a finite population of μ genomes will not in general contain $\mu \cdot 2^{n_\mathrm{g}}$ different schemata, he derived that it can nevertheless usefully process about $\mathcal{O}(\mu^3)$ of them. This feature of genetic algorithms is known as implicit parallelism and has widely been regarded as one of the main reasons for their efficiency.

schemata and variation operators

During the execution of an evolutionary algorithm, the number of instances of a schema that are present in the current population will depend on the schemata's fitness value as well as on the effects of the applied variation operators. Recombination and mutation can create new instances but can also destroy previously existing examples.

mathematical formulation

In the case of the SGA, the probability of a schema H to be disrupted by crossover and mutation can be calculated and expressed in terms of its order $o(H)$, its defining length $d(H)$, and the length of the genome n_g. Let $f(H)$ denote the effective fitness of the schema, i.e., the average fitness of all its instances in the current generation. If $<f>$ is taken to be the mean fitness of all present individuals, then the on average expected number of instances in the next generation $\overline{m}(H, t+1)$ can be estimated from the number of examples $m(H,t)$ in the current population according to

$$\overline{m}(H,t+1) \geq m(H,t) \cdot \frac{f(H)}{<f>} \cdot \left[1 - \left(p_c \cdot \frac{d(H)}{n_\mathrm{g} - 1}\right)\right] \cdot [p_m \cdot o(H)] \qquad (3.7)$$

where p_m and p_c are the respective probabilities for applying the mutation and crossover operators [96]. The common interpretation of this result is that the representations of schemata which exhibit a sufficiently high fitness and also stand a reasonable chance to be preserved under mutation and crossover will continually grow from generation to generation.

Schemata and Variation Operators

destructive and constructive effects

The fact that equation 3.7 can merely estimate a lower bound on the number of instances of a given schema H is ultimately owed to the circumstance that Holland's derivation does not account for the constructive effect of the variation operators. Only the probabilities of crossover and mutation to destroy a given example of a schema are considered. Analyzing the constructive potential of the variation operators is considerably more difficult, since these effects depend on the explicit composition of the current population. But it has later been shown that under some simplifying conditions and for an arbitrary recombination operator,

the expected number of instances of a schema that are destroyed are in fact equal to the expected number of examples that are created [198].

positional and distributional bias

Besides that, the evaluation of variation operators in terms of their destructive effect on schemata provides a more solid foundation for the concepts of positional and distributional bias (see section 3.4.4) [51]. Regarding two schemata H_1 and H_2 with equal fitness $f(H_1) = f(H_2)$ but different defining lengths $d(H_1) < d(H_2)$, an operator is denoted to exhibit positional bias if it is less likely to destroy the schema with the shorter defining length H_1. This, e.g., applies to one-point crossover.

On the other hand, a variation operator is said to show distributional bias if its probability to disrupt a schema H is a function of the schema's order $o(H)$. Uniform crossover as well as every single-gene mutation operator fall into this category.

3.5.3 Building Blocks, Deception and Challenges to the Schema Theorem

estimated growth of representations

A closer investigation of equation 3.7 reveals that assuming equal fitness values, short low-order schemata generally have a higher probability to be preserved in the next generation of an SGA than longer schemata or those with higher order. It has been argued that by virtue of equation 3.7, the representations of short schemata with low order are in fact expected to grow approximately exponentially.

the building block hypothesis

This comprehension forms the basis of what has become widely known as the building block hypothesis [71]. According to this model, a genetic algorithm starts by selecting short, low-order schemata and successively combines them to build longer, higher-order schemata until, in the ideal case, a schema of order n_g is found that represents the globally optimal solution. The building block hypothesis stresses the role of recombination and in the field of genetic algorithms, crossover is regarded to be the primary force behind the optimization process, whereas mutation is merely regarded as a background mechanism that is to warrant the necessary genetic diversity [71].

Deceptive Problems

characterization

Analyzing the operation of genetic algorithms in terms of building blocks eventually gives rise to the question of what is expected to happen once the global solution to a given problem is not an instance of the low-order schemata that exhibit a high mean fitness. To illustrate this, consider an exemplary three-bit problem where the fitness values of the possible bit-strings are given as follows [225]:

example problem

$$\begin{aligned} f(000) &= 28 & f(001) &= 26 \\ f(010) &= 22 & f(100) &= 14 \\ f(110) &= 0 & f(011) &= 0 \\ f(101) &= 0 & f(111) &= 30 \end{aligned}$$

In this artificial problem, all schemata of order $n < 3$ that have a 1 in one

of their defining positions exhibit a lower mean fitness than their corresponding counterparts that show a 0 in the same position. More specifically, the averaged fitness values $f(h)$ of the different schemata H obey:

$$f(0\#\#) > f(1\#\#) \qquad f(00\#) > f(11\#),\ f(10\#),\ f(01\#)$$
$$f(\#0\#) > f(\#1\#) \qquad f(0\#0) > f(1\#1),\ f(1\#0),\ f(0\#1)$$
$$f(\#\#0) > f(\#\#1) \qquad f(\#00) > f(\#11),\ f(\#10),\ f(\#01)$$

Yet, the global optimum is represented by the string 111. In other words, all schemata of order $n < 3$ effectively lead the search away from the perfect solution and towards the suboptimal string 000. Problems of this kind are commonly denoted as deceptive, although this term is not unambiguously defined [76] [195] [225].

practical relevance

It could be apprehended that such deceptive problems considerably challenge conventional genetic algorithms since no appropriate building blocks are present. However, experimental investigations have been reported that do not reveal a direct correlation between the probability of an optimization problem to be deceptive and the ability of the SGA to find the global solution [195]. Also, it is not generally agreed whether deceptive problems are to be regarded as adequate models of realistic tasks in practice [76] [225].

Challenges to the Schema Theorem

It shall not remain unmentioned that the Schema Theorem and the building block hypothesis have faced some criticism. As stated above, the schema theorem is limited in so far as it does not consider the potential of the variation operators to create new instances of a schema but only accounts for their destructive effects.

finite population effects

Furthermore, the estimated fitness $f(H)$ of a schema is exclusively based on the fitness values of those instances that are present in the current population and might therefore not be representative of the schema as a whole. In so far, equation 3.7 is restricted to make predictions only for the following generation. Since the representation of H and all other present schemata will have changed in the next generation, so will the estimated fitness values, and the composition of the current population cannot reliably be employed to yield sound estimates for later generations [50]. In particular, the rate by which the representation of a given schema H increases is not exponential: As soon as its share of the population grows, its selective advantage $f(H)/ <f>$ decreases accordingly due to the increased mean fitness of the whole population.

estimated growth of representations

experimental counter-arguments

The experimentally verified efficiency of operators with a high distributional bias, most notably uniform crossover [197] [203], as well as the SGA's apparent ability to tackle deceptive problems [195] cannot satisfactorily be reconciled with the propositions of the schema theorem and the building block hypothesis. Considering the recombination of building blocks to be the main force behind genetic search can only insufficiently explain the success of those approaches that do not incorporate recombination at all, e.g., most forms of evolution strategies.

Nevertheless, it shall be repeated that analyzing evolutionary algorithms in terms of schemata persists to provide valuable insights into at least some aspects

of how this kind of optimization procedures work. Among other things, the discussed results considerably affect the applicability of evolutionary algorithms to the training of neural networks. This will be the topic of the following chapter.

Chapter 4

Evolving Artificial Neural Networks

Human beings, who are almost unique in having the ability to learn from the experience of others, are also remarkable for their apparent disinclination to do so.

Douglas Adams, Last Chance to See

Evolutionary algorithms have the ability to cope with complex, multimodal search spaces and do not depend on a detailed model of the system to be optimized (chapter 3). In particular, they do not require the applied performance measure to be differentiable with respect to the free parameters. A such, they represent a promising approach to train neural networks, even those kinds of neural systems for which no other feasible training algorithm is immediately available, e.g., recurrent networks, multi-layer networks of binary threshold neurons or networks implemented in analog VLSI (see section 2.4).

possible combinations

Several examples in the preceeding chapter have already hinted at some suitable approaches for optimizing the synaptic weights of a network, but besides that, evolutionary algorithms have been combined with artificial neural networks also in various other ways (for a comprehensive overview see, e.g., [26] and [234]). Most notably, they can be employed to optimize the architecture of a network, i.e., the number of neurons and the available connections. Often, the training of the weight values and the construction of the architecture are combined. Further applications of evolutionary algorithms within the field of neural computation include the evolutionary tuning of learning rules [13] [36] or the optimization of neuron transfer functions [234].

This chapter will focus on the evolution of the synaptic weights and briefly discuss the evolutionary design of architectures. Since this thesis is primarily concerned with training strategies for hardware-implemented, feedforward networks that cannot be trained via conventional algorithms and which exhibit fixed neuron transfer functions (see chapter 5), the evolution of learning rules and activation functions will not be covered.

Instead, two alternative methods for the optimization of neural network weights

shall be discussed that do not depend on a differentiable performance measure and a detailed model of the network operation as well: Simulated annealing and the weight perturbation algorithm.

4.1 Evolving Synaptic Weights

If the topology of a neural network is fixed, training its synaptic weights can be regarded as an optimization process that aims to minimize the network's error on the training patterns (see sections 1.2.4, 2.1.2, 2.1.3, and 2.2.2). In so far, evolutionary algorithms could be expected to be readily applicable to this kind of task without major modifications.

benefits

Indeed, compared to most gradient-based training approaches like, e.g., the backpropagation algorithm discussed in section 2.2.2, evolutionary algorithms yield the advantage of not posing any restrictions on the architecture of the optimized network. Given a suitable fitness function that appropriately quantifies the performance of the network on the task in question (section 4.1.1), the same evolutionary algorithm can in principle be applied to feedforward networks, radial basis function networks, or recurrent networks [74]. In the following sections, emphasis is placed on feedforward networks, but most of the results can readily be transferred to different architectures.

challenges

On the other hand, it turns out that the structural features of neural networks call for special precautions during the design of suitable genetic representations and corresponding variation operators. These issues are closely related to what is commonly known as the permutation problem which will be addressed in section 4.1.2

4.1.1 Performance Evaluation and Fitness Function

network error

Equations 2.15 and 2.16 represent suitable means for quantifying the performance of a network on a given task and it seems reasonable to utilize them for evolutionary algorithms as well. If the fitness of an individual is desired to increase with improving performance, the network error E on the training patterns can easily be transformed into an adequate fitness function that meets this requirement. Common approaches are to choose $1/E$, $1/(1+E)$, or $(E_{max}-E)/E_{max}$ as the used fitness measure.

additional constraints

Apart from that, evolutionary algorithms allow to incorporate additional demands to the trained network into the formulation of the fitness function. In the case of highly recurrent architectures, for example, the fitness function could be formulated as to penalize networks that exhibit an undesirable oscillating behavior (see sections 2.3.3 and 2.3.4).

Although there are no direct limitations concerning the special requirements that might be accounted for by the fitness measure, it is to be remembered that the exact form of the fitness function can have considerable impact on the success of the evolutionary search (see section 3.2.1). More elaborate forms of performance measures might run a greater risk of assuming multiple local optima on

the accessible search space and thus promoting the premature convergence to a suboptimal solution. An example will be discussed in section 8.3.3.

4.1.2 Representations and the Permutation Problem

binary and real-valued codings

It is popular to either code the weights of a neural network in binary form and arrange them to a linear string [33] [221] [223] — such that each bit position is regarded as a single gene — or to represent them by a vector of real values [58] [145]. Traditionally, the former approach is mainly used in connection with classical genetic algorithms that primarily rely on recombination while the latter encoding is most often combined with evolution strategies where mutation is the only form of variation operator [58] (see also chapter 3). In principle, the real-number representation can be used with appropriate crossover operators as well [145] [206].

Ordering the Weight Values within the Genome

It has already been stated in section 3.4.4 that recombination operators with positional bias are expected to work best if the genotypic representations are arranged such that subsets of genes which code related features of the individuals are grouped close to each other in the genome. This intuitive reasoning receives theoretical support by the schema theorem which in turn constitutes the basis of the building block hypothesis (see sections 3.5.2 and 3.5.3).

grouping related weights

Transferred to the evolutionary optimization of neural networks, the implications of the building block hypothesis motivate to place the weights of functionally related synapses near to each other. But the distributed processing of information within a neural network makes it hard to identify these groups of related synaptic connections in advance — at least in the case of realistic tasks where the structural features of the final solution are not known. Nevertheless, a commonly proposed heuristic is to group together the weights of all connections that lead to one single neuron of the network [26] [206] [234].

The Permutation Problem

structural symmetries

Another property that arises as a direct consequence of the organizational principles of neural networks and which considerably impedes their optimization by common recombination-oriented evolutionary algorithms is their inherent symmetry. Figure 4.1 a) shows two functionally equivalent but structurally different neural networks and their exemplary genotypic representations. The networks differ solely in the labeling of their hidden neurons and the corresponding ordering of synaptic weights within their genomes. For a network with n hidden nodes, $n!$ functionally identical networks can be constructed simply by permuting the inner neurons.

On the genotypic level, the two networks of figure 4.1 a) are significantly different. In the general case when there are no degeneracies between the two sets of weights a_i and b_i, $1 \leq i \leq 3$, the genomes differ in each single gene position. The resulting phenotypes, however, are equivalent in so far as the corresponding

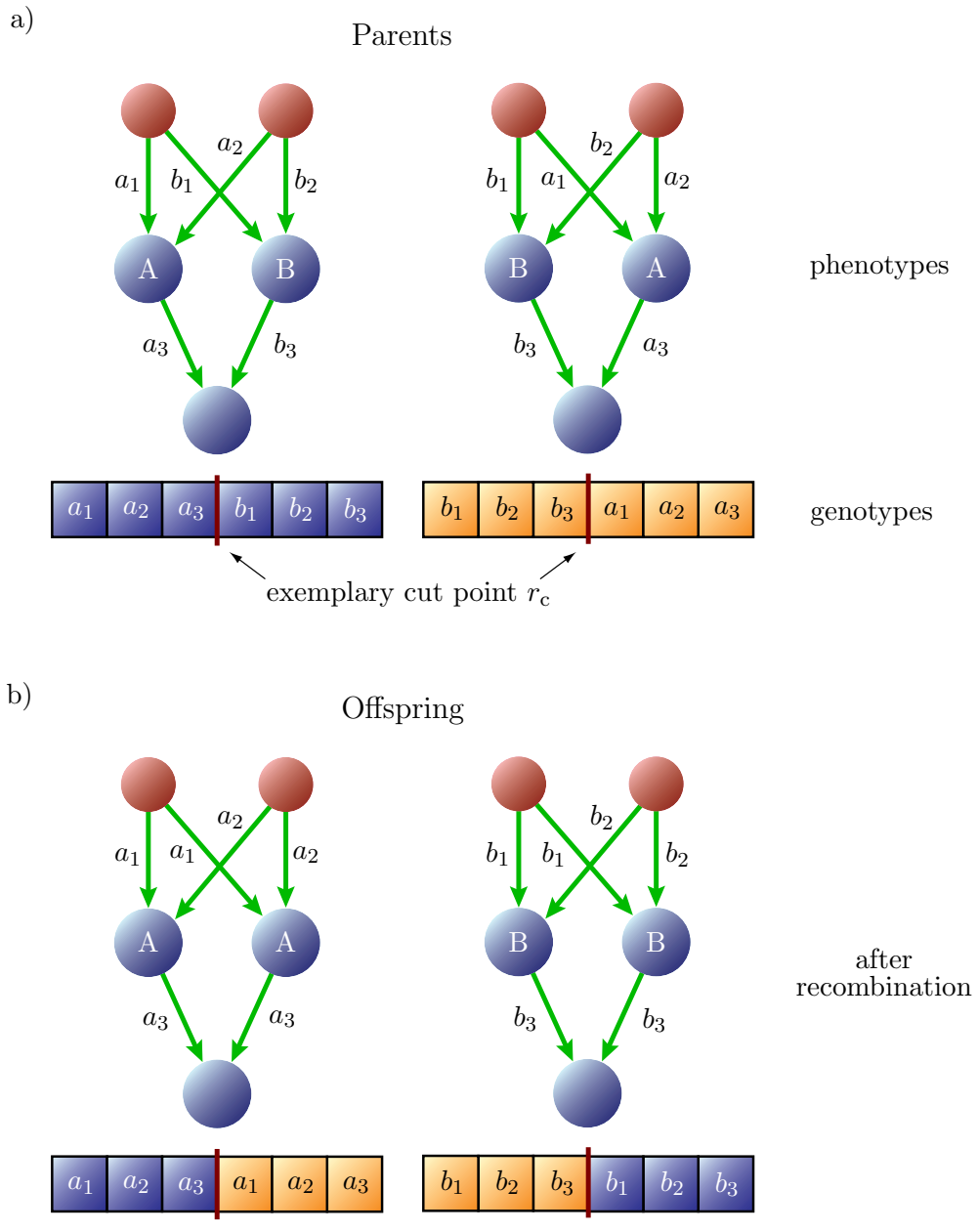


Figure 4.1: a) *Two functionally equivalent individuals can have different genetic representations. The shown networks differ solely in the labeling of their hidden neurons, but as long as there are no degeneracies between the two sets of weights a_i and b_i, $1 \leq i \leq 3$, the corresponding genomes differ in each single gene position.* **b)** *When the two genotypes are recombined using, e.g., a common one-point crossover operator, the offspring runs a high risk of being inappropriate. In the shown example, each child turns out to contain two identical hidden neurons. Assuming that the high fitness of the parents has been based on the appropriate combination of the different nodes A and B, the offspring is likely to yield a worse performance than the original networks.*

networks yield equal responses when being applied the same input, i.e., they implement the same functional mapping. In fact, with the numbering of the hidden nodes being initially arbitrary, one would usually regard the two individuals as one and the same network.

symmetries and recombination

Using the genetic representation shown in figure 4.1 a), the evolutionary algorithm is bound to ignore this symmetry. As illustrated in figure 4.1 b) a recombination of the genomes that involves a common one-point crossover operator is likely to produce inappropriate offspring. This phenomenon is commonly known as the permutation problem (also: competing conventions problem) [17] [80] [81].

weight symmetries

There is a related but somewhat more subtle issue that is connected to the special symmetry of neurons with odd activation functions [26]: A neuron of this kind retains its effective influence on the receiving nodes when the signs of the weights of all its incoming and outgoing connections are flipped simultaneously. For the types of activation function presented in section 1.2.3, this requires an additional adjustment of the receiving neurons' bias values, but the general symmetry abides. Within a network of n hidden neurons, one is free to flip the signs of any of these nodes which gives rise to $\sum_{i=0}^{n} \binom{n}{i} = 2^n$ different, but functionally identical networks.

Implications of the Permutation Problem

speciation

Each of the different conventions — i.e., the individual numbering of the neurons in connection with the agreed signs of the weights — is represented by a distinct region of the search space. Only a recombination of individuals from the same region promises to yield improved offspring. Therefore, it is reasonable to assume that the impact of the permutation problem could be alleviated by employing one of the speciation mechanisms introduced in section 3.3.4 (e.g., [199] [200] or see [26]).

schemata

In the light of the schema theorem, it is a direct consequence of the permutation problem that there exist multiple competing schemata with high average fitness that each specify the values of the same genes but exhibit different alleles on these positions. When multiple such building blocks are recombined, they do not automatically yield better individuals. On the contrary, given the high degree of symmetry in large networks, a combination of these building blocks stands a considerable chance to produce worse offspring. Seen from this angle, the training of neural networks can be regarded as a deceptive optimization task in terms of section 3.5.3. The simplest way to avoid the negative impact of the permutation problem would thus be to use no recombination at all.

population sizing and selective pressure

It has been argued that the effects of the permutation problem are not as severe as widely supposed [80] [81] and that they will most probably depend on parameters like the sizes of the network and the population [80] [81] [224]. Small population sizes in combination with strong selection and more aggressive mutation should prevent the algorithm from exploring multiple alternative but equivalent regions of the search space in parallel. On the other hand, this strategy runs the risk of promoting premature convergence and can be expected to be feasible only for small networks.

improving recombination

Several approaches have been proposed that try to match the hidden neurons of two networks and either appropriately reorder the genotypes before recombination or confine themselves to merely swapping corresponding hidden nodes between the two mating individuals (for an overview see [26]). Finally, the permutation problem can be circumvented by evolving the weights and the architecture of the network simultaneously (section 4.2).

4.1.3 Comparison with Gradient Based Training

speed considerations

Evolutionary algorithms are comparably expensive in terms of computational effort. In each generation, multiple new networks have to be implemented, tested and evaluated. Compared to fast variants of backpropagation or other gradient-based training algorithms, an evolutionary optimization of the weight values is often found to be measurably slower (see [234]).

Nevertheless, some reported results suggest that evolutionary training can be significantly faster than backpropagation at least on some problems (e.g., [188]). The apparent discrepancy between these observations can partly be attributed to the different evolutionary algorithm and backpropagation implementations compared.

Evolutionary Training of Hardware Neural Networks

optimizational capabilities

As already indicated above, the rationale behind the application of evolutionary algorithms to neural network training is not initially given by speed considerations. Rather, evolutionary algorithms lend themselves to the optimization of synaptic weights primarily due to their ability to find global optima in complex, multimodal search spaces and their insensitiveness to starting conditions. Furthermore, they can successfully and reliably train even those kinds of neural networks that cannot be optimized by common gradient-based approaches.

In this respect, evolutionary algorithms do not directly compete with traditional learning algorithms in terms of speed, but are to be seen to complement these approaches, especially considering those types of networks that cannot immediately be trained via common gradient-based optimization. Besides highly recurrent networks, this particularly applies to neural networks implemented in analog VLSI (see section 2.4).

training speed

For the latter case, the evolutionary approach is deemed to be feasible also with regard to the absolute speed of the training, since a parallel hardware realization potentially allows to evaluate multiple different networks in a short time. When being realized as chip-in-the-loop algorithms, evolutionary strategies therefore represent a competitive training approach for fast analog or hybrid hardware neural networks (section 2.4.5).

Combined Approaches

combining evolution and gradient descent

Several strategies have been reported in literature that aim to combine the principles of simulated evolution with gradient-based training (e.g. [17], or see [26]

and [234] for an overview). The common approach is to let the evolutionary algorithm identify the region of the global optimum in the search space and let a gradient descent algorithm perform the final fine-tuning of the synaptic weights. This procedure is motivated by the observation that most evolutionary algorithms perform well during the exploration phase but are comparably inefficient during the exploitation phase (see section 3.3.1).

efficiency

The reported results show that a hybrid algorithm of this kind can perform significantly better than either the evolutionary algorithm or backpropagation alone [17]. In practice, a conventional gradient-based training algorithm has to be restarted several times from different initial conditions in order to make up for its tendency to get stuck in local optima. A hybrid strategy that circumvents this problem by employing an evolutionary algorithm for the search of suitable initial weight values can thus be competitive also in terms of training speed.

4.2 Evolving Network Architectures

motivation

It is evident from what has been said in chapter 2 that the architecture of a neural network has significant impact on its representational capabilities. While a one-layer perceptron might not suffice to perform the task at hand, a multilayer network with too many inner neurons runs the risk of overfitting noise in the training data and may thus exhibit a poor generalization ability (see sections 2.1.1 and 2.2.1).

Although two-layer feedforward networks are known to be universal approximators, it is not clear in how far alternative architectures — with additional layers and/or shortcut connections — could solve the same tasks more efficiently, e.g., with fewer neurons (see section 2.2.1). Neither is it possible to determine the ideal structure for the investigated problem in advance, nor can a given architecture be proven to be optimal [26].

heuristic approaches

Several non-evolutionary heuristics have been proposed that extend common gradient-based approaches by an automated construction of the neural network architecture during training [62] [139] [157] [156], but these approaches still limit the range of allowed topologies to strictly layered feedforward networks. Evolutionary algorithms, on the other hand, promise to be a feasible approach for systematically searching through large regions of the architecture space without being restricted to a limited subset of valid structures.

4.2.1 Performance Evaluation - Architectures and Weights

evolving pure architectures

When evolving the optimal network topology for a given task, two different strategies can be distinguished. In the first case, the evolutionary algorithm is solely concerned about the structure of the network while the weights are to be trained separately [141] [224]. The fitness of an individual, i.e., a specific network architecture, is then commonly evaluated by training multiple networks of this topology and averaging their resulting fitness values. The single instantiations of the evaluated network structure can be trained by any suitable algorithm, e.g., backpropagation [141]. This kind of fitness determination is not only computationally

expensive but also retains a considerable uncertainty, since the number of trained networks per architecture is ultimately limited. Essentially, this problem can be attributed to the inherent one-to-many mapping from the genotype to the possible phenotypes.

evolving architectures and weights

The second approach therefore aims to evolve the architecture and the weights of the network simultaneously [199] [200] [236]. Each genome translates into a completely and uniquely defined network, and the resulting performance directly determines the fitness of the individual. This strategy is attractive in so far as it produces complete and immediately usable neural systems and can at the same time freely explore the space of all possible networks, not just those of a fixed architecture. On the other hand, the sheer size and complexity of the search space poses a considerable challenge to the genetic coding and the variation operators of the used evolutionary algorithm.

optimizing complexity

In either case, the fitness function can be formulated as to include a measure for the complexity of the individual network topologies such that the evolution favors simple and compact networks [224] [234]. This way, it can be ensured that the resulting structure is not only optimal with regard to its suitability for the given task, but also that the corresponding networks can be realized most efficiently. The latter feature is of particular interest if the networks are implemented in software on sequential processors.

4.2.2 Genetic Representations

Genetically representing the architecture of a neural network is not as straightforward as in the case of the mere weight values. It has to be ensured that the used genetic representation can encode an optimal or at least near-optimal solution. At the same time, it might be desirable to exclude invalid or meaningless network architectures, and it is to be investigated to what extent the applied variation operators are likely to yield valid offspring. Regardless of whether only the topology of the network is evolved or also its weight values, the reported representations either follow a direct or an indirect encoding scheme.

Direct Encoding Schemes

connectivity matrix

In a direct representation scheme, each connection within the network is specified explicitly [141] [177] [224]. The simplest way to encode the architecture of a network with N nodes is a connectivity matrix $C = (c_{ij})_{N \times N}$, such that $c_{ij} \in \{0, 1\}$, $1 \leq i, j \leq N$ specifies the presence or absence of a connection leading from node i to node j. An example is given in figure 4.2. A value of $c_{ij} = 1$ indicates an existing connection while $c_{ij} = 0$ marks the respective connection as being absent. The number of neurons as well as the functionality of each node, i.e., whether it is a network input, a hidden node, or an output neuron, are fixed in advance. Further constraints to the topology can be incorporated by forcing the diagonal elements of C to be zero — which prevents neurons from being connected to themselves — and/or ignoring nonzero entries of the lower left triangle, in which case

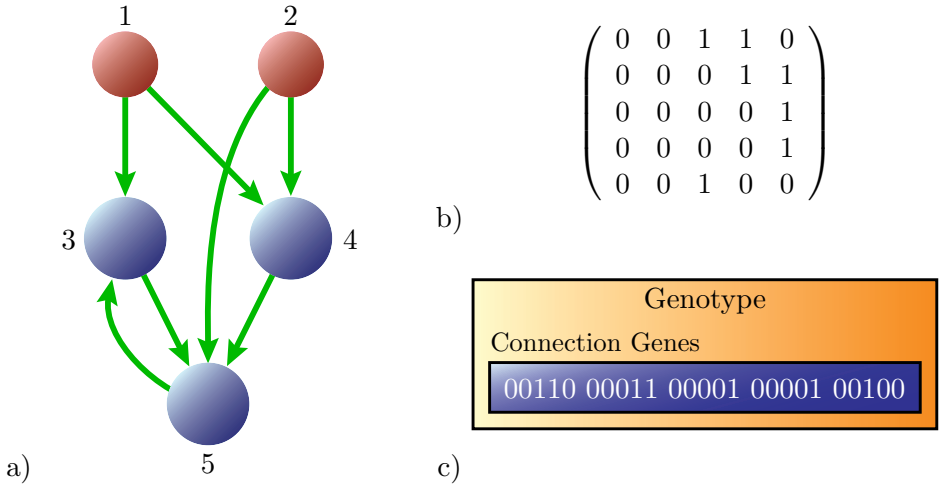


Figure 4.2: *Given a well-defined labeling of the single nodes (a), the architecture of a neural network can unambiguously be specified by a corresponding connection matrix $C = (c_{ij})_{N \times N}$, such that $c_{ij} \in \{0, 1\}$, $1 \leq i, j \leq N$ marks the presence or absence of a connection leading from node i to node j (b). The rows of this matrix can be arranged to form a linear genotypic representation (c) that is compatible with common genetic operators. This representation can easily be extended to also code the values of the present connections.*

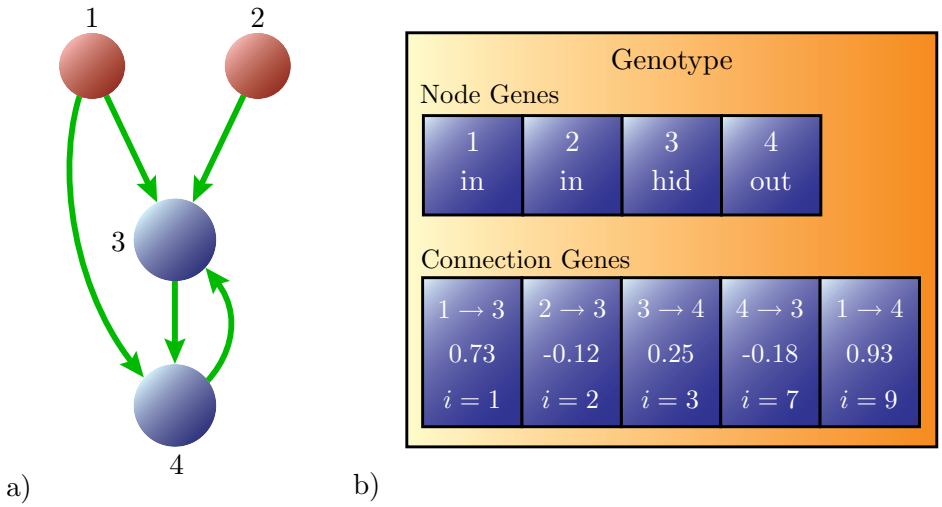


Figure 4.3: *The genetic representation of the NEAT algorithm [199] [200] codes the architecture and weights of a given network a) via two different sets of genes b): node genes and connection genes. Each node gene codes the label of a respective node of the network and also specifies whether it is an input, a hidden neuron or an output. A connection gene includes the source, target neuron, and weight value of its coded connection. Furthermore, each connection gene exhibits a unique so-called innovation number that allows to match corresponding genes during recombination [199] [200].*

the architecture is restricted to be feedforward[1].

genetic representation

The rows of the connectivity matrix are usually concatenated to form a linear genome (figure 4.2 c)) which allows for the application of common mutation and recombination operators. This encoding can easily be extended to evolve the architecture and the weights of the network simultaneously, in which case $c_{ij} \in \mathbb{R}$ is used to code the weight value of the respective synaptic link [24].

benefits and drawbacks

This simple kind of direct encoding scheme is particularly easy to implement, and investigations reveal that it has the potential to yield networks with better generalization ability than those with manually constructed topologies [177]. However, the connectivity matrix representation has repeatedly been criticized for its bad scalability, since the size of the genome grows quadratically with the number of allowed neurons. Relying on a unique labeling of all neurons, the direct matrix encoding also persists to suffer from the permutation problem brought forward in section 4.1.2.

Several newer approaches aim to avoid these issues either by adding various extensions to the connectivity matrix encoding or by using different, more sophisticated representations. Two examples shall be described here in more detail, since they can be regarded as to represent the current state-of-the-art in direct encoding schemes for artificial neural networks.

the EPNet system

Yao *et al.* have proposed the EPNet system [236] that combines evolutionary computation with common backpropagation learning and simulated annealing (see section 4.3.1) and which has successfully been applied to numerous problems [236] [237]. The genetic representation used in EPNet persists to be based on the above connectivity matrix concept. One binary version codes the architecture, while another, real-valued matrix defines the corresponding weight values. In order to avoid the negative impact of the permutation problem, this approach does not use any crossover operator. Instead, it employs five different forms of mutation: hybrid training, node deletion, connection deletion, connection addition, and node addition.

hybrid training in EPNet

During hybrid training, the network is trained via backpropagation for a user-defined number of iterations and is then further optimized by simulated annealing. Being understood as a variation operator, this partial training is not intended to yield even a local optimum in performance, but it is the only operator in EPNet that modifies the weights of the network. The use of backpropagation for partial training restricts the investigated architectures to be feedforward.

structural mutation and scalability in EPNet

Although the number of hidden neurons is variable, the sizes of the two connectivity matrices are kept constant and are determined by the user-defined maximum number of inner nodes. An additional binary vector of the corresponding size is contained in each genotype and specifies the presence and absence of the individual neurons. On the basis of this encoding, all structural mutations can be implemented as simple bit-flipping operators. Any entries of the two connectivity matrices that belong to an absent node are simply ignored. Hence, while the EPNet system readily circumvents the permutation problem and even allows to

[1] It is assumed that the indices of the nodes increase from the inputs, over the hidden neurons, to the output neurons.

evolve individually sized networks, it retains an unfavorable scaling behavior.

the NEAT algorithm

Recently, a more elaborate direct encoding scheme has been proposed which is referred to as the NEAT method (NeuroEvolution of Augmenting Topologies) [199] [200]. It incorporates two different kinds of genetic units: node genes and connection genes. Like in a common connectivity matrix encoding, each connection gene specifies the presence/absence and the weight value of a single synaptic connection. But in contrast to the above representations, the genes are not arranged according to a fixed order. Rather, each connection gene internally codes the source and target nodes of the represented synapse. Beyond that, the genotype also contains a list of node genes that encode the labels of the present input nodes, hidden neurons, and outputs (see figure 4.3).

structural mutation in NEAT

In addition to the traditional random modifications of the weight values, a special mutation operator adds a new connection between two previously unconnected nodes. The number of neurons in the network is variable as well. A third dedicated mutation operator adds neurons to the network by splitting an existing connection in two and inserting the new node where the old connection used to be. As an important result, the numbers of the connection genes and node genes in the genotype are not fixed and will in general differ between the processed individuals.

recombination and speciation in NEAT

In order to match corresponding genes within two mating genomes, each connection gene contains a fixed, population-wide index (the "global innovation number") that is increased with every newly introduced connection in any genome of the whole population. Among other things, this allows for the implementation of a feasible crossover operator that can handle the varying size of the genotypes. It is not the purpose of this work to examine the elaborate details of the NEAT algorithm and the applied recombination operators. The interested reader is referred to the original publications [199] [200]. But it is worth noting that NEAT incorporates a specific distance measure to evaluate the similarity between two genomes. This measure is employed to implement speciation through fitness sharing as it is described in section 3.3.4.

It turns out that given the variability of the genome and the complexity of the search space, the implemented speciation mechanism is vitally necessary to efficiently explore different competing architectures. Otherwise, networks with more complex structures which might prove to be beneficial in the long run would soon be outperformed and suppressed by simpler networks that can be optimized more easily and thus reach higher fitness values considerably faster [200].

permutation problem: solved

Apart from that, it is a striking feature of the NEAT encoding scheme that it does not suffer from the permutation problem. This is primarily due to the fact that the source and target nodes of each connection are coded within the respective gene and that corresponding genes can be identified via similar values of the global innovation number. Similar, yet not as sophisticated, approaches that partly decouple the effect of a gene from its exact position in the genome and/or allow for genotypes of varying size have also been proposed by other authors [24] [166] [148].

Indirect Encoding Schemes

The undesirable scaling properties of most direct encoding schemes and their common susceptibility to the effects of the permutation problem have motivated the development of several indirect encoding methods [82] [117]. The basic idea behind an indirect representation is that it does not specify each single node or connection of the network explicitly but rather codes either general characteristics of the neural network architecture itself or specifications of a suitable network construction procedure.

parametric encodings

One approach that has been proposed by Harp *et al.* lets dedicated segments of the genotype code the features of corresponding subareas (layers) of the network [82]. Each segment primarily contains information about the number of neurons included in the respective area as well as their average connectivity to the nodes in other regions of the network. According to the rather coarse structural specifications that are contained in the genome, networks are then built via an automated, and partly stochastic construction rule. In order to yield a completely specified network, the synaptic weights of the resulting connections are finally trained by a suitable algorithm. The training parameters for each area, e.g., the learning rate η and the momentum ν of the backpropagation algorithm, are coded in the genome as well.

benefits and limitations

An even simpler variant of the indirect encoding would be to assume a strictly layered, homogeneously connected feedforward architecture and merely code the number of layers and their individual sizes. Coding schemes like these two examples which contain only certain parameters of the final network are commonly denoted as parametric representations. Although a parametric encoding method can significantly reduce the length of the genotype, the resulting network architectures are usually limited to a subset of the whole feasible architecture space.

developmental rule representations

According to another indirect representation that has originally been introduced by Kitano, the genotype codes the rules of a deterministic graph grammar which is then used to build the connectivity matrix C [117]. In other words, instead of including only the parameters of a fixed building rule that is used to construct the final network, the genome also codes the development rule itself. Such an encoding scheme is therefore referred to as a developmental rule representation.

compact genotypes

Like parametric encodings, developmental rule representations usually exhibit a good scaling behavior. Depending on the evolved set of rules, the resulting networks can become very complex without requiring the genotype to grow accordingly. The coding of construction rules also facilitates the definition of suitable representations that yield useful building blocks and can thus efficiently be combined with corresponding crossover operators. Some good results have been reported with this encoding method [117].

limited architecture range

On the other hand, developmental rule representations favor regular network architectures [26]. It can generally be said that depending on the used way of encoding, not every structure is equally probable, and this inherent bias may effectively prevent the algorithm from finding the optimal topology as it has actually been desired in the first place. Finally, the developmental rule encoding can only efficiently be employed for the evolution of architectures while the weight

values have to be trained separately. Experimental results suggest that it is not automatically superior to common direct encodings *per se* [191].

4.2.3 Fixed vs. Evolved Architectures

feasibility of topology evolution

In summary, it can be concluded that using either of the representational approaches, the evolutionary search for optimal network architectures ultimately faces adherent challenges. Simple direct encoding schemes exhibit poor scalability and persist to suffer from the permutation problem. The latter can be circumvented by abstaining from the use of recombination like it is done in EPNet, but even then, the poor scaling behavior remains. More elaborate variants like the NEAT algorithm constitute promising approaches. But they also require additional computational and organizational overhead in order to implement valid recombination operators and the necessary speciation mechanisms.

While indirect encodings allow for compact representations, they are inherently limited in the range of resulting architectures. Moreover, they have to defer the training of the weight values to separate algorithms which results in a computationally expensive an noisy fitness evaluation of the individuals.

evolving architectures: necessary?

Against the background of these issues it is worthwhile to reflect the initial motivation for evolving the topologies of neural networks: First, it is desired to find a network structure that yields the best performance on the investigated task, more specifically, that leads to an optimal generalization ability of the final network. Second, in order to feasibly implement the resulting networks on ordinary sequential computers, they are preferably compact and simple, i.e., comprise of as few neurons and connections as possible. It can be concluded that if sufficiently large and complex networks could efficiently be realized on a suitable substrate and if these networks could simultaneously be trained to yield the desired generalization performance — even when being limited to a predetermined architecture — an automated optimization of the topology might not be worth the effort.

Evolving Architectures for Neural Networks Implemented in Hardware

It has been motivated in section 2.4 that analog and/or hybrid CMOS VLSI technologies have the potential to provide efficient means of realizing such large, yet fast neural networks. The evolutionary training approach can greatly benefit from the high configuration and evaluation speeds of these systems and in so far ideally complements dedicated hardware implementations.

speed vs. computational complexity

Yet, if the training algorithm is not to compromise the speed advantage of the hardware substrate, it must itself be implemented efficiently (see also sections 6.2.6 and 6.3.3). From what has been said above, it can thus be deduced that an evolutionary training strategy for fast hardware neural networks should rather aim to keep up pace with the network implementation instead of dedicating major computational effort to the unnecessary minimization and optimization of the network topology. As long as the algorithm can succeed in producing networks with competitive generalization capability, abandoning the search for an ideal architecture does not necessarily constitute a limitation in practice.

On the other hand, the arguments brought forward in sections 2.1.1 and 2.2.1 suggest that improving the generalization ability of a neural network ultimately boils down to the optimization of its architecture, primarily the appropriate adaptation of its size. It has repeatedly been argued that the former does not seem to be achievable without the latter (e.g. [12] [139] [141]). In so far, it still awaits demonstration that fast and simple training algorithms which refrain from optimizing the network architecture can efficiently exploit the speed and potential size of hardware implemented networks while simultaneously yielding the desired generalization performance. This will be investigated more closely in chapters 8 and 9.

4.3 Alternative Black-Box Approaches

One of the main features of evolutionary algorithms that qualifies them for the training of highly recurrent and/or hardware implemented neural networks is that they do not require the used performance measure to be differentiable with respect to the free parameters, i.e., the weight values. In fact, the fitness needs not to be consistently expressible as a function of the synaptic weights at all. As noted in section 3.3.3, optimization algorithms with this characteristic are denoted as model-free or black-box approaches.

Although all of the experiments presented in this thesis employ evolutionary algorithms, other model-free, stochastic optimization algorithms have been proposed that should in principle be capable of training neural networks implemented on the used hardware substrate (chapter 5) as well. Since the main aspects of the results presented in part III are deemed to be transferable also to these alternative training algorithms, the two most prominent examples — simulated annealing and weight perturbation — shall briefly be discussed in the following.

4.3.1 Simulated Annealing

physical annealing

Similar to evolutionary algorithms, simulated annealing is inspired by an optimization process in nature. In condensed matter physics, annealing is a procedure to obtain the low-energy states of a solid. Initially, the material exhibits multiple imperfections in its crystalline structure, and it is the aim of the annealing process to yield the perfect regular structure that corresponds to the energetic ground state: First, the temperature of the solid is raised to the material's melting point. Then, the temperature is slowly lowered according to a predetermined cooling scheme until the desired low-energy state of the solid is reached.

Statistical Mechanics and the Metropolis Algorithm

statistical ensembles

In statistical mechanics, the behavior of a many-particle system is commonly analyzed by regarding large ensembles of identical systems and averaging their properties [54]. If the system is in thermal equilibrium with its environment at a temperature T, the contribution of each possible state[2] $\mathbf{r}_i$ that exhibits the energy

[2] The vector $\mathbf{r}_i$ specifies the values of all free parameters of the system in the represented state.

E_i is weighted by its Boltzmann probability factor

$$p_{\mathrm{B}}(\mathbf{r}_i) = \exp -\frac{E_i}{k_{\mathrm{B}}T} \tag{4.1}$$

where k_{B} is the Boltzmann constant.

the Metropolis algorithm

In 1953, Metropolis *et al.* introduced a simple iterative algorithm to simulate the behavior of a collection of atoms in thermal equilibrium at a given temperature T [138]. In each step, one of the atoms is applied a small random displacement in the phase space which results in a corresponding change ΔE of the overall systems energy E. If $\Delta E \leq 0$, the displacement is immediately accepted and the algorithm proceeds with the resulting new configuration of the system. Otherwise, the displacement is only accepted with a probability $P(\Delta E)$ calculated according to equation 4.1 with E_i being replaced by ΔE. When this procedure is iterated, it efficiently simulates the thermal motion of atoms in thermal equilibrium. The particular choice of $P(\Delta E)$ eventually causes the system to evolve into a Boltzmann distribution [54] [138], i.e., the probability $P_{\mathrm{T}}(\mathbf{s}_i)$ to find the system in a given state $\mathbf{s}_i$ with energy E_i is given by

$$P_{\mathrm{T}}(\mathbf{s}_i) = \frac{\exp -\frac{E_i}{k_{\mathrm{B}}T}}{\sum_j \exp -\frac{E_j}{k_{\mathrm{B}}T}} \tag{4.2}$$

where the sum in the denominator runs over all possible states.

Optimization by Simulated Annealing

basic idea

The basic idea behind solving optimization problems by simulated annealing is to combine the Metropolis algorithm with a slow decrease of the temperature parameter T [116].Here, the energy E_i of a given state is replaced by the performance measure $E(\mathbf{s}_i)$ of the candidate solution $\mathbf{s}_i$. Unlike in common evolutionary algorithms, the used performance measure is reasonably formulated such that the better the candidate solution $\mathbf{s}_i$, the lower the value of $E(\mathbf{s}_i)$. For neural networks, the error on the training set (equations 2.15 and 2.16) thus represents a suitable choice. Following the common nomenclature of simulated annealing, the used performance measure E is also referred to as the energy function.

iterative procedure

The optimization process starts with an arbitrary initial system of energy E_0 and a sufficiently high temperature parameter T_0. For a given number $N^I(T_0)$ of iterations, the Metropolis algorithm is applied, i.e., the candidate solution is repeatedly subject to random modifications — comparable to the mutations in an evolutionary algorithm — and the changes are accepted or rejected according to the scheme described above.

temperature variation

Once the $N^I(T_0)$ iterations have been performed, the temperature parameter T is lowered to a new value $T_1 < T_0$ and the Metropolis algorithm is continued for a number of $N^I(T_1)$ iterations, this time using the new temperature T_1 to calculate the transition probabilities from one candidate solution to another. This whole procedure is repeated until after the nth temperature step, the resulting temperature T_n has decreased below a given threshold. In analogy to physical

annealing, this algorithm is expected to exhibit a high probability of converging to the global minimum of the energy function E on the search space.

Discussing Simulated Annealing

adjustable parameters

In order to be successfully applied to a specific optimization task, several aspects of the simulated annealing algorithm have to be tuned: The initial temperature T_0, the number of iterations per temperature step $N^I(T)$, $N^I\colon \mathbb{R} \to \mathbb{N}$, the size of the ith temperature step $\Delta T(i)$, $\Delta T\colon \mathbb{N} \to \mathbb{R}$, the termination temperature T_t, and, finally, the stochastic perturbation mechanism that modifies a given candidate solution to yield a new one. The first four parameters are commonly denoted as the applied cooling scheme.

thermal equilibrium

The initial temperature T_0 is preferably chosen high enough as to allow all possible modifications of the candidate solution to be accepted with approximately equal probabilities. This corresponds to the liquid state of a physical system where the particles are arranged totally randomly. If the temperature is lowered sufficiently slowly — as determined by $N^I(T)$ and $\Delta T(i)$ — the algorithm can at each temperature T_i reach a state that corresponds to the thermal equilibrium in a physical system, i.e., the probability $P_\mathrm{T}(\mathbf{s}_i)$ to encounter a specific candidate solution $\mathbf{s}_i$ with energy E_i will be given by equation 4.2. Besides the capability of the applied perturbation mechanism to create every possible candidate solution from any previous one, the thermal equilibrium condition is essential for proving the asymptotic convergence of simulated annealing to the optimal solution [102].

efficiency

heuristic character

Depending on the current temperature T, even disadvantageous modifications of the candidate solution stand a chance to be accepted. This way, simulated annealing can in effectively escape local minima of the energy function E and therefore exhibits an increased capability of finding the true optimal solution compared to ordinary local search schemes [102]. Unfortunately, its warranted asymptotic convergence merely guarantees that the optimal solution is found in the limit of an infinite number of steps. In practice, one is naturally interested in finding a good solution within a finite time. On the other hand, like evolutionary algorithms, simulated annealing has proven to be readily and successfully applicable to a multitude of different problems, and in so far, it constitutes another example for a heuristic (see section 3.3.3).

4.3.2 Weight Perturbation

gradient descent

As described in sections 2.1.3 and 2.2.2, common gradient descent training approaches iteratively calculate the modification of a synaptic weight w_{ij} on the basis of the corresponding derivative of the network error E:

$$\Delta w_{ij} = -\eta \, \frac{\partial E}{\partial w_{ij}} \tag{4.3}$$

where η is an adjustable learning rate and E quantifies the difference between the network response and the target output on a fixed set of training examples (equations 2.15 and 2.16).

In the case of strictly feedforward networks of neurons with continuous activation functions, the necessary derivatives can readily be calculated. This eventually leads to the formulation of the delta-rule and the backpropagation algorithm for single-layer and multi-layer networks, respectively.

Derivative Approximation

evaluating perturbations

When the gradient of the error function is either costly to obtain or cannot be determined analytically at all, it stands to reason to avoid its calculation by probing the dependence of the error on the single weights directly. The derivative of the error E with respect to a single weight w_{ij} can be approximated by applying a small perturbation $\Delta_\mathrm{p} w_{ij}$ to the current weight value and evaluating the resulting change in the error $\Delta E = E(w_{ij} + \Delta_\mathrm{p} w_{ij}) - E(w_{ij})$:

$$\frac{\partial E}{\partial w_{ik}} = \frac{E(w_{ij} + \Delta_\mathrm{p} w_{ij}) - E(w_{ij})}{\Delta_\mathrm{p} w_{ij}} + \mathcal{O}(\Delta_\mathrm{p} w_{ij}) \tag{4.4}$$

As long as $\Delta_\mathrm{p} w_{ij}$ is sufficiently small, a good approximation of the optimal weight change is then given by

$$\Delta w_{ij} = -\eta \, \frac{E(w_{ij} + \Delta_\mathrm{p} w_{ij}) - E(w_{ij})}{\Delta_\mathrm{p} w_{ij}}. \tag{4.5}$$

training with weight perturbations

This weight update rule does not require any information beyond the size of the applied perturbation and the respective performance of the original and modified network. The iterative training procedure that adopts this update scheme is referred to as the weight perturbation algorithm [106]. Originally, it has been suggested as an efficient training procedure for recurrent neural networks implemented in analog VLSI. Several similar algorithms have been proposed which also approximate the derivative of the error function by measuring the differences between slightly perturbed versions (e.g., [34] [47] [55] [226]).

Discussing Weight Perturbation

generality

Since it allows to apply the general principles of gradient descent to any optimization problem, even those for which the necessary derivatives cannot be obtained analytically, the concept of approximating the desired gradients by finite differences is not limited to neural network training. On the other hand, weight perturbation naturally inherits all problems of the gradient descent approach, i.e., it is highly susceptible to local optima of the error function.

local optima

efficiency

While they solely rely on the network's response to the applied set of training examples — an information which is reasonably available for any network implementation — weight perturbation and most related approaches lead to a considerably larger amount of network evaluations than true gradient descent methods. Each single weight update requires to measure the error E of a new, slightly modified network. In this respect, these algorithms are optimally suited for the on-chip or chip-in-the-loop training of dedicated hardware implementations that do not provide detailed information about their internal state but have the capability of efficiently implementing multiple networks with high speed (see section 2.4.5).

4.3.3 Comparison to Evolutionary Algorithms

In the light of the No Free Lunch theorem (see section 3.3.3), none of the three described black-box approaches for neural network training — evolutionary algorithms, weight perturbation, and simulated annealing — can initially be deemed to be superior to the remaining two. Furthermore, either approach represents feasible means of training neural networks implemented in analog CMOS VLSI.

global and local search

Nevertheless, compared to evolutionary algorithms and simulated annealing, weight perturbation is more likely to get trapped in local optima of the performance measure. Like true gradient descent algorithms, it is good at fine-tuning the weight values but is less effective in reliably finding the global optimum.

simulated annealing vs. evolutionary algorithms

Evolutionary algorithms and simulated annealing exhibit some important similarities. Both incorporate random changes of the candidate solutions, and the probabilistic acceptance scheme of the Metropolis algorithm can be seen in analogy to the survivor selection process in evolutionary algorithms. In so far, simulated annealing could even be viewed as a special kind of evolutionary algorithm with a population size of 1, a generational replacement scheme, a probabilistic survivor selection process, and an automated adjustment of the selection pressure (given by the slow decrease of the temperature parameter T). But while common evolutionary algorithms do not incorporate an automated adaption of the selection parameters, they usually benefit from the parallel processing of more than one individual and potentially draw additional computational power from the appropriate recombination of different candidate solutions.

Due to these considerations and motivated by the multitude of reported successes in combining evolutionary computation with artificial neural networks (see above), the experiments described in this thesis exclusively employ the evolutionary approach. However, the discussed results do by no means aim to suggest the general superiority of evolutionary algorithms over alternative model-free training algorithms. In fact, the presented observations are regarded to be of more universal nature, and this topic will be returned to in section 9.5.

Part II

Hardware Neural Network Framework

Chapter 5

The HAGEN Chip

Any sufficiently advanced technology
is indistinguishable from magic.

Arthur C. Clarke

The investigations presented in this thesis explore the applicability of evolutionary algorithms (see chapters 3 and 4) to the training of large hardware neural networks (see chapters 1 and 2). The central part of the hardware platform that is used throughout all of the presented experiments is the HAGEN chip (Heidelberg Analog Evolvable Neural network). HAGEN is a mixed-signal[1] ASIC that allows for the efficient implementation of fast, low power consuming, and nearly arbitrarily scalable networks. It has been designed by Dr. Johannes Schemmel [179].

Although the majority of the results presented in chapters 8 and 9 is regarded to be of general nature and should readily be transferable to other network implementations as well — regardless of whether they be realized in hardware or in software (see also section 10.4) — the introduced training strategy has been devised with particular regard to the characteristics of the used network chip. In turn, the concepts that underlie the design of HAGEN pay special tribute to the compatibility with highly-iterative chip-in-the-loop training algorithms like they are, e.g., represented by evolutionary strategies[2].

While the HAGEN prototype demonstrates the general feasibility of its underlying ideas, the proposed model for the realization of fast and scalable hardware neural networks is also deemed to be a promising basis for more evolved future ASICs that take full advantage of contemporary CMOS technologies. Therefore, this chapter does not only intend to introduce the HAGEN prototype itself but also to discuss and motivate some of the general considerations that form the basis of its design.

The chapter will be concluded with an investigation of how the inevitable device variations introduced during fabrication of an analog ASIC affect the network operation of the HAGEN chip. Procedures for the evaluation of these deviations

[1] That is, it follows a hybrid approach that combines analog computing with digital signaling (see section 2.4).

[2] Hence the name.

are devised, experimental results are presented, and strategies for the efficient compensation of the observed effects are proposed.

5.1 Design Considerations

The development of the HAGEN ASIC has been driven by the goal of implementing a general-purpose, mixed-signal VLSI neurochip in a standard CMOS technology that combines the high speed and efficiency of analog devices with the scalability and flexibility of digital computing (see section 2.4).

5.1.1 Speed and Efficiency

speed considerations

A high operational speed of the network as well as a fast reconfigurability of its weights and architecture are vital preconditions for the efficient application of highly iterative training strategies such as evolutionary algorithms. However, given the ongoing progress in computer technology, the absolute speed of a system has proven to be a short-lived quality. The presented design concept rather pursues to exploit the fundamental speed advantage of analog computing in comparison to digital solutions.

challenges to analog VLSI

Apart from speed considerations, the aim to make optimal use of the silicon substrate also implies to minimize area and power consumption. While analog VLSI has the potential to fulfill all of these requirements, the automation of analog integrated circuit design is considerably harder than in the case of purely digital implementations. The feasibility of an analog solution will thus be judged against the background of the increased design times and non recurring engineering (NRE) costs. It has been argued that any analog approach needs to exhibit an enhanced efficiency in performance, size, or power consumption of at least one order of magnitude to become more commercially attractive than digital ones [122].

This is most unlikely to be achieved for general purpose analog processors but can realistically be targeted at for specific applications. In the particular case of neural networks, several analog implementations have been proposed in literature [137] [16] [215] [216].

exploiting transistor physics

The characteristic curves of transistors provide a wide range of functionality (nonlinear amplification, linear amplification, thresholding, etc.) [215]. In order to take full advantage of the capabilities of the CMOS technology, the used neural network model has to be mapped onto the analog properties of these elementary components instead of using them merely as binary switches. This is not a trivial task and requires the application of a model that explicitly allows for an efficient representation by physical quantities [16] [215] [216].

5.1.2 Scalability

The need for scalability towards larger networks usually arises from the complexity of the tasks to be solved. Up-scaling constitutes a challenge not only to the technical realization of the network but also to the underlying model and the applied training strategies.

up-scaling and analog noise

It has to be taken into account that analog designs are potentially more vulnerable to noise and offsets than digital systems. Digital quantities can be stored and transmitted far easier than analog signals. The reliable propagation of information across large distances is of vital importance for the available connectivity in a large-scale network. In this respect, purely analog implementations can hardly be anticipated to provide satisfactory scalability.

miniaturization

On the other hand, for a large-scale realization of the network to be feasible at all, the elementary operations must be implemented efficiently. Again, this refers to power consumption as well as area requirements and militates in favor of analog implementations.

5.1.3 The Mixed-Signal Approach

The aim for high performance, low power consumption and feasible miniaturization is to be reconciled with the desire for good scalability to larger systems. This motivates a mixed-signal design that combines analog computing with digital signaling. While all computationally expensive tasks are to be implemented in analog VLSI, digital circuits will provide reliable communication and flexible routing to allow for the implementation of large networks.

array-based structures

It has been proposed that the analog operation is preferably organized in a regular, array-based structure with digital interface [25] [122] [68]. This represents an efficient way to implement vector-matrix multiplications that form the basis for various neural network paradigms such as feedforward networks (chapter 2) or support vector machines (section 2.3.1).

5.1.4 Trainability

When arguing in favor of a mixed-signal implementation, it cannot be ignored that inevitable device mismatches introduced during manufacturing as well as fluctuations in the analog signals during operation persist to impose restrictions on the applicable training algorithms. As motivated in section 2.4.5, off-chip learning can deal with such unreliable substrates only to a limited extent, and this leaves either on-chip or chip-in-the-loop algorithms as viable alternatives.

chip-on-the-loop vs. on-chip training

Compared to on-chip learning, the chip-in-the-loop approach naturally provides enhanced flexibility. Hence, as long as the design of appropriate training algorithms remains a subject of research by itself, network models and implementations that are suited for chip-in-the-loop algorithms are deemed the more appropriate alternative.

5.2 Network Model

McCulloch-Pitts neurons

The chosen network model is designed as to account for the prescriptions summarized in the preceeding paragraphs. Its basic computational unit is the McCulloch-Pitts neuron described in section 1.2.3, and multiple neurons are arranged to form so-called network blocks as shown in figure 5.1. One network block implements a fully connected, single-layer perceptron with multiple outputs by connecting each

fully connected perceptron

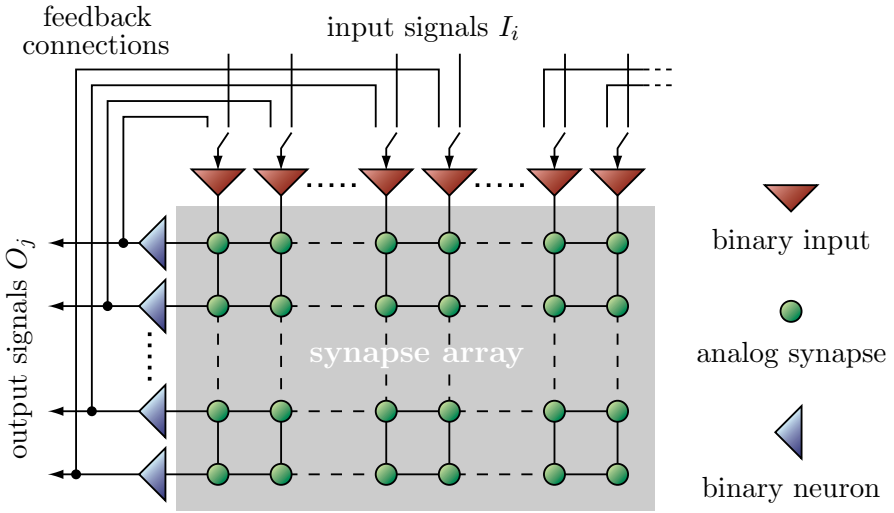


Figure 5.1: *One network block implements a single-layer perceptron by connecting each of the binary input nodes I_i to each binary output neuron O_j via an individual analog synapse with programmable weight w_{ij}. Feedback connections allow to apply the network output of a preceeding network cycle as input for the present loop. This way, a large variety of network architectures can be realized (see also figure 5.2).*

of the N_{in} binary inputs I_i with each of the N_{out} binary output neurons O_j via an individual synaptic connection with the analog weight w_{ij}.

Similar to equation 1.5, the output of the jth neuron in response to the application of a set of inputs is thus given by

$$O_j = \varphi\left(\sum_{i=1}^{N_{\text{in}}} w_{ij}\, I_i\right) = \Theta\left(\sum_{i=1}^{N_{\text{in}}} w_{ij}\, I_i\right) \qquad \text{with } I_i, O_j \in \{0,1\}. \tag{5.1}$$

mixed-signal implementation

The utilization of threshold neurons automatically sets up the desired border between the analog and the digital part of the network implementation. Both, the weighting of inputs via individual synaptic strengths and the calculation of the total input *net* (see section 1.2.3) can efficiently be realized in analog VLSI. At the same time, the inputs are reduced to binary switches and the neuron outputs are digital signals.

5.2.1 Configurable Topology

time-discrete operation

The network operates in discrete time steps: After the application of an input vector $\mathbf{I}$, the network output is available after a finite time interval Δt. This is denoted as one network cycle, and given the length of one such loop, the network frequency f_{net} is defined as

$$f_{\text{net}} := \frac{1}{\Delta t}. \tag{5.2}$$

feedback connections

External connections allow to feed the neuron output back to the inputs of the network block. Thereby, the network's response $\mathbf{O}(t)$ after a preceeding cycle

can partially contribute to its input in the present loop. More specifically, the feedback connections define a mapping $c\colon \mathbb{N} \to \mathbb{N}$ from the indices of the inputs to the indices of the neurons such that $c(k)$ denotes the index of that specific output which is fed back to the input k: $I_k(t) = O_{c(k)}(t)$. The actual form of this mapping depends on the current set of activated feedback connections. Some of the input nodes i will not receive feedback at all and rather accept external input[3] I_i. In this case, c shall be defined as to obey $c(i) = 0$. The network response then assumes the form

$$O_j(t + \Delta t) = \Theta\left(\sum_{\substack{i=1 \\ c(i)=0}}^{N_{\text{in}}} w_{ij}\, I_i(t) + \sum_{\substack{k=1 \\ c(k)\neq 0}}^{N_{\text{in}}} w_{kj}\, O_{c(k)}(t)\right). \tag{5.3}$$

Here, it is reasonably assumed that any input node can only receive either external signals or feedback, but not both. Also, the output of each neuron can be fed back to at most one input of the same network block. Since within the block, each input I_i potentially contributes to the total input *net* of all neurons, this is no restriction in practice.

recurrent architectures

In combination with an appropriate choice of weights, the activation of specific feedback connections allows to implement different network architectures. If the network is operated for more than one cycle and if there exist indices k such that $c(k) \neq 0$ and $w_{k\,c(k)} \neq 0$, the network contains direct feedback loops and is thus a recurrent network in terms of section 1.2.2. In the case of more than two network cycles, more complex closed loops can be realized in which a neuron feeds back to itself via other neurons. By setting all respective weights to zero, strictly feedforward networks with multiple layers can be implemented as well. Simple examples for both configurations are shown in figure 5.2. For a feedforward network, the number of necessary cycles n_{nc} directly corresponds to the number of desired layers.

feedforward architectures

5.2.2 Multiple Network Blocks

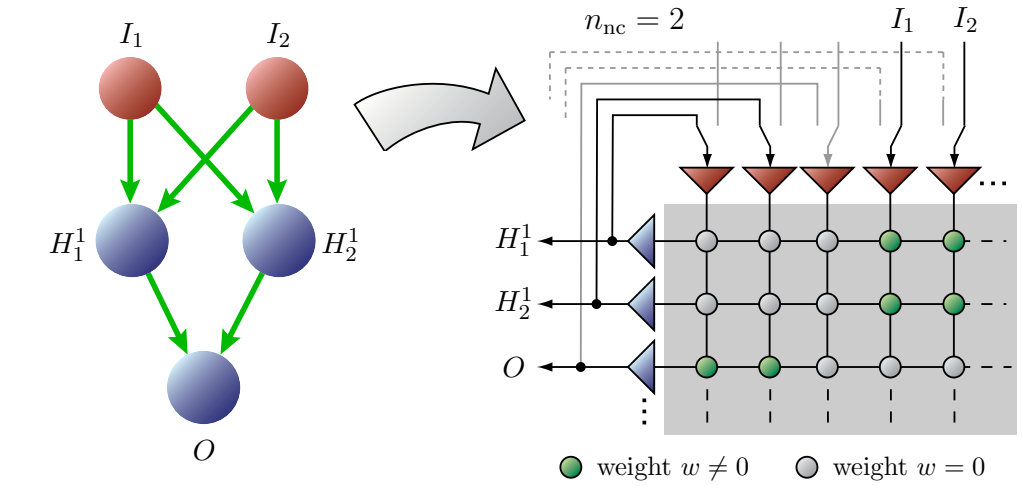
inter-block connections

It is one of the key features of this model that it can readily be expanded to multiple network blocks. Similar to the feedback connections described above, inter-block connections can transfer the neuron output of one block to the inputs of other blocks (see figure 5.3). In the spirit of the preceeding paragraph, define the mapping $c^{ba}\colon \mathbb{N} \to \mathbb{N}$ such that $c^{ba}(l)$ is the index of the specific neuron in block b the output of which is fed back to the input node l of block a. The initial formalism introduced above is contained within this expanded model in the form of the mappings c^{aa}. For a more concise notation, let $c^{ba}(l)$ simply be denoted as l^{ba}.

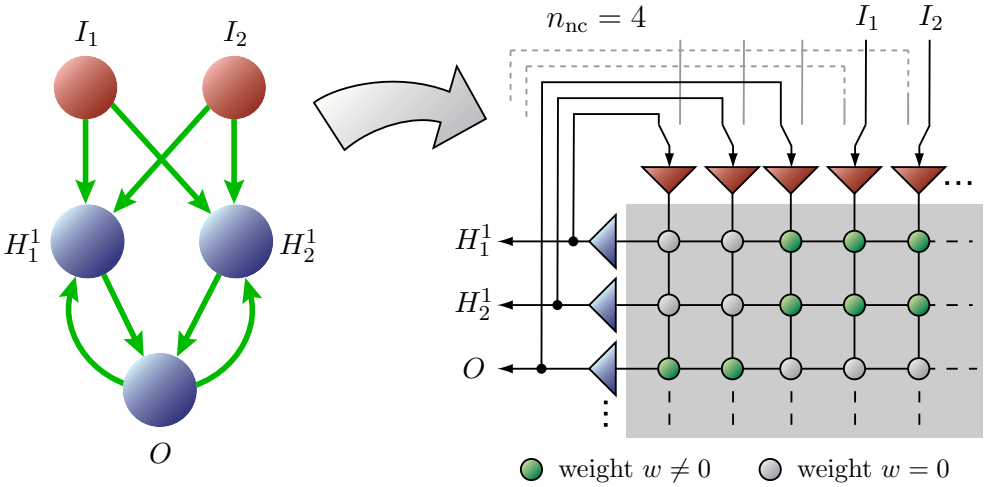
synchronous operation

Any input that originates from another block will in general arrive with a time delay $\Delta t' \geq \Delta t$. If it is ensured that all blocks operate at the same network frequency f_{net} and that the time delay introduced by the inter-block connections

[3] It is assumed that the number of output neurons N_{out} is chosen to be smaller than the number of inputs N_{in} to the network block (see also section 5.4.1).



a)



b)

Figure 5.2: ***a)*** *By utilizing the available feedback connections and a corresponding number of network cycles n_{nc}, networks of multiple layers can easily be implemented. Absent connections are realized by setting the respective synapses to zero. The shown example employs $n_{\mathrm{nc}} = 2$ network cycles to implement a two-layer architecture.* ***b)*** *By enabling additional feedback lines, allowing for more network cycles, and setting the appropriate synapses to finite values, recurrent network architectures can be implemented as well (see also section 1.2.2). If fewer synapses are restricted to zero, even more complicated topologies are possible.*

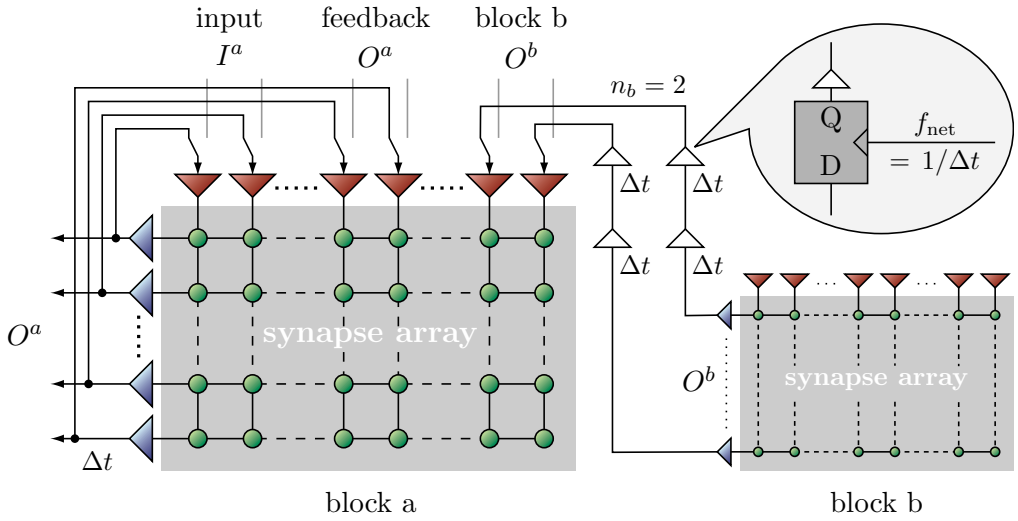


Figure 5.3: *Inter-block connections transfer the output of one block to the inputs of other blocks. These inputs are ensured to arrive with a time delay $\Delta t'$ that is an integer multiple of $\Delta t = 1/f_{\text{net}}$. This way, several networks blocks can be synchronously operated as formulated in equation 5.4 (after [179]).*

is always an integer multiple of $\Delta t = 1/f_{\text{net}}$, the response of a given neuron j in block a can consistently be formulated as

$$O_j^a(t+\Delta t) = \Theta\left(\sum_{\substack{i=1\\ i^{aa}=0}}^{N_{\text{in}}} w_{ij} I_i^a(t) + \sum_{\substack{k=1\\ k^{aa}\neq 0}}^{N_{\text{in}}} w_{kj} O_{k^{aa}}^a(t) + \sum_{b\neq a} \sum_{\substack{l=1\\ l^{ba}\neq 0}}^{N_{\text{in}}} w_{lj} O_{l^{ba}}^b(t - n_b \Delta t)\right) \tag{5.4}$$

where $n_b \in \mathbb{N}$ for all b. The second term refers to feedback connections within the block, the last term represents incoming connections from other blocks b that arrive with a delay of n_b cycles, respectively. For the example shown in figure 5.3, n_b equals 2. As long as the condition $\Delta t' = n\Delta t = n/f_{\text{net}}$, $n \in \mathbb{N}$ is fulfilled for all connections, this model allows the synchronous operation of arbitrary large networks.

5.3 VLSI Implementation

By enclosing the analog part of the network operation into blocks with digital interfaces, the described model lays the foundation for the desired up-scaling ability. The targeted low power consumption, area-efficiency, and high speed remain to be achieved by suitable realizations of the neuron and synapse circuits. In this context, it is a notable feature of equation 5.4 that the use of binary inputs I_i effectively reduces the calculation of the total input *net* to a mere summation of weight values. A summation of currents can easily be implemented in CMOS.

5.3.1 Binary Neurons, Trainability, and VLSI Design Implications

multi-valued inputs

Many hardware implementations of feedforward networks do not restrict themselves to binary inputs and outputs. If the inherent parallelism in the network operation is to be fully exploited, the necessary multiplier circuits do in this case have to be incorporated within each synapse [79]. This naturally leads to increased area requirements of the synapse implementations. Another disadvantage of this approach is the difficulty for analog multiplier circuits to achieve simultaneously a high dynamic range, a good noise resistance and a low power consumption [79].

Multi-valued inputs can be simulated by using binary inputs in connection with appropriately dimensioned weights as will be described in section 8.2.2. In so far, the use of binary input nodes does not inhibit the processing of multi-valued inputs. At the same time, it allows to reduce the complexity of the synaptic circuits and thus promotes smaller and faster synapse implementations.

analog multiplication

There is another important advantage of using binary inputs that arises from the fact that an analog summation is only affected by additive offset mismatches [122]. In contrast, analog multiplication potentially suffers from offsets in the inputs as well as additional variations in the gain. A purely additive offset can easily be compensated, since it is independent of the inputs and the weights (see section 5.5). This does not hold for the gain and offset mismatches in a multiplication circuit [144].

training networks of threshold neurons

Nevertheless, the use of binary neurons also restricts the range of suitable training approaches: It has been discussed in section 2.2.2 that due to the credit assignment problem, the formulation of efficient training algorithms for networks of simple threshold neurons with more than one layer remains a considerable challenge. The popular backpropagation training algorithm for multi-layer networks (section 2.2.2) vitally depends on the neurons to exhibit a continuous output and is thus not viable in this case. But it has repeatedly been motivated (chapters 2, 3, and 4) that adequate black-box optimization strategies, most notably evolutionary algorithms, can successfully be utilized for this kind of task.

benefits of chip-in-the-loop training

The efficient application of these highly iterative generate-and-test approaches requires a high speed of the network operation. On the other hand, when being implemented either on-chip or as chip-in-the-loop training, evolutionary algorithms can adequately cope with potential device mismatches and high noise levels. Consequently, a network implementation that is intended for use in conjunction with evolutionary algorithms or other model-free strategies should be able to exploit the small feature sizes of contemporary CMOS technologies to the fullest without running the risk of compromising its trainability.

local weight storage

Finally, striving to benefit from the high degree of parallelism that is inherent in the concepts of neural information processing does not only imply to include the necessary computational circuitry within each single synapse. It also requires the synapses to have direct access to their predetermined weight values and thus to incorporate an adequate means of weight storage into the synapse implementation.

5.3.2 Circuit Design

mixed-mode operation

The proposed synapse and neuron circuits are designed in consideration of the above conclusions and are schematically illustrated in figure 5.4. The synapse combines voltage mode and current mode operation: While the postsynaptic activity is represented by a current $\mathrm{I_{ps}}(\mid w \mid) > 0$ that is sank into the synapse according to the absolute value of its weight w, the weight value itself is stored as charge on a capacitor. On a general level, the synapse acts like a voltage controlled current sink (for details see [179]).

excitatory and inhibitory currents

The sign of the weight is stored in a dynamic latch. Depending on the state of this latch, a synapse is either connected to the excitatory line $\mathrm{I_+}$ or the inhibitory line $\mathrm{I_-}$ of the postsynaptic neuron. By virtue of Kirchhoff's law, the total excitatory and inhibitory input currents $\mathrm{I_{pos}}$ and $\mathrm{I_{neg}}$ to the neuron that result from the contribution of all its activated synapses become

$$\mathrm{I_{neg}} = \sum_i \mathrm{I_{ps}}(|w_i|) \quad \forall\, i \in \{j \mid w_j < 0\} \tag{5.5}$$

$$\mathrm{I_{pos}} = \sum_i \mathrm{I_{ps}}(|w_i|) \quad \forall\, i \in \{j \mid w_j > 0\} \tag{5.6}$$

As long as the input node of a specific synapse is not activated, it gets connected to the third current line $\mathrm{I_{park}}$ (see next section). The Resistors $\mathrm{R_+}$ and $\mathrm{R_-}$ of the neuron circuit convert the respective sums of the excitatory and inhibitory currents to voltages $\mathrm{U_{pos}}$ and $\mathrm{U_{neg}}$ according to Ohm's law:

$$\mathrm{U_{neg}} = \mathrm{V}_{dd} - \mathrm{R_-} \cdot \mathrm{I_{neg}} \tag{5.7}$$

$$\mathrm{U_{pos}} = \mathrm{V}_{dd} - \mathrm{R_+} \cdot \mathrm{I_{pos}} \tag{5.8}$$

A comparator amplifies the resulting difference $\mathrm{U_{pos}} - \mathrm{U_{neg}}$ to logic levels.

5.3.3 Implementation Properties

constant power consumption

Since the synapses continuously sink a current to either $\mathrm{I_+}$ $\mathrm{I_-}$ or $\mathrm{I_{park}}$, the power consumption of this design is independent of the input activity and solely determined by the programmed weights. From a technical point of view, a constant power consumption is advantageous in so far as it also implies a constant temperature and thus largely avoids temperature induced side effects on the circuit operation. Besides that, it suppresses power supply noise [178]. Finally, the chosen concept allows to adjust the overall power consumption via the programming of appropriate weights.

weight storage and weight refresh

The proposed weight storage circuit not only fulfills the demands for a small, fast, and energy-saving synapse as well as a truly parallel synapse operation. It also admits frequent changes in the weight values, e.g., during training. On the other hand, leakage currents in the synapse circuit will cause the weight capacitor to discharge over time. The actual time scale on which these weight deterioration effects measurably occur depends on the utilized CMOS process. In the explicit case of the 0.35 μm process used for the HAGEN prototype, the leakage time is in the order of 10 to 100 ms and therefore several orders of magnitude larger than

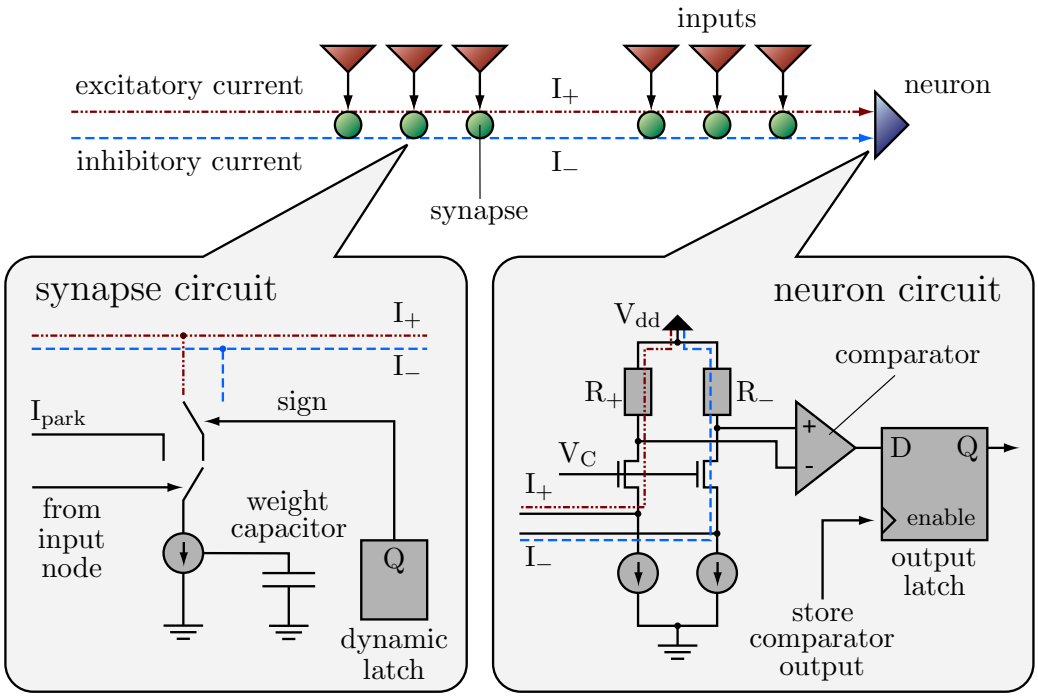


Figure 5.4: *Operational principle of the implemented synapse and neuron circuits (after [179]). The synapse acts like a voltage controlled current sink that is connected to either the excitatory line I_+ or the inhibitory input line I_- of the neuron, depending on the stored sign. The weight is stored as charge on a capacitor. Within the neuron, the total excitatory and inhibitory currents I_{pos} and I_{neg} are converted to respective voltages U_{pos} and U_{neg} and their difference is evaluated by a comparator. The shown circuits are simplified and merely illustrate the general principle of operation. For a detailed account of the implementation see [179].*

the time of one network cycle Δt (see next section). Hence, the necessary weight refreshes have to be performed comparatively rarely.

weight generation and digital storage

Still, the limited life expectancy of the programmed values requires an additional digital weight storage that can serve as a fast and long-lasting memory and which can in principle either be located on-chip or off-chip. The same applies to the digital to analog converters (DACs) that are used to convert the stored digital values into the analog synaptic weights. However, realizing the necessary DACs on-die has the advantage of reducing the chip interface to an entirely digital communication, and in the case of the HAGEN chip, it has therefore been chosen to integrate the DACs on the ASIC. At the same time, the digital weight storage itself remains off-chip as to retain the possibility to explore different training strategies that can more flexibly be implemented in software or in a programmable logic.

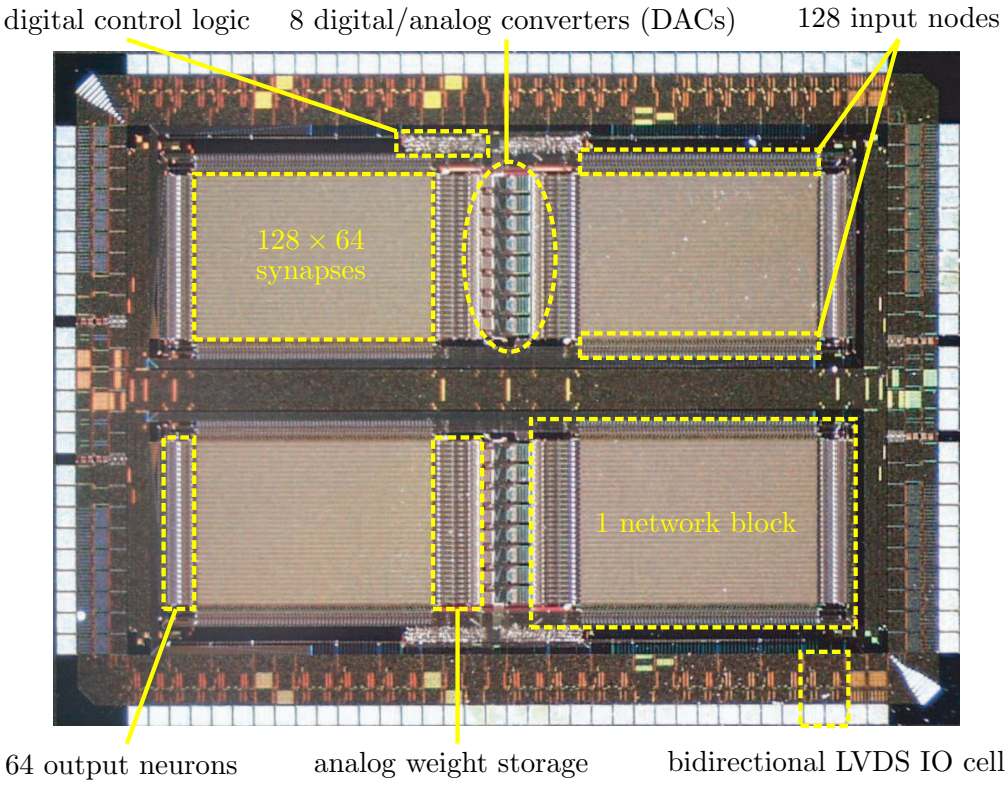


Figure 5.5: *Photograph of the HAGEN ASIC. The chip hosts four network blocks and is organized in two halves, upper and lower. The halves are mirrored but otherwise identical (see also figure 5.6).*

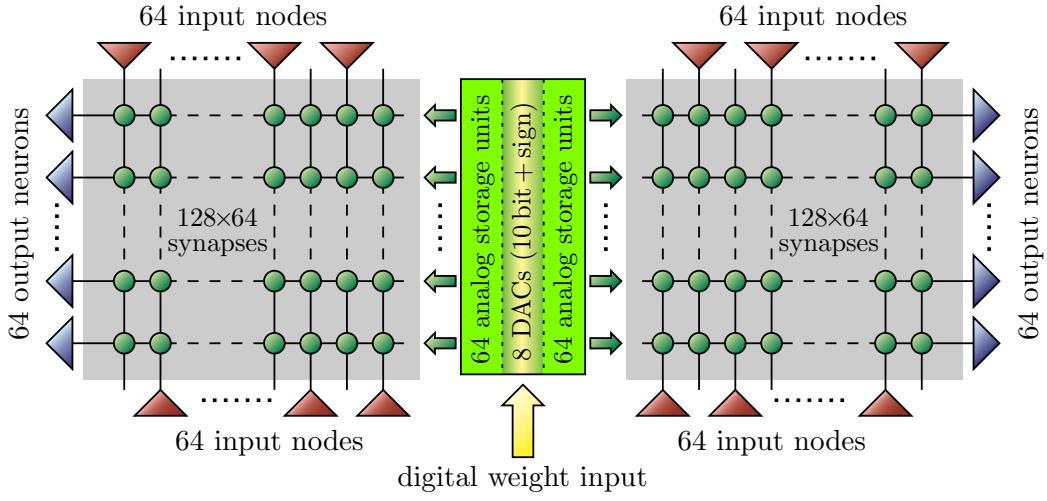


Figure 5.6: *Schematic drawing of the upper half of a HAGEN ASIC. The two synapse arrays are fed by inputs from the top and the bottom but the output neurons of the blocks all face towards the respective outside. Each block is accompanied by its own set of 64 analog storage units. The central DAC unit of 8 DACs is shared by both blocks (see section 5.4.5).*

specification	HAGEN prototype
process features	0.35 μm, 1 poly, 3 metal
die/core size	$4.1\times3\,\text{mm}^2$ / $3.6\times2.5\,\text{mm}^2$
synapse size/density	$8.7\times12\,\mu\text{m}^2$ / 9580 per mm^2
blocks/neurons/synapses	4/256/32768
supply voltage	3.3 V
network frequency f_{net}	50 MHz typ.
CPS	1.64 Teracps max.
CUPS	400 Megaweights/s max.
LVDS bus data transfer rate	11.4 Gigabit/s max.
weight resolution	10 bit (nominal) + sign

Table 5.1: *Nominal specifications of the HAGEN prototype.*

5.4 The HAGEN Prototype

The HAGEN chip represents a prototype implementation of the introduced network model and circuit designs that aims to demonstrate their general usability for the implementation of low power consuming, area-efficient, fast, and scalable neural network ASICs. HAGEN is fabricated in a 0.35 μm process by Austria Mikro Systeme International [7]. A die photograph is shown in figure 5.5, and table 5.1 lists some of the chip's specification data. HAGEN has been described in detail in a number of technical publications [179] [180]. The following sections concentrate on summarizing what is relevant for the work presented in this thesis.

overview

The ASIC contains four equally sized network blocks and is organized in two identical halves, labeled *upper* and *lower* in correspondence to their physical arrangement. Within each half, the two adjacent network blocks — *left* and *right* — are laid out such that the neuron outputs of each block face towards the respective die edge while the weight storage circuitry is provided on the other. This allows two share the DACs for the analog weight generation between the two blocks of each half (see figures 5.5 and 5.6).

LVDS interface

HAGEN includes a digital control logic that communicates with external hardware via bidirectional LVDS (Low Voltage Differential Signaling) IO cells [5]. This bus interface is used to write the weight values into the synapse array, to write and read back the network input and output data, and to control the network operation. In its current implementation, the interface does not provide a pipelined handling of input and output data and can therefore not actually cope with the speed of the network. Apart from that, the prototype character of the HAGEN chip manifests itself mainly in the limited available silicon area and a corresponding restriction to the number of contained synapses.

5.4.1 Block Dimensioning

The four network blocks comprise of 128 input nodes and 64 output neurons each (see figures 5.5 and 5.6). The resulting 8192 synapses of one block are arranged in a rectangular array that is fed by inputs from the top and the bottom[4].

complete feedback and external input

The four network blocks each include a number of inputs that is twice the number of their outputs. This dimensioning potentially allows to completely feedback each block to itself and to simultaneously accept input from other blocks or external sources. Regarding the absolute numbers of inputs and neurons, larger network blocks are possible, but the number of inputs to one neuron is ultimately limited by the efficiency of the analog processing. Using the proposed circuit designs, up to approximately 1000 inputs per block can be realized [178], and the total number of neurons is limited only by the desired size of the ASIC.

feedforward networks on multiple blocks

In fact — as it has been motivated above — one of the primary goals of the HAGEN prototype is to explore the feasibility of encapsulating the analog operation into blocks with digital interfaces that can easily be combined to form larger networks. Beyond that, implementing several smaller blocks instead of a single large one can also be motivated by another consideration: In a strictly feedforward topology with multiple layers, a large fraction of the available synapses remains unused, i.e., has to be set to zero. Since the power consumption of the synapse circuit is solely determined by the value of the programmed weight, any unused synapses have no further negative effect apart from occupying silicon area. Nevertheless, for the realization of feedforward architectures with multiple layers that contain only few neurons each, a single network block with a large number of inputs is inadequate. The same topology can more efficiently be implemented with several smaller blocks and corresponding inter-block connections.

5.4.2 Block Interconnectivity

hard-wired connections

As another tribute to the prototypic nature of the HAGEN chip, the available feedback and inter-block routing is limited to a fixed set of selected, hard-wired connections. The resulting connectivity is schematically depicted in figure 5.7.

left blocks

The blocks on the left side can potentially feedback all of their 64 neuron outputs to a given subset of their own inputs and can receive up to 40 connections from blocks on the opposite side. Apart from that, 24 input nodes per block can be connected to 12 standard CMOS IO pads of the ASIC to allow for the direct connection of external data sources to the network (denoted as “direct in” in figure 5.7). The upper and lower left block both receive signals from the same set of direct input pads plus the respective inverse signals.

right blocks

The blocks on the right side only exhibit 32 feedback connections but can receive extensive input from blocks on the left side via up to 96 inter-block links. In similarity to the direct inputs of the left-sided blocks, each of the blocks on the right side exhibits 8 connections that lead from its neurons directly to corresponding CMOS pads of the die (“direct out”). The direct inputs and outputs have been

[4]This mutual setup is primarily dictated by signal routing considerations.

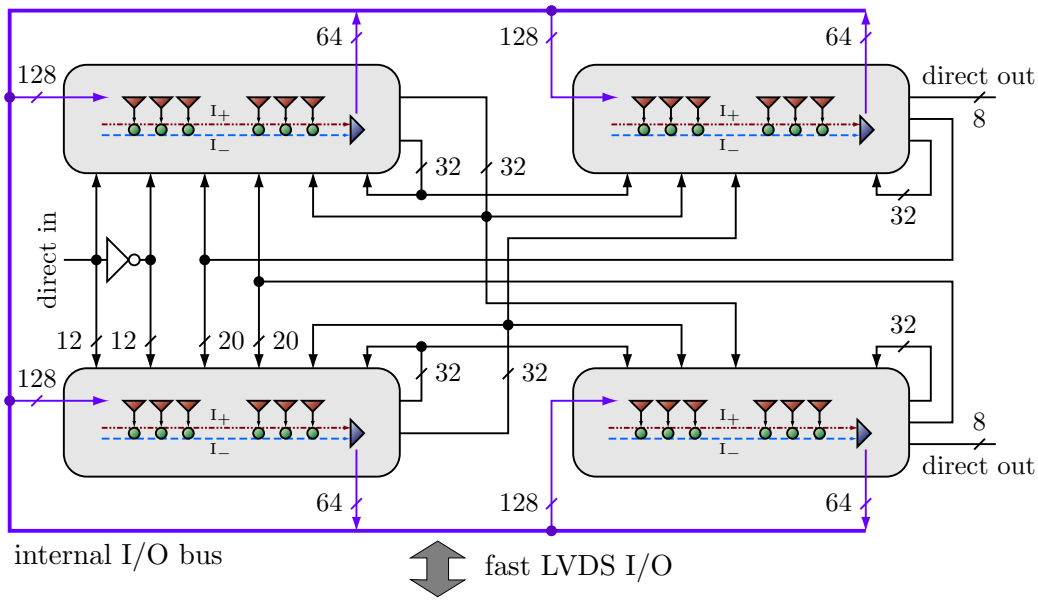


Figure 5.7: *Schematic of the hard-wired feedback and inter-block connections on the HAGEN prototype. The available routing capabilities promote a predominant data flow from left to right. A block of the left side can feed all of its outputs back to its own inputs as well as to the inputs of its counterpart on the right side. In total, each block on the right can receive up to 96 inputs from blocks on the left (after [179]).*

integrated to allow the HAGEN chip to be utilized for fast data communication applications.

In the presented design, every single input node can either receive ordinary input from the digital external bus or from exactly one fixed alternative source, i.e., either a dedicated neuron of the same block, a neuron of one of the other blocks, or one direct input. For each node, the choice of its actual input source is thus a binary decision. Consequently, in order to specify the complete feedback setup for one HAGEN chip, one bit per input node is sufficient (this topic will be returned to in section 6.2.2).

5.4.3 Weight Resolution and Dynamic Range

weight resolution

The on-die digital to analog converters that are responsible for the generation of the analog weight values have been implemented to yield a resolution of 10 bit. In connection with the sign of each weight value, this results in a total nominal weight resolution of 11 bit. When expressed in units of the least significant bit (LSB) of the DACs, the set $\mathbb{W}$ of possible weight values w is thus given by $w \in \mathbb{W} = \{-1023, -1022, \ldots, 1022, 1023\}$.

dynamic range

It is a notable feature of the implemented circuits that the postsynaptic current $\mathrm{I_{ps}}(\,|\,w\,|\,)$ which is eventually contributed to either the inhibitory or excitatory input line of the neuron is essentially identical to the current $\mathrm{I_{DAC}}(\,|\,w\,|\,)$ that is generated by the digital to analog converter to write the analog weight value

into the synapse [179] (possible deviations will be discussed in section 5.5). The effective dynamic range of the synapse currents is therefore determined by the dynamic range of the DACs and can be adjusted externally. The synapse implementation itself is designed for a maximum current $I_{ps}^{max} = I_{ps}(1023)$ of at most $50\,\mu A$ which corresponds to a resolution of approximately 50 nA. The comparator of the neuron circuit also achieves a resolution of 50 nA and exhibits a dynamic range of $\pm 300\,\mu A$ for each input line.

neuron saturation

Hence, if the maximum synapse current I_{ps}^{max} is set to the largest allowed value, six activated synapses with maximum weights that are all connected to either I_+ or I_- will cause the respective neuron input branch to reach saturation. This is deemed to be no severe restriction in practice: First, in common networks, synapses with maximum weight are expected to be the exception rather than the rule. Second, for most of the time, only parts of the neuron inputs will be activated simultaneously. Finally, chip-in-the-loop training algorithms can automatically account for these general restrictions and adjust the magnitudes of the weight values accordingly.

dynamic range and noise

Apart from that, I_{ps}^{max} is usually chosen to be smaller than the allowed maximum value. This is not only reasonable with respect to a better exploitation of the dynamic range of the neuron, but also yields a lower power consumption (section 5.3.3). On the other hand, changes in the dynamic range of the synapse current also affect its signal to noise ratio and thereby influence the effective resolution of the synaptic weights (see section 5.5.4).

5.4.4 Weight Configuration

A network block cannot be used for data processing while its weights are written. Therefore, a fast weight transfer is desired. Each of the respective DAC units that are located between the two opposing blocks of the upper and lower half (see figure 5.6) comprises of 8 single digital to analog converters. Each one of these DACs is employed for the conversion of the synaptic weights of $2 \cdot 8$ neurons, 8 in each of the two adjacent network blocks. This setup constitutes a compromise between the resulting area consumption and the maximum achievable weight update rate.

two stage setup

The analog values of the single weights are written into the synapse array according to a column-wise pattern. In order to retain a high transfer speed, a two stage structure is employed and combined with an alternating transfer scheme [179]: Every DAC has two banks of digital input latches that are capable of the full speed of the LVDS interface. While the DAC converts the weights for one block, the buffers for the other side can already be filled with the corresponding digital weight values. Furthermore, each DAC is accompanied by $2 \cdot 8$ analog storage units, 8 per side. Unlike the sign bits of the weights that are transferred directly into the synapses of the currently served column, the analog weight values are first written into these current memory units. Once all eight buffers are filled, their contents are finally being transferred into the respective synapses. Simultaneously, the DACs can fill the analog storage units of the opposite side.

The conversion time of a single digital to analog converter is 40 ns. With eight DACs, eight conversion cycles are necessary to fill the 64 current memories of one

side. The implemented circuitry can transfer the analog weights from the current storage units into the synapses within 320 ns. Together with the alternating scheme described above, this allows for a continuous DAC operation.

high configurational speed

In summary, the weights of all 32768 synapses of all four blocks on a HAGEN chip can be updated within 82 μs which corresponds to a maximum achievable CUPS value during training (section 2.4.3) of about 400 Megaweights/s. Using a weight refresh rate of 10 ms, the network speed is thus reduced by less than 1 %.

5.4.5 Performance and Scalability

high network speed

The network operates at a maximum frequency f_{net} of 50 MHz. Assuming that a given network configuration can fully exploit the parallel operation of one block, the resulting performance becomes

$$8192 \text{ synapses} \cdot 50\,\text{MHz} = 4.1{\cdot}10^{11}\,\text{CPS}. \tag{5.9}$$

comparison to standard microprocessors

A direct comparison to standard microprocessors is not unproblematic, since the utilized 0.35 μm process has been used by manufacturers in the years around 1995–1997. At the time of writing of this thesis, contemporary microprocessors of commercially available PCs have become strong competitors in terms of mere performance. Consider the exemplary case of a 3.6 GHz processor with SIMD (Single Instruction, Multiple Data), e.g. a 0.09 μm Pentium IV processor. If it is assumed that an optimum of eight 16-bit integer instructions can be performed in each clock-cycle and that the processor is used to simulate the operational principles of the HAGEN block, the resulting performance turns out to be

$$8 \text{ integer additions} \cdot 3.6\,\text{GHz} = 2.88{\cdot}10^{10}\,\text{CPS}. \tag{5.10}$$

It is understood that the 16 bit precision of the used integer representation is not the most efficient way to code the 11 bit weights of HAGEN. But even in terms of the achievable connection primitives per second (see section 2.4.3), the performance of a HAGEN block and the Pentium IV microprocessor become $4.5{\cdot}10^{12}$ CPPS and $4.6{\cdot}10^{11}$ CPPS, respectively.

low power consumption

Besides providing a gain in computational performance of about one order of magnitude, the second main advantage of the neural hardware approach is given by its superior power efficiency: While the peak power consumption of a Pentium IV is in the order of 100 W, a single HAGEN block reaches a maximum of only approximately 1 W [179] [178]. Achieving a two orders of magnitude lower power consumption particularly facilitates the efficient realization of large networks.

Admittedly, these are rather coarse calculations. The above considerations are primarily intended to convey a general feeling for the high potential of the concepts that underlie the desing of the HAGEN prototype. A thorough discussion of the computational performance and power consumption of the HAGEN ASIC lies beyond the scope of this thesis. A more detailed account of these topics can be found in [185].

feasible up-scaling

The utilized network model (equation 5.1) allows for a feasible up-scaling of the implemented networks not only by integrating more network blocks on future

ASICs or by increasing the size of the blocks itself. The digital chip interface in connection with the clocked network operation and the fast LVDS bus readily allows to distribute suitable neural network architectures over multiple interconnected HAGEN chips [52]. As long as it can be ensured that connections from network blocks on different ASICs always retain a time delay $\Delta t'$ that is an integer multiple of the time of one network cycle $\Delta t = 1/f_{\text{net}}$, the synchronous operation of the whole network is ensured (see equation 5.4). A corresponding hardware setup will briefly be introduced in section 6.3.

5.5 Network Calibration

motivation

The presented synapse and neuron circuit implementations ensure that the device mismatches introduced during manufacturing of the ASIC primarily lead to mere additive offsets in the analog operation [179]. Hence, the major part of the static variations can be compensated for by adequately modifying the desired weights before they are programmed into the synapse arrays. Within certain limitations, this even allows to train the networks off-chip. While for some applications, this might be a feasible or even necessary alternative to chip-in-the-loop training approaches, the main motivation to calibrate the static variations of the used network ASIC is the ability to transfer the obtained sets of weights from one chip to another without a severe loss in performance. In any case, detailed knowledge of the sources and magnitudes of the involved offsets is required.

5.5.1 Types of Fixed Pattern Offsets

Three types of static variations can be distinguished that affect the calculation of the network output:

Neuron offset The switching points of the single neurons do not consistently equal 0. In general, each neuron exhibits an individual threshold I_{noff} such that it will only be activated if $\mathrm{I}_{\text{pos}} - \mathrm{I}_{\text{neg}} \geq \mathrm{I}_{\text{noff}}$ and remain inactivated otherwise. These neuron offsets can be be positive or negative.

neuron calibration

An easy way to compensate for this deviation is to reserve one column of synapses per block for the neuron calibration: The corresponding input node is permanently activated and the synapses are each assigned an individual weight σ_{noff}, such that the resulting synaptic current equals the offset I_{noff} of their respective neuron. Regarding only the contributions I'_{pos} and I'_{neg} of the remaining synapses, the firing condition of a given neuron then becomes $\mathrm{I}'_{\text{pos}} - \mathrm{I}'_{\text{neg}} \geq 0$ as desired. Hence, while the calibration of the neuron offset is comparably unproblematic, it effectively reduces the available inputs of a block to 127 (or less, see section 5.5.4).

Weight generation error A second type of of static deviation has its origin in the special operation of the on-chip DACs (see sections 5.4.3 and 5.4.4). In order to speed up the transfer of the generated weight to the intermediate current memory,

a fixed offset has to be added to its analog value (for technical details see [179]). Dedicated circuitry is included in the weight generation unit that accounts for this effect by subtracting the temporary offset before the desired weight value is eventually written into the synapse. But due to the present device variations, this compensation is not entirely perfect, and a small deviation from the original weight remains. Although this effect is the result of an imprecise operation of the compensation circuit rather than being caused by the DAC itself, it shall in the following be denoted as DAC offset for simplicity.

The DAC offset can be positive or negative and its exact magnitude cannot reliably and exactly be predicted in advance. But since all synapses of a given neuron are served by the same DAC, they all suffer from the same, initially unknown offset $\Delta \mathrm{I}_{\mathrm{DAC}}$. If w is the desired weight expressed in LSB units, the actually generated current $\mathrm{I}_{\mathrm{DAC}}(|w|)$ that is written into the synapse via the current memory buffer (section 5.4.4) obeys

$$\mathrm{I}_{\mathrm{DAC}}(|w|) = J(|w|) + \Delta \mathrm{I}_{\mathrm{DAC}}. \tag{5.11}$$

Here, $J(w)$ is the part of the DAC output that is exclusively determined by the programmed weight and fulfills $J(0) = 0$.

Individual synapse offset In addition to the above deviation which is the same for all synapses of a given neuron, each single synapse exhibits its own individual offset due to charge injection [4] [179]. The postsynaptic current $\mathrm{I}_{\mathrm{ps}}(\,|\,w\,|\,)$ that is eventually contributed to either the inhibitory or excitatory input line of the neuron will differ from the generated value $\mathrm{I}_{\mathrm{DAC}}(|w|)$ (see equation 5.11) according to

$$\mathrm{I}_{\mathrm{ps}}(|w|) = \mathrm{I}_{\mathrm{DAC}}(|w|) + \Delta \mathrm{I}_{\mathrm{syn}}. \tag{5.12}$$

The offsets $\Delta \mathrm{I}_{\mathrm{syn}}$ are generally positive. Furthermore, considering all synapses of one specific neuron, the single values $\Delta \mathrm{I}_{\mathrm{syn}}$ can be described in terms of a row-specific mean offset $\Delta \mathrm{I}_{\mathrm{syn}}^{\mathrm{avg}}$ and an individual offset $\Delta \mathrm{I}_{\mathrm{syn}}^{\mathrm{ind}}$:

$$\mathrm{I}_{\mathrm{ps}}(|w|) = \mathrm{I}_{\mathrm{DAC}}(|w|) + \Delta \mathrm{I}_{\mathrm{syn}}^{\mathrm{avg}} + \Delta \mathrm{I}_{\mathrm{syn}}^{\mathrm{ind}} \tag{5.13}$$

While the DAC offset $\Delta \mathrm{I}_{\mathrm{DAC}}$ and the individual synapse offset $\Delta \mathrm{I}_{\mathrm{syn}}^{\mathrm{ind}}$ can be positive or negative, the absolute value of their sum is guaranteed to be smaller than the row-wise synapse offset mean, $|\Delta \mathrm{I}_{\mathrm{DAC}} + \Delta \mathrm{I}_{\mathrm{syn}}^{\mathrm{ind}}| < \Delta \mathrm{I}_{\mathrm{syn}}^{\mathrm{avg}}$. In other words, the resulting total deviation

$$\Delta \mathrm{I} = \mathrm{I}_{\mathrm{DAC}} + \Delta \mathrm{I}_{\mathrm{syn}}^{\mathrm{avg}} + \Delta \mathrm{I}_{\mathrm{syn}}^{\mathrm{ind}} > 0 \tag{5.14}$$

remains to be always positive. Depending on whether a synapse is connected to I_{+} or I_{-}, the excitatory or inhibitory effect of the synapse will be enlarged.

Regarding the different terms in equation 5.14, only the last summand differs between the individual synapses. The first two offsets are common to all synapses of a neuron and can therefore be combined to a row-specific mean offset $\Delta \mathrm{I}_{\mathrm{row}} = \mathrm{I}_{\mathrm{DAC}} + \Delta \mathrm{I}_{\mathrm{syn}}^{\mathrm{avg}}$. In summary, the final postsynaptic current can then be written in the form

$$\mathrm{I}_{\mathrm{ps}}(|w|) = J(|w|) + \Delta \mathrm{I}_{\mathrm{row}} + \Delta \mathrm{I}_{\mathrm{syn}}^{\mathrm{ind}}. \tag{5.15}$$

row-wise offset calibration

The HAGEN prototype provides special functionality for the compensation of any deviations that are common to all synapses of a given neuron. Additional dedicated circuitry is included in the weight storage setup that allows to store an externally defined offset which is then automatically subtracted from all weights that are being copied from the intermediate current memories into the final synaptic storage capacitances [179]. This circuitry can be used to balance the effect of the row-wise offsets ΔI_{row}, once their respective values have been determined.

synapse offset calibration

Furthermore, given that the magnitudes of all individual synapse offsets ΔI_{syn}^{ind} are known, they can in principle be compensated by subtracting corresponding values from the respective programmed weights w. Unlike in the cases of the neuron offsets and the row-specific deviations that can be accounted for by constant values and independently of the desired weights, the calibration of the individual synaptic deviations requires to readjust every single value in each new set of weights that is to be transferred into the synapse array. Such a procedure leads to unwanted additional costs in the weight generation and is preferably avoided.

limitations

Finally, since the weight storage capacitor of a given synapse cannot be charged with less than no current at all, subtracting the necessary offset from the absolute value $|w|$ of the desired weight must not yield a negative number. If this situation does occur, the weight value is reasonably set to zero. In other words, even if it is decided to calibrate the single synapse offsets individually, only those weights can reliably and accurately be stored in the synapse array whose absolute values exceed a given threshold that is determined by the magnitude of the synaptic fixed-pattern noise.

In either case, it is desired that the single synapse offsets are reasonably small and therefore do not compromise the achievable resolution of the weights too severely. In order to verify that the HAGEN prototype fulfills this requirement — and to actually allow for the calibration of the chip — it is necessary to devise means of determining the magnitudes of the discussed types of offsets. This will be the topic of the following sections.

5.5.2 Determining the Offset Values

The HAGEN ASIC does not provide any means for measuring the DAC output, the individual neuron thresholds, or the single weight values directly. The only information that is provided by the network is its response to a given input pattern. In the course of the work presented in this thesis, a calibration method has been developed that allows to deduce the values of the above fixed pattern offsets solely by evaluating how the neurons' responses to a specific input depend on the specified weight values.

basic setup

The involved measurements are based on the following basic procedure: The inputs of the examined block are deactivated entirely except for two selected nodes. For a given neuron, the two corresponding synapses are set to opposite weight values w and $-w$ which would ideally result in $I_{pos} = I_{neg}$. In the general case, however, the individual effective offsets of the two synapses will cause the resulting total neuron input to be different from zero and the offset of the neuron will have moved its threshold to negative or positive values. Consequently, the neuron may

either be activated or deactivated but will usually not be at its switching point.

pairwise synapse sweep

In order to find this point, the weight of one of the synapses — without loss of generality it shall be the excitatory one — is swept across a range centered around the original value w and large enough to include all variations. The inhibitory synapse is kept fixed at $-w$. It has to be borne in mind that the total static offset $\Delta\mathrm{I}$ of the varied synapse only affects the absolute value of the postsynaptic current and thus inverts its effect once the swept weight crosses zero. In order to avoid the resulting discontinuity, range and center of the sweep have to be chosen such that it only covers positive weight values and retains a sufficient distance from zero.

repeated measurement

Starting from the chosen minimum value, the sweep is done upwards in LSB units of the DAC. For each weight value, the network response is evaluated five times in order to account for analog noise in the system. If the neuron has fired three or more times, it is considered to have reached the threshold. The difference between the corresponding weight value and the original weight w is taken as the result of the measurement x.

Let the swept synapse be assigned the index i and the fixed reference synapse be labeled a. According to what has been stated in the previous section, the selected reference weight value w, the total individual offset currents of the two synapses $\Delta\mathrm{I}_i$ and $\Delta\mathrm{I}_a$, the neuron offset current $\mathrm{I}_{\mathrm{noff}}$, and the measured value x_i^a obey:

$$\mathrm{I}_{\mathrm{noff}} = -J(w) - \Delta\mathrm{I}_a + J(w + x_i^a) + \Delta\mathrm{I}_i \tag{5.16}$$

abstracted measurement

As a general convention, the subscripts and superscripts of x_i^a shall denote the swept and fixed synapse, respectively. For the calibration of the chip, the offset currents $\Delta\mathrm{I}_i$, $\Delta\mathrm{I}_a$, and $\mathrm{I}_{\mathrm{noff}}$ need to be determined in LSB units of the DACs. Therefore, the corresponding values σ_i, σ_a, and σ_{noff} shall be defined according to $J(\sigma_i) = \Delta\mathrm{I}_i$, $J(\sigma_a) = \Delta\mathrm{I}_a$, and $J(\sigma_{\mathrm{noff}}) = \mathrm{I}_{\mathrm{noff}}$. Furthermore, it will be assumed that the used digital to analog converters exhibit a sufficient linearity as to regard $J(x)$ as a linear function [179]. Equation 5.16 can then be simplified to become

$$\begin{aligned} \sigma_{\mathrm{noff}} &= -(w + \sigma_a) + w + \sigma_i + x_i^a \\ &= -\sigma_a + \sigma_i + x_i^a. \end{aligned} \tag{5.17}$$

Since the original weight value w cancels, equation 5.17 remains to contain only the measurand x_i^a as a known quantity.

exchanged roles

If the roles of the two synapses are exchanged, a second measurement yields the result x_a^i and one obtains the set of equations

$$-\sigma_a + \sigma_i + x_i^a = \sigma_{\mathrm{noff}} \tag{5.18}$$

$$-\sigma_i + \sigma_a + x_a^i = \sigma_{\mathrm{noff}} \tag{5.19}$$

which can be transformed into an expression for the neuron offset σ_{noff}

$$\sigma_{\mathrm{noff}} = \frac{x_i^a + x_a^i}{2}. \tag{5.20}$$

triple sweep

Apart from that, it turns out that pairwise measurements of the above type do not suffice to uniquely determine the weight offsets σ_a and σ_i. It is required to

involve a second reference synapse b and perform a slightly modified sweep of the synapse i where the inhibitory weight $-w$ is distributed over the two synapses a and b:

$$\begin{aligned}\sigma_{\text{noff}} &= -(\frac{w}{2}+\sigma_a) - (\frac{w}{2}+\sigma_b) + w + \sigma_i + x_i^{ab} \\ &= -\sigma_a - \sigma_b + \sigma_i + x_i^{ab}\end{aligned} \tag{5.21}$$

After a last pairwise measurement between the synapses b and a, one finally obtains a set of four linearly independent equations

$$-\sigma_a + \sigma_i + x_i^a = \sigma_{\text{noff}} \tag{5.22}$$
$$-\sigma_i + \sigma_a + x_a^i = \sigma_{\text{noff}} \tag{5.23}$$
$$-\sigma_a - \sigma_b + \sigma_i + x_i^{ab} = \sigma_{\text{noff}} \tag{5.24}$$
$$-\sigma_a + \sigma_b + x_b^a = \sigma_{\text{noff}} \tag{5.25}$$

that can easily be solved to express all four unknowns σ_a, σ_b, σ_i, and σ_{noff} in terms of the four measured values x_i^a, x_a^i, x_b^a, and x_i^{ab}. The expression for the neuron offset σ_{noff} has already been given in equation 5.20 and the offset of synapse i becomes:

$$\sigma_i = x_i^{ab} + x_b^a - 2x_i^a \tag{5.26}$$

Similar results can readily be derived for σ_a and σ_b.

row-wise offset and single synapse deviations

Each of the resulting values σ_i represents the combination of the row-specific offset $\Delta\text{I}_{\text{row}}$ and the individual effective synapse offset $\Delta\text{I}_{\text{syn}}^{\text{ind}}$. Therefore, it stands to reason to identify the mean of all measured values σ_i over a given row of the synapse array with the corresponding row-wise offset $\Delta\text{I}_{\text{row}}$. The individual deviations from this mean value are then regarded as to represent the individual offsets $\Delta\text{I}_{\text{syn}}^{\text{ind}}$ of the respective synapses.

5.5.3 Calibration Measurements and Results

measurement scheme

A set of measurements has been performed to quantify the magnitudes of the discussed deviations within two typical HAGEN chips. The sweeps 5.22–5.25 are repeated for every synapse i, $2 \leq i \leq 127$ of each neuron in the investigated network blocks. The synapse columns 1 and 128 are excluded to avoid edge effects. The reference synapses a and b are randomly chosen for each synapse i and it is ensured that the three involved synapses are not adjacent to each other in order to exclude any corruption of the results due to potential crosstalk. In practice, all synapses of a given column of the network block can be swept simultaneously.

Besides the individual offsets σ_i, the 126 measurements of the investigated synapses also yield a total of 126 values for the respective neuron offset. Their mean value is taken to be the final result σ_{noff}. The whole calibration procedure for one block is repeated 100 times in order to obtain information about the temporal fluctuations as well.

measurement parameters

Table 5.2 shows the results for two network blocks on two different HAGEN chips. The dynamic range of the synapses has been adjusted to yield a maximum

	chip 1	chip 2
neuron offset width (temporal mean of spatial rms of neurons)	74	73
synapse offset width (temporal mean of spatial rms of synapses)	2.4	2.5
synapse noise width (spatial mean of temporal rms of synapses)	2.2	2.3
row offset mean (mean of spatio-temporal mean of synapse rows)	10.7	12.4
row offset width (rms of spatio-temporal mean of synapse rows)	2.9	2.7

Table 5.2: *Results of the calibration measurements on two HAGEN chips. The values are given in LSB units of the used DACs.*

current $I_{ps}^{max} = 30\,\mu A$, and the chosen reference weight $w = 180\,\text{LSB}$ thus corresponds to $5.4\,\mu A$. The numbers in table 5.2 are given in LSB. The mean values of the neuron offsets averaged over each block are zero, and the same applies to the temporal fluctuations of the measured single synapse offsets.

results: neuron offset & synapse variations

In terms of the neuron input range of $\pm 300\,\mu A$, the measured width of the neuron offset distribution of approximately $74\,\text{LSB} \mathrel{\widehat{=}} 2.2\,\mu A$ is less than 1 %. The individual synaptic fixed pattern offsets turn out to be in the same order as the temporal noise (rows two and three of table 5.2). The latter is deemed to be dominated by crosstalk from the digital parts of the system [178].

It has already been stated in section 5.4.3, that the resolution of the neuron as well as the nominal resolutions of the DAC units and the weight storage are 50 nA. In this respect, the resulting sigma of the single synapse offset distribution of 75 nA and the observed temporal fluctuations of about 70 nA are satisfactory.

results: row-wise mean offset

The row-wise mean of the individual synapse offsets is obtained by averaging the offsets of all synapses of the corresponding neuron. This value is denoted as "row offset mean" and is averaged over all neurons of the respective block as well as over all 100 successive measurements (second last row in table 5.2). In agreement with what has already been stated in section 5.2, these mean offsets are observed to be larger than the single synapse variations. The resulting total synapse offset is always positive. Although, as expected, the row-wise averages of the synaptic offsets do vary between the individual neurons of a block, the variations are only small (last row of table 5.2) and comparable in magnitude to the synaptic fixed-pattern noise.

5.5.4 Calibration Practice

omitting the synapse calibration

It can be inferred from table 5.2 that an individual calibration of the single synapses is not to be considered essential since the static offsets are in the same order of magnitude as the inevitable analog noise, and it is therefore omitted in practice. Nevertheless, the observed fixed-pattern offsets and temporal fluctuations persist to ultimately limit the achievable accuracy of the weights. Given a synaptic dynamic range of $30\,\mu A$ like it is used for the measurements above, weight values can be specified with an uncertainty of only a few LSB. To limit the chips overall power consumption and to better exploit the dynamic range of the neuron, a lower dynamic range of the synapses I_{ps}^{max} might be desired. This would necessarily result in a decreased precision of the weight values when measured in

available dynamic range

LSB, and it has been discussed in section 5.5.1 that this particularly affects the feasibility of small absolute weight values (see also section 8.2.2).

assuring $w = 0$

In order to allow for an adequate realization of absent connections within the implemented networks, all synapses that are assigned a weight value of zero are switched off completely. Regardless of the state of the corresponding input node, these synapses do not contribute to the input current of the respective neuron. In other words, a weight value of zero can always be realized without any deviations.

Working with Calibrated HAGEN Chips

standard calibration measurement

In practice, the measuring scheme that is employed to determine the necessary calibration information for the used HAGEN chip is slightly different from the one described in section 5.5.3. The set of sweeps 5.22–5.25 is repeated five times for each synapse i and the five obtained values of σ_i are averaged to yield the final result. The neuron offsets and the row-specific averages of the synaptic offsets are calculated by taking the respective mean of all corresponding $5 \cdot 126$ measurements within the respective row, again excluding the first and the last synapse. This procedure is repeated for each block. For a given HAGEN ASIC, the whole described measurement needs to be conducted only once and the obtained data can henceforth be used for the calibration of this chip as desired.

edge effects

A closer investigation of the results reveals that in contrast to the last synaptic column, the first column of synapses (facing towards the DACs in the center of the ASIC) is not affected by edge effects and is therefore considered to be usable for the implementation of networks without problems. Excluding it from the calculation of the neuron and row-specific averaged synapse offsets is merely a precaution to err on the side of conservatism. In the case of the last column, edge effects measurably manifest themselves in the form of increased static offsets of about 40 % of the synaptic dynamic range. Hence, for safety, the 128th input node of each block commonly remains unused.

improved neuron offset calibration

According to the scheme described in section 5.5.2, the neuron offset σ_{noff} is measured in a setup where the two swept synapses are operated at potentially large weight values of approximately $-w$ and $w + \sigma_{\text{noff}}$. Due to small remaining nonlinearities of the neuron and the respective DAC [179], compensating the neuron offset by only a single weight with a comparably small value of $\sigma_{\text{noff}} - \sigma_i$ gives rise to slight but measurable deviations. Therefore, the proposed procedure for the neuron offset calibration (section 5.5.1) is modified in practice: The accuracy of the compensation is improved by using two permanently activated input nodes c_1 and c_2 instead of one. The first synaptic column is set to $-w + \sigma_{c_1}$ in order to yield the reference value w used during the offset measurement. The weights of the other column are set to $w - \sigma_{c_2} + y$, using the respective neuron offset σ_{noff} of the given row. This setup approximately reproduces the conditions of the measurement, thereby minimizing potential deteriorations due to present nonlinearities.

remaining resources

In summary, omitting the last column of each block to avoid edge effects and reserving two columns for the neuron calibration leaves a total of 125 usable inputs per block. By convention, columns 126 and 127 are employed for the compensation

remaining deviations

of the neuron offset. As described in section 5.5.1, the calibration of all row-wise synapse offsets is performed in hardware. The operation of a calibrated chip then deviates from the ideal model formulated in equation 5.4 merely by the limited dynamic range of the neurons, the slight nonlinearities of the implemented DACs [179], as well as the analog noise and the static individual synapse offsets. For typical values of the maximum synapse current $I_{ps}^{max} \approx 22\text{–}30\,\mu A$, the static and temporal weight variations are in the order of less than 1 %. The same applies to all potential deviations that are caused by slight imperfections of the used digital to analog converters [179].

Implications for Training and Reusability

off-chip training

It can be concluded that the use of calibrated HAGEN chips in principle allows for the utilization of suitable off-chip training approaches, provided that several precautions are taken. First, the theoretically calculated weight values have to be feasible also in the presence of random static and temporal variations in the order of up to 1 %. Second, the used training model has to account for the limited dynamic range of the neurons (section 5.4.3).

chip-in-the-loop training

While it is possible to incorporate these aspects into the formulation of dedicated off-chip training algorithms, it is evident that chip-in-the-loop approaches persist to be advantageous in so far as they automatically account for these peculiarities of the hardware substrate. Most notably, training the networks directly on the ASIC will favor systems that inherently exhibit an improved robustness against the present analog noise. From this point of view, the use of calibrated chips remains to be attractive mainly in one respect: the reusability of the trained networks on different ASICs.

improving chip transferability

Since the neuron offsets and the row-wise deviations are readily accounted for by the calibration, the main issue that remains to potentially impede the usability of a given set of weights on several different ASICs is given by the individual synaptic variations that differ from chip to chip. This problem can be overcome in at least three conceivable ways.

one-time calibration

- Although a compensation of the single synaptic deviations is not feasible during iterative training, the measured offset values can be used to modify the weights of an already trained network as to reduce their specialization to the chip they have been optimized on. In turn, before the resulting generic set of weights is used in another ASIC, it can once be corrected by the individually measured offset values of the target synapses.

re-training

- The magnitudes of the single synapse offsets are only in the order of 1 % of the total weight range. Therefore, if the network's performance shows a measurable deterioration after being transferred to another chip, it is reasonable to assume that the original quality can be restored by a short and cautious re-training.

- Similar to the robustness against temporal fluctuations that is automatically promoted by chip-in-the-loop approaches, the utilized network structures and employed training strategies could be optimized to yield networks that inherently exhibit the required insensitivity to spatial variations as well.

promoting inherent robustness

The best results can be expected from the combination of all three approaches. However, if successful, the last strategy is particularly attractive in so far as it potentially renders the first two procedures unnecessary. Moreover, aiming for neural networks that are robust against unreliable substrates by construction is not only motivated by the properties of biological neural systems. If feasible means of generating such networks could be devised, the underlying concepts might also be applicable to other systems, even those for which the first two solutions do not constitute available alternatives. It remains to be demonstrated that corresponding strategies for the construction and training of neural networks on the HAGEN ASIC can be found. A promising approach will be investigated in chapters 9 and 10.

Chapter 6

The Hardware Environment

Any technology distinguishable from magic is insufficiently advanced.

Gregory Benford

The preceeding chapter has introduced the HAGEN neural network chip as a feasible substrate for the flexible implementation of fast, massively parallel, and scalable neural networks in a low power, mixed-signal hardware. In order to be applicable to desired information-processing tasks, the ASIC itself needs to be embedded within a complete neurocomputer framework (see section 2.4.2) that provides the necessary functionality to interface and train the implemented networks. The following sections will describe the corresponding hardware environment that is used for all experiments presented in this thesis.

Due to its fast reconfigurability, the HAGEN ASIC is particularly well suited for highly iterative chip-in-the-loop training and it has been motivated in section 2.4.5 that model-free algorithms like, e.g., evolutionary strategies represent promising approaches. Still, from what has been discussed in chapters 3 and 4 it can be inferred that efficient evolutionary neural network training strategies persist to be a topic of research themselves. Hence, the flexibility of a hardware neural network framework to implement and evaluate different training algorithms constitutes an important aspect.

At the same time, the configurational speed and the fast network operation of the used HAGEN ASIC (see sections 5.4.4 and 5.4.5) can only efficiently be exploited during training if the algorithm itself is capable of generating new candidate solutions at a sufficient rate. Apart from that, it has been stated in section 2.4.5 that chip-in-the-loop training gives rise to considerable data transfer between the algorithm implementation and the neural network ASIC. Therefore, in order to best exploit the potential of the HAGEN chip also during training, both, the used algorithm and the required data transfer need to be realized as efficient as possible.

Among other things, the presented hardware setup features a specialized coprocessor architecture that is implemented in a configurable logic and allows to speed up the execution of evolutionary algorithms by performing the data inten-

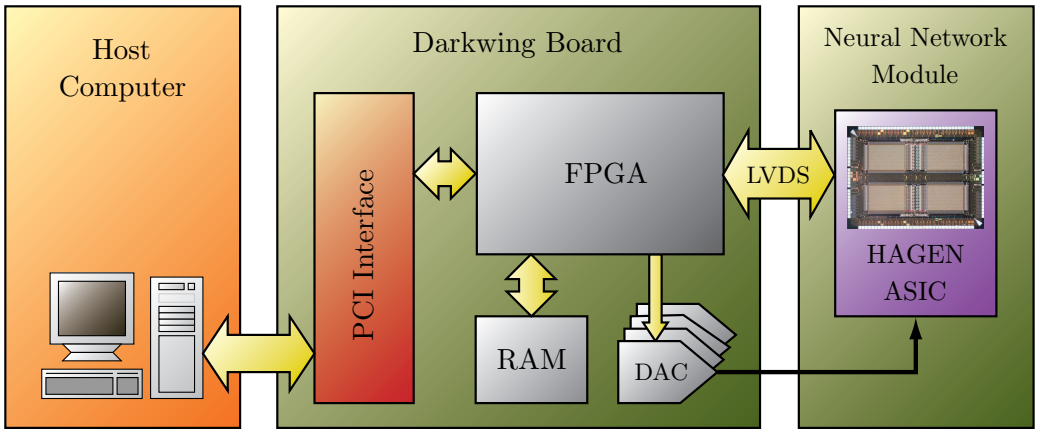


Figure 6.1: *Schematic illustration of the hardware setup that is used for all presented experiments. The used HAGEN ASIC is connected to an IBM compatible general purpose computer by the custom-built PCI-based FPGA adapter board called Darkwing [14].*

sive processing of the population's genetic material in a dedicated hardware. This so-called evolutionary coprocessor will be described in section 6.2 and can be programmed externally by an extensive instruction set. This allows for the remaining parts of the training algorithm to be implemented in software and executed on a common microprocessor, thereby reconciling the aim for flexibility with the desire for enhanced training speed.

Nevertheless, while the hardware setup employed for the experiments presented in this thesis reasonably accounts for efficiency and flexibility considerations, it does not yet fully exploit the potential of the described HAGEN ASIC — neither in terms of speed nor with regard to an efficient distribution of large networks over multiple chips. For this reason, an improved, modular hardware environment has been developed that overcomes these limitations. At the time of writing of this thesis, this setup is near completion. In several respects, the novel evolutionary training strategies that are introduced and evaluated in chapter 9 have been devised in anticipation of this advanced setup and it will therefore briefly be described at the end of this chapter.

6.1 The Used Hardware Framework

general setup

Figure 6.1 schematically illustrates the hardware environment that is used for the presented experiments. A single neural network ASIC is connected to a standard IBM compatible general purpose computer by a custom-built, PCI-based FPGA (Field Programmable Gate Array) card. This PCB (Printed Circuit Board) has been developed within the Electronic Vision(s) Group by Joachim Becker and is called Darkwing [14]. It is inserted into one of the available standard PCI slots of the host computer.

FPGA and PC

A central component of the used hardware setup is the programmable logic which is located on the Darkwing board. It serves as a digital controller for

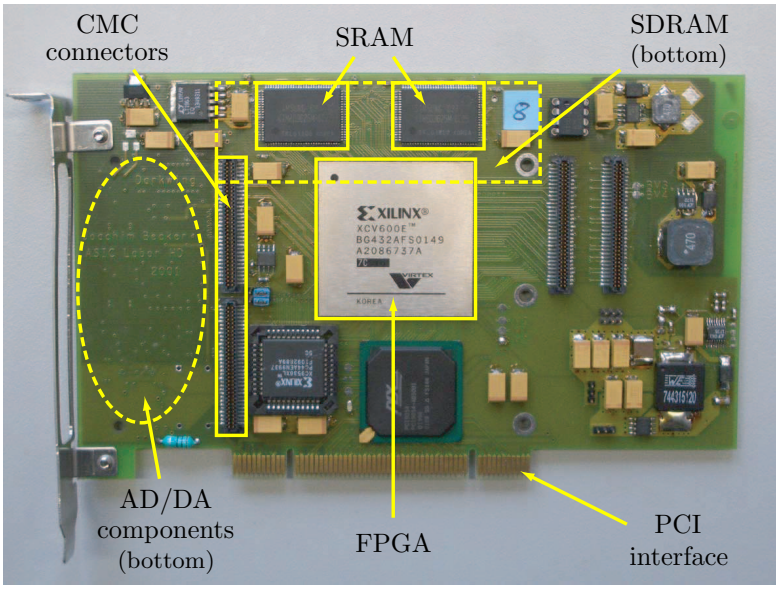


Figure 6.2: *Photograph of the top side of a Darkwing board (photograph by F. Schürmann [185] with kind permission). Apart from the central FPGA, Darkwing includes AD/DA conversion functionality, analog power supply for the connected ASICs, as well as local memory. The provided CMC connectors allow to add specialized adapter boards that are required to connect the different types of chip-carrier PCBs that host the actual mixed-signal ASICs (see also figure 6.3).*

the network chip and also provides its interface to the host PC. Furthermore, time-critical parts of the evolutionary chip-in-the-loop training algorithm can be migrated to the FPGA, as will be described in section 6.2. Alternatively, the training can also be implemented purely in software. In general, at least the higher-level parts of the training algorithm as well as the common user interaction with the connected HAGEN ASIC are realized in software and executed on the host PC (see also chapter 7).

general usability

Being designed as a suitable test system for the evaluation of different mixed-signal ASIC prototypes, the Darkwing Board is also used for other projects of the Electronic Vision(s) group, such as evolvable hardware experiments [125] [209] or the evaluation of an optical sensor with logarithmic response [27].

6.1.1 The Darkwing Board

Figure 6.2 shows a photograph of the top side of the Darkwing board. Apart from the central FPGA and the PCI interface to communicate with the host PC, Darkwing includes analog-to-digital (AD) as well as digital-to-analog (DA) conversion functionality, analog power supply for the connected neural network ASIC, and local memory (compare figure 6.1).

FPGA and PCI Interface

configuration

The Darkwing board is designed to host a commercially available Xilinx Virtex-E FPGA [229]. It is compatible with different speed grades of various existing types (XCV300E, XCV400E, and XCV600E). In order to allow for this flexibility, the FPGA is not configured at startup. Instead, the configuration data has to be provided by the host computer via the PCI interface. While a stand-alone PCI bridge (a PLX PCI 9054 [162]) interfaces the FPGA to the PCI bus, the actual configuration of the FPGA is assumed by a non-volatile programmable logic device (a Xilinx CPLD XC9536XL [233]).

responsibilities

Once configured, the FPGA assumes the operation of all remaining components, i.e., the on-board memory, the DA and AD converters, and the interface to the connected network chip. The FPGA configuration that implements this functionality has been programmed in the high-level hardware description language VHDL [49] (Very high speed integrated circuit Hardware Description Language).

PCI interface

The external PCI interface is a common 32 bit bus that operates at a frequency of 33 MHz, i.e., has a maximum bandwidth of 132 Mbyte/s. Since this is only about 10 % of the nominal bandwidth of the HAGEN chip [185] (see table 5.1), the FPGA needs an associated local memory that can be accessed at a sufficient rate to meet the interface requirements of the network ASIC.

Local Memory

possible configurations

Two types of local memory are provided: first, two static RAM (SRAM) chips with 1 Mbyte each and second, a removable synchronous dynamic RAM (SDRAM) of up to 256 Mbyte that can be inserted into a corresponding socket on the back side of the board. The smaller SRAM guarantees a fixed latency, while the larger SDRAM exhibits a varying latency. The different types of memory can only be used mutually exclusive. Which of the two RAM configurations will be eventually employed depends on the targeted application (see also section 6.2).

memory access

Within a typical chip-in-the-loop training setup, the primary purpose of the local memory is to provide fast access to the weights and input data as to allow for an operation of the network chip at a reasonable frequency (see also section 6.1.3). Apart from that, the RAM is required to store the network response that is read back after the execution of a network on the ASIC. A memory controller is implemented in the FPGA that enables the software on the host computer to transparently access this memory.

DA/AD Converters

Being designed as a test system for various kinds of mixed-signal ASICs, the Darkwing board readily supports analog signals. Since the FPGA itself is purely digital, analog-to-digital as well as digital-to-analog converters are included that allow the configurable logic to cope with analog input and output, respectively.

DA conversion

Two types of DACs are available. A pair of dual-channel 12-bit voltage DACs with a settling time of about 12 μs are used to generate slowly varying signals, like e.g., static bias voltages for the used HAGEN ASIC. Some applications require

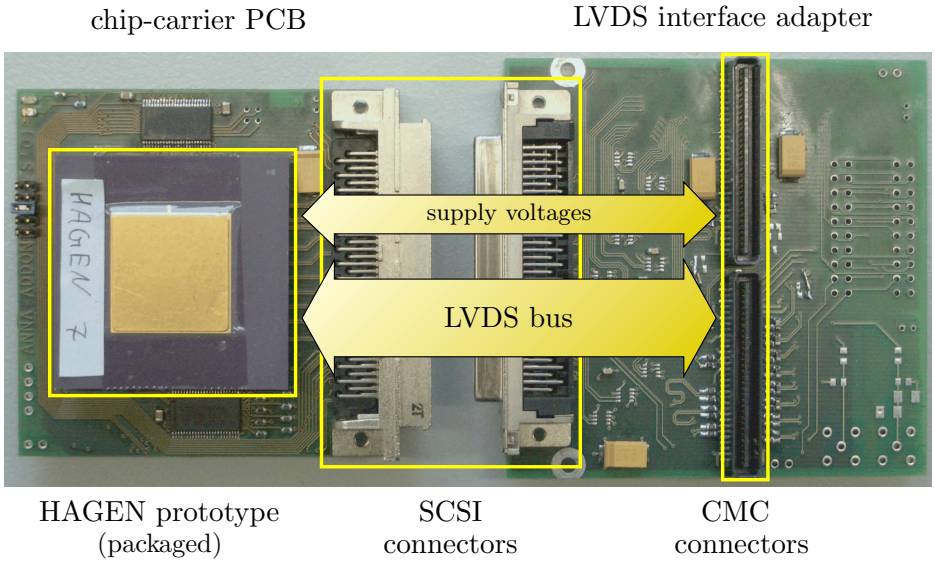


Figure 6.3: *Photograph of the HAGEN carrier PCB and the LVDS interface adapter board (photograph by F. Schürmann [185] with kind permission). The LVDS extension board gets connected to the Darkwing card via the shown CMC connectors. Apart from the LVDS signals, the SCSI interface to the chip-carrier PCB also conveys the digital and analog supply voltages for the HAGEN ASIC (see text).*

a faster generation of analog values and therefore, one 16-bit current DAC is included on the board that has a conversion time of only 25 ns. For the operation of the HAGEN ASIC, this DAC remains unused.

AD conversion

Besides that, a 12-bit ADC (Analog to Digital Converter) with a sampling rate of 40 MHz allows to perform the necessary analog-to-digital conversions whenever analog signals need to be connected to the FPGA. With the inputs and outputs of the used HAGEN chip being entirely digital, no AD conversion is required for its operation. In the employed setup, the ADC thus remains unused as well.

Analog Power Supply

The primary power supply for the Darkwing board is provided by the PCI interface (3.3 V, 5 V, and ±12 V are available). These power supplies are shared between all components of the PC and are electrically noisy. Besides special power supplies that meet the requirements of the various digital components, the board includes extra power supplies with adequate blocking capacitors that serve its analog components. The DACs and the ADC are fed by an analog 5 V power supply that can also be used for the connected ASIC.

Peripheral Bus and Connection to the HAGEN chip

connecting ASICs

The employed Xilinx FPGAs provide enough I/O capabilities not only to manage the local memory, serve the PCI interface, and control the AD/DA converters, but

also — and most importantly — to serve the peripheral bus to the connected neural network chip. Dedicated CMC (Common Mezzanine Card) connectors allow to connect special PCB extensions to the Darkwing board that translate its generic interface to the specific hot-plug interface of the particular chip-carrier PCBs that host the different mixed-signal ASICs.

connecting HAGEN

A photograph of both, the HAGEN carrier PCB and the special LVDS interface adapter that forms the corresponding extension to the Darkwing board is shown in figure 6.3. Due to the generic features that are already provided by the Darkwing card itself, the functionality of the chip-carrier PCB is mainly reduced to passive electronics and mechanical adapters. In the case of the HAGEN ASIC, the connectors of the carrier PCB and the interface adapter are mechanically compatible with the SCSI Parallel Interface connector 2 type P [159]. Electrically, the pin configuration is adapted to suit the purpose of interfacing to the HAGEN prototype (for details see [185]). In addition to the necessary supply voltages, the interface mediates 16 bi-directional and 5 uni-directional LVDS links that can be operated at up to 300 MHz. Voltage regulators on the carrier PCB derive the required analog voltage of 3.3 V (see table 5.1) from the 5 V that are provided by the Darkwing board.

6.1.2 The Host Computer

Within the described hardware configuration, the Darkwing card is inserted into one of the PCI extensions slots of the used IBM compatible general purpose computer. Since contemporary PCs provide multiple slots, this in principle allows to operate several boards simultaneously in one machine.

operating system

At present, the used host computers execute a Linux operating system with kernel 2.4.x. As it will be discussed in section 7.1.4, the software that is employed for the operation and training of the HAGEN chip is largely platform independent and is deemed to be easily portable to other operating systems as well.

used setup

For the various investigations within the Electronic Vision(s) group, a total of 11 similar configurations are currently in operation that include either Intel Pentium IV or AMD Athlon XP processors. The three setups that are used for the presented experiments feature Intel Pentium IV CPUs with 2.4 GHz. One single Darkwing board is connected to each computer.

6.1.3 Common Chip-in-the-Loop Operation

basic scheme of iteration

In general, the following three basic steps are iterated during a common chip-in-the-loop training process (see also figure 2.11): First, the algorithm generates a new candidate solution, i.e., a new network configuration for the HAGEN ASIC that is to be used to solve the task in question. Besides the individual weight values, the unambiguous definition of a network on the HAGEN chip requires the specification of the activated feedback and inter-block connections as well as the number of network cycles that are operated for each applied input pattern (see section 5.2).

Second, the generated network configuration is sent to the HAGEN ASIC for implementation and the desired set of input patterns is applied successively. After the network has finished processing all its input patterns, the resulting network responses are stored in the local memory of the FPGA such that, third, the algorithm can read back this data for evaluation. As long as the observed performance of the evaluated network is not satisfactory — and all potential additional termination conditions remain unfulfilled — the algorithm proceeds with the next iteration, i.e., starts by generating a new candidate solution, etc.

Data Handling

required data transfer

If the training algorithm is purely implemented in software, all parts of the network configuration that potentially change between the single candidate solutions (e.g., the weight values) need to be retransmitted from the host PC over the PCI bus to the local memory of the FPGA for each tested network. The same applies to the network response that needs to be read back from the memory of the FPGA into the RAM of the host computer for evaluation.

input data handling

In contrast, the input patterns that are applied to the single candidate solutions during training are usually the same for all networks. Within the experiments presented in part III, these input patterns are therefore transferred to the local memory of the FPGA only once at the beginning of a given training run and remain there to be applied to all tested candidate networks.

Speed Considerations

HAGEN interface speed

In the used setup, the HAGEN chip can be operated at frequencies of up to 100 MHz which yields a maximum bandwidth of the data transfer between the FPGA and the HAGEN ASIC of about 500 Mbyte/s [185]. This is about 4 times the nominal maximum speed of the PCI bus (see section 6.1.1). If the complete network specification and the desired input patterns are available in the local memory, an exemplary network that requires two network cycles and utilizes all 33k synapses in all four blocks can be loaded into the HAGEN chip and process 1000 input patterns within less than 1 ms.

PCI bottleneck

Using a software implemented algorithm, the speed of the system during training is limited by the considerable amount of data that has to be transmitted via the PCI bus from the host PC to the local memory on the Darkwing board or vice versa. According to what has been stated above, this primarily refers to the weight values and the output data of the network.

In order to avoid this bottleneck, it suggests itself to at least partly implement the generation of new candidate solutions and/or the evaluation of the network outputs within the FPGA and thereby to cut down the data that needs to be exchanged with the host computer. Ultimately, it would be desirable to entirely avoid any significant data transfer over the PCI bus, e.g., by reducing the parts of the training algorithm that need to be executed on the host PC to mere controlling and user interaction. A feasible approach is described in the next section.

6.2 The Evolutionary Coprocessor

During the experiments presented in this thesis, the HAGEN chip is trained with evolutionary chip-in-the-loop algorithms (see chapters 3 and 4). Apart from the fitness calculation that needs to be performed for each new individual in every generation, one of the most time-critical aspects of evolutionary optimization is the processing of the genetic material in the population. Using a generational replacement scheme (see section 3.4.1), a whole generation of new genotypes needs to be created from the genomes of the previous population in each iteration.

data-intensive genetic operations

Depending on the number of free parameters, the creation of offspring can involve large amounts of data. For example, even if only the weights of half of the synapses in one network block of the HAGEN ASIC were to be optimized, a single genome would still contain 4096 genes. On the other hand, given a desired set of mutation and recombination operators, all pairs of mating individuals are processed in a similar way, and all genes within the genome are usually regarded as equivalent. Tasks that require a similar set of simple operations to be repeatedly performed on large sequences of data are particularly well suited for a pipelined implementation in a specialized hardware.

genetic operations in hardware

This section introduces a dedicated coprocessor architecture that speeds up evolutionary training algorithms by performing the required genetic variation operations within a configurable logic. This so-called evolutionary coprocessor is coded in VHDL [49] and has been designed by Tillmann Schmitz [182]. A detailed description of the coprocessor will be given in Tillmann Schmitz' PhD thesis[1]. The following sections confine themselves with discussing those aspects that are relevant for the work presented in this thesis.

6.2.1 Coprocessor Setup Overview

general setup

In the described hardware setup, the coprocessor resides within the FPGA of the Darkwing board. In order to retain a sufficient flexibility in the realizable evolutionary training approaches, the remaining parts of the training algorithm persist to be implemented in software and are executed on the microprocessor of the host computer. At the same time, the coprocessor completely takes over the management and processing of the genomes in the evolved population and is controlled by the software via a given set of instructions. Figure 6.4 schematically illustrates the data flow within this setup.

The genomes that are being processed by the coprocessor are stored within the local memory of the FPGA. In order to allow for reasonably sized populations of genomes of the required size, the used hardware setup employs a 64 Mbyte SDRAM module that is inserted into the corresponding socket of the Darkwing board (see section 6.1.1).

advantages

Since the genetic material that defines a given candidate solution is stored within the local memory, the evaluation of individual networks on the HAGEN ASIC does not require this configuration data to be transmitted over the PCI bus. In addition to the acceleration of the genetic operations, this yields a substantial speed gain

[1] At the time of this writing, the mentioned thesis has not yet been finalized.

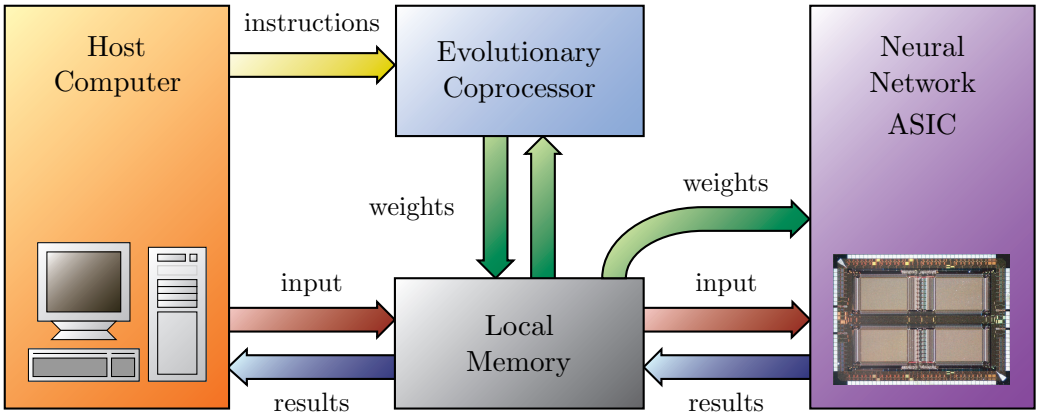


Figure 6.4: *Schematical overview of the evolutionary coprocessor setup (after [182]). The genetic material is stored within the local memory of the FPGA. It is managed and processed exclusively by the coprocessor which in turn is controlled by the training software via an extensive set of instructions.*

in comparison to a pure software implementation of the training algorithm that needs to be executed on the host computer (see also section 10.1).

6.2.2 Genetic Representation and Translation

According to the terminology introduced in section 3.4.2, the coprocessor employs an integer representation and implements a direct encoding scheme. In correspondence to the nominal resolution of weight values on the HAGEN prototype, one gene is expressed with 11 bit resolution. Nevertheless, in order to simplify the addressing of the genes in the SDRAM, each gene occupies two bytes. The remaining 5 bit are used internally to control the operation of the coprocessor (see below).

integer representation

The single genes are linearly arranged to chromosomes that are stored consecutively in the used SDRAM. Following the considerations brought forward in section 3.4.4, complete genomes can be formed from multiple chromosomes. In principle, the coprocessor can readily process chromosomes of variable length and genomes with arbitrary numbers of chromosomes.

genome structure

In order to fully benefit from the acceleration that is achieved by performing the genetic operations in a specialized hardware, the translation of the processed genotypes into valid network configurations for the HAGEN ASIC is implemented within the FPGA as well. In its current version, the translation unit only supports a direct decoding of genes into corresponding weight values: A valid chromosome needs to contain one gene for the weight of each of the 128 synapses that lead to one specific neuron on the HAGEN chip. A complete weight configuration for the ASIC is then defined by a genome of $4 \cdot 64$ chromosomes (a thorough description of the genetic representation that is used for the presented experiments will be given in section 8.3.1).

direct encoding

Architecture Specification

fixed architecture

Apart from the mere weight values, the additional parts of a complete chip configuration that are required to uniquely define the architecture of the final network — i.e., the activated feedback and inter-block connections in combination with the used number of network cycles — can, as yet, not be coded within the genome. Hence, in the current setup, the potential of the coprocessor can only be exploited if the architecture of all networks is fixed during training. At the beginning of the training run, the specification of the corresponding parameters can then be transferred to the local memory of the FPGA in order to be used for all individuals that will be evaluated during the following simulated evolution. Otherwise, the individual architecture specification for each new candidate solution has to be transferred from the host PC which significantly impedes a fast execution of the training procedure.

determining the architecture

Given a desired architecture, not all neurons and synapses of the HAGEN ASIC are necessarily used for the actually implemented network. Those genes of a chromosome that correspond to unused synapses of the coded neuron are reasonably set to zero. The coprocessor allows to mark single genes as deactivated such that they are not affected by mutation (internally, this involves one of the 5 spare bits in each gene, see below). In other words, any genes that are set to zero and are also marked as deactivated within all genomes of the initial population, will remain zero in all individuals throughout the whole evolution. In combination with the fixed feedback and inter-block connections as well as the specified number of network cycles, this allows to optimize the weight values of networks with any desired architecture that can be implemented on the HAGEN ASIC.

6.2.3 Pipeline Operation Overview

The design of the coprocessor pursues two aims: First, it needs to process a high throughput of genetic data. Second, it is desired to allow for a wide range of conceivable evolutionary algorithms to be implemented without changing the hardware configuration. In consideration of these requirements, the coprocessor is implemented as a pipeline whose single stages can be managed and controlled by a given set of instructions.

processing scheme

A simplified schematic of this pipeline is shown in figure 6.5. Given the addresses of the two parent chromosomes “parent 1” and “parent 2” in the local memory, the respective genetic material is fed through the pipeline successively. According to the instructions that control each stage of the pipeline, the offspring chromosome is constructed gene by gene and is written to a specified target address.

processing speed

In fact, the pipeline is designed fourfold parallel, i.e., four genes can be processed in each clock cycle. At a clock frequency of 80 MHz, this leads to a nominal maximum rate of $320 \cdot 10^6$ processed genes per second. However, the following description will only assume a single pipeline for simplicity; the generalization to four pipelines is straight forward.

Each of the single stages shown in figure 6.5 involves numerous decisions that are partly random and can partly be controlled via instructions. For the random

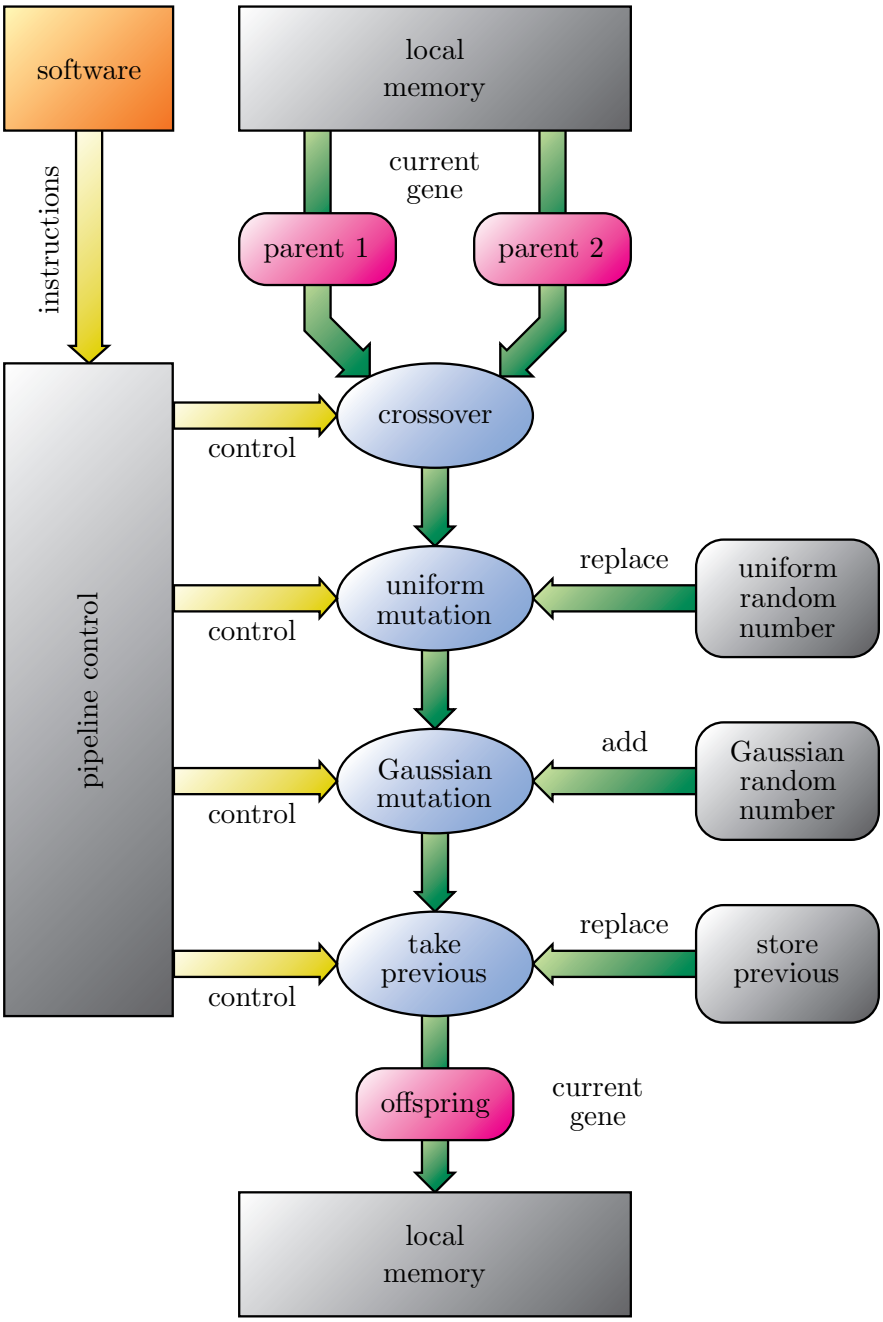


Figure 6.5: *Simplified schematic of the coprocessor pipeline (after [182]). The genetic data of the two parent chromosomes is taken from the local memory and is fed through the pipeline gene by gene. The operation of the single pipeline stages can be controlled by the software via an extensive set of instructions. In the first stage, it is decided from which of the two parents the current gene is to be taken. The following two stages perform a uniform and a Gaussian mutation, respectively. The last pipeline stage, finally, allows to ignore the result of the previous steps and rather set the gene to half the value of its predecessor (see also section 8.3.1). The finally obtained gene is appended to the offspring chromosome and is written to the specified address in the local RAM.*

decisions, a random number generator is implemented that produces a pseudo-random bit sequence. Considering the specification and handling of instructions, a more detailed discussion will be given in sections 6.2.4 and 6.2.5. For now, the processing scheme of the pipeline is to be described on a general level.

pipeline stages

The first stage implements the recombination step. Here, it is determined from which of the two parents the current gene g_i is to be taken. In the second stage, it is tested whether a uniform mutation is applied. If this is the case, the result of the first stage is replaced by a new random value that is uniformly distributed within a previously specified range. Otherwise, the outcome of the first stage remains unchanged. In any case, the gene is passed to the third stage where it is potentially made subject to a Gaussian mutation. If it turns out that this form of mutation is indeed to be applied, a random number is added to the original value which obeys a normal distribution of zero mean and a selectable width σ_r (if the result exceeds the allowed range, it is set to the respective maximum or minimum value).

The last stage, finally, allows to entirely ignore the results of the first three stages and to use another value instead that is given by half of the allele value of the previously processed gene $g_{i-1}/2$. This third stage has been incorporated in anticipation of the special requirements of the experiments presented in part III. The purpose of this procedure will be illuminated in section 8.3.1.

resulting data

The eventually obtained gene value is transferred to the desired address in the local memory that corresponds to the respective ith position in the child chromosome. If the recombination process is desired to yield two complementary offspring, the two parent chromosomes can be processed a second time where the original decisions of the first stage are simply inverted (compare figure 3.4).

6.2.4 Pipeline Control

In order to allow for a large variety of evolutionary algorithms to be implemented, the decisions that have to be made at each stage of the pipeline can be controlled by the software part via the specification of different control options. In general, two types of variables can be distinguished that affect the operation of the pipeline.

global parameters

The first kind is of global nature and is assumed to be constant during the whole evolution run or at least for several successive generations. For example, this refers to the mutation rate or the choice of the applied mutation operator(s).

gene-specific decisions

In contrast, the actual decision from which parent a given gene is to be taken or whether it is mutated or not is typically made for each gene position separately. Consequently, the second kind of control option changes from gene to gene. Partly, these gene-specific decisions will involve the random numbers that are provided by the mentioned random number generator. Another source of input that can be used to influence the operation of each stage are the states of the 5 spare bits that accompany every gene. For example — as it has been discussed above — one of these flags is used to switch off the mutation for single genes entirely. Finally, for the recombination stage, a binary crossover mask of the appropriate length can be provided by the software part that specifies the desired parent chromosome for

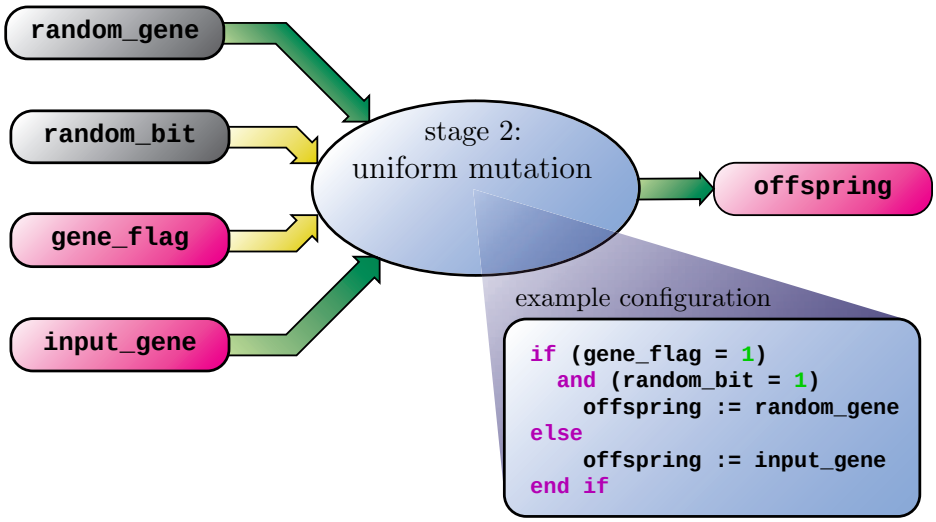


Figure 6.6: *A simple exemplary configuration for the uniform mutation stage. The considered gene is mutated only if both, the corresponding gene flag and the current random bit assume a state of 1. The probability for the random bit to be 1 is given by the specified mutation rate. The remaining gene flags and the mask bit that is provided by the software are ignored. This decision rule can be coded in the form of the lookup table shown in table 6.1.*

each individual gene position[2].

stage configuration

It remains that given the specified global parameters, the 5 gene-specific flags, the current bit of the provided binary mask, and the current random number(s), the outcome of each stage needs to be determined according to some desired rule. Unlike the gene-specific input parameters themselves, these rules can reasonably be assumed to remain unchanged during extended periods of the evolution procedure, the whole run, or even several successive simulated evolutions. Therefore, they can feasibly be specified by the software without compromising the speed of the pipeline operation.

example configuration

A simple example for a possible configuration of the second stage (uniform mutation) is illustrated in figure 6.6. For each gene position, one new random bit is generated that assumes a state of 1 with the specified probability for uniform mutation. If for a given gene, the mutation is not turned off entirely (i.e., the respective gene flag is not 0) it is mutated whenever the current random bit is 1. For this simple exemplary procedure, the provided binary mask and the remaining gene flags are readily ignored. More complicated procedures might take this additional information into consideration. In particular, most configurations of the crossover stage are likely to incorporate the binary mask that is provided by the software.

[2] In fact, two masks can be provided by the software that can in principle be used for different purposes. For the presented experiments, only one mask is used and exclusively affects the crossover stage. For more details see the technical publication [182] or the aforementioned PhD thesis of Tillmann Schmitz.

random bit	first gene flag	mask bit	remaining gene flags	do mutate
0	0	X	X	0
0	1	X	X	0
1	0	X	X	0
1	1	X	X	1

Table 6.1: *The lookup table that specifies the decision rule for the exemplary configuration of the uniform mutation stage shown in figure 6.6. An "X" in the table denotes "don't care". Only the current random bit and one of the gene flags are considered. The remaining gene flags as well as the binary mask that is provided by the software have no influence on whether the current gene is mutated or not. More complicated decision rules might incorporate this additional information. By specifying an individual lookup table for each stage of the pipeline, a large variety of genetic operators can be implemented.*

stage-specific lookup tables

In any case, the decision rule for each stage can be represented by a corresponding lookup table (LUT). The lookup table that defines the simple decision rule shown in figure 6.6 is given by table 6.1. By providing an according lookup table for each stage of the pipeline, the software can configure the coprocessor to implement a large diversity of conceivable variation operators.

6.2.5 Instruction Handling

Against the background of what has been said in the preceeding sections, the set of instructions that allows the software part of the training algorithm to control the operation of the coprocessor can be divided into four groups:

stage configuration

- Instructions that define the decision rule for each stage of the pipeline by providing an according lookup table. Since these configurations define the actually implemented genetic operators, they are usually only specified once per evolution run or even remain unchanged for several successive runs. In fact, for the described experiments, the desired operation of each stage is hard coded, i.e., the predefined functionality is not changed by the software at all (the used genetic operators will be introduced explicitly in section 8.3.1).

data transfer

- Data transfer instructions control the reading and writing of genetic data from and to the local memory of the coprocessor. Whole genomes as well as any sequence of genes may be transferred. Typically, these instructions are used at the beginning and the end of the simulated evolution to define the initial population and to read back the training result (e.g., for visualization or long-term storage), respectively.

global parameters

- Global parameter instructions allow to specify the values of all evolution parameters that are assumed to change only infrequently, e.g., the mutation rates for uniform and Gaussian mutation or the desired width of the Gaussian distribution for the latter.

- A given set of recombination instructions is used to define the addresses of the two parent chromosomes, their respective lengths, the target address for the offspring, and the desired crossover mask. Typically, a new corresponding set of information is provided for each recombination. *recombination*

Using a control software that is executed on the host PC, these instructions need to be transferred via the PCI bus. The first three types of instructions are uncritical in terms of speed since they are only used infrequently. In contrast, the recombination instructions need to be issued for each mating process. Considering genomes with N chromosomes, a total of $2N$ parent addresses, N chromosome lengths, N crossover masks, and N target addresses need to be specified for each crossover procedure.

For an efficient handling of these instructions, a dedicated data and instruction buffer is provided that can be utilized to pool instructions and store frequently used data. After an entire list of instructions has been written into this buffer, a corresponding start command transmits all instructions to the coprocessor and initiates the execution of the whole sequence. During the following operation of the coprocessor, the software can address other tasks (e.g., calculating the fitness of previously evaluated individuals). *instruction buffer*

Parts of the instruction buffer are reserved to allow for the generation of an internal address table. Frequently used chromosome addresses and/or chromosome lengths can be stored in this list such that instead of specifying all source and target addresses directly, the corresponding instructions merely need to include the index of the respective information in the internal address list. Using only a fixed set of addresses for all individuals that are created throughout the whole evolution run, this indirect addressing scheme effectively reduces the amount of data that needs to be exchanged between the software and the coprocessor during training. In principle, the same indirect addressing can also be used to repeatedly switch between different lookup tables for the individual stages of the pipeline. Within the current state of the setup, the buffer allows to store a total of 2048 instructions, chromosome address, and/or chromosome lengths. *data buffer*

The evolutionary coprocessor is subject to constant development and improvement. Since its original version, several advanced features have been added that provide more efficient ways of transmitting recombination instructions. For example, having specified the number, addresses, and lengths of multiple chromosome pairs, special loop functionality allows to iteratively apply predefined recombination operators, like, e.g., one-point crossover, to the whole set of chromosomes via effectively only one single instruction. New random crossover masks are created automatically and autonomously by the coprocessor for each mating pair. The usage of such pre-set recombination operators further minimizes the amount of data and instructions that needs to be exchanged between the software and the coprocessor. However, this novel functionality has not yet been employed for the experiments presented in this thesis. The detailed description of these additional features is therefore deferred to later publications. *advanced features*

6.2.6 The Evolutionary Coprocessor: Reflection and Outlook

The advantages of using the evolutionary coprocessor for the training of networks on the HAGEN chip are twofold: First, performing the required genetic operations in a dedicated hardware can potentially yield a significant speed gain compared to pure software solutions, especially for large genomes. Second, having the genetic material of the whole population available in the local memory allows to evaluate the different candidate solutions without the need to repeatedly transmit large amounts of configuration data over the PCI bus.

fitness calculation in hardware

In order to further reduce the required data transfer between the host PC and the memory of the coprocessor, the calculation of the fitness could be implemented in the FPGA as well. In the current state of the system where suitable training algorithms and appropriate fitness functions are a topic of research themselves, the fitness calculation remains to be realized in software. A hardware implementation of the fitness calculation is planned for future investigations.

limited complexity

Apart from that, the coprocessor persists to impose certain restrictions on the complexity of the realizable genetic operators. The present version can only be used to evolve the synaptic weights of networks with a fixed architecture using a direct encoding scheme, an 11 bit integer representation, and two simple single-gene mutation operators.

implications for training

More complex encoding schemes and operators could in principle be realized by processing the same set of chromosomes multiple times using different configurations for the single stages of the pipeline. Alternatively, the functionality of the translation unit could be improved to allow for more abstracted genotypic representations. In both cases, the increase in complexity would most likely be paid for by a reduction in training speed. Also with regard to the fitness function, it can thus generally be said that in order to fully benefit from hardware acceleration and to be capable of keeping up pace with the used neural network hardware, all time-critical parts of the evolutionary training algorithm are preferably kept simple. While the next section introduces an advanced hardware environment that successfully avoids some of the drawbacks of the current setup, the above considerations will turn out to hold also for this improved system.

6.3 An Advanced Hardware Environment

By confining the analog network operation into blocks with an entirely digital interface, the employed network model of the mixed-signal HAGEN prototype lays the foundation for a feasible up-scaling to larger networks (see sections 5.2 and 5.4.5). Apart from including more blocks in future ASICs, this can also be achieved by interconnecting a desired number of the existing HAGEN chips.

scalability considerations

Regarding the hardware infrastructure that has been described in the preceeding sections, multiple Darkwing boards could be used in the same host computer to operate several HAGEN prototypes simultaneously. However, if the different chips are to be interconnected to form one enlarged network, considerable amounts of data will have to be transmitted isynchronously from one ASIC to the other via the PCI bus (see also equation 5.4). Apart from the fact that the latency of a

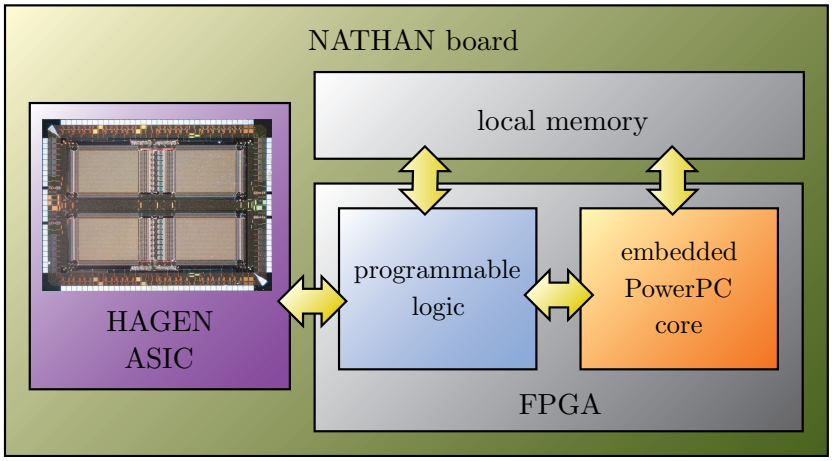


Figure 6.7: *The NATHAN board integrates all parts of the previously presented setup on one single PCB: a neural network ASIC, a configurable logic, local memory, and an embedded PowerPC microprocessor core. This way, any time-consuming data transfer over the PCI bus during training can efficiently be avoided.*

PCI transfer is not guaranteed, this is bound to significantly reduce speed of the network operation.

6.3.1 The NATHAN Board

In order to allow for an efficient interconnection of multiple network chips, an advanced hardware environment[3] has been developed that can successfully avoid the bottleneck of the PCI bus. The central part of this improved setup is a custom-built FPGA board called NATHAN that has been developed by Andreas Grübl [78].

autonomous modules

A single NATHAN module integrates all parts of the previously described setup on one PCB (see figures 6.7 and 6.8): In addition to the used neural network ASIC, a configurable logic, and local memory, each board includes a microprocessor that can execute the required control and training software. This is made possible by the utilization of a Xilinx Virtex-II pro [230] that hosts an embedded PowerPC 405 core [232] [231]. This PowerPC microprocessor runs an embedded Linux operating system [193]. Due to the special care that has been taken to keep the used software environment largely platform independent (see section 7.1.4), it can readily be utilized also in this setup.

For the local memory, NATHAN includes two SRAM units with 512 kbyte each as well as a socket for a removable double data rate SDRAM (DDR-SDRAM) module of up to 1 Gbyte. A synchronous LVDS interface of up to 32 links is used to communicate with the hosted HAGEN chip, and additional extension connectors even allow to connect special daughter boards that might host future network

[3]This work is supported in part by the European Union under the grant no. IST-2001-34712 (Sensemaker).

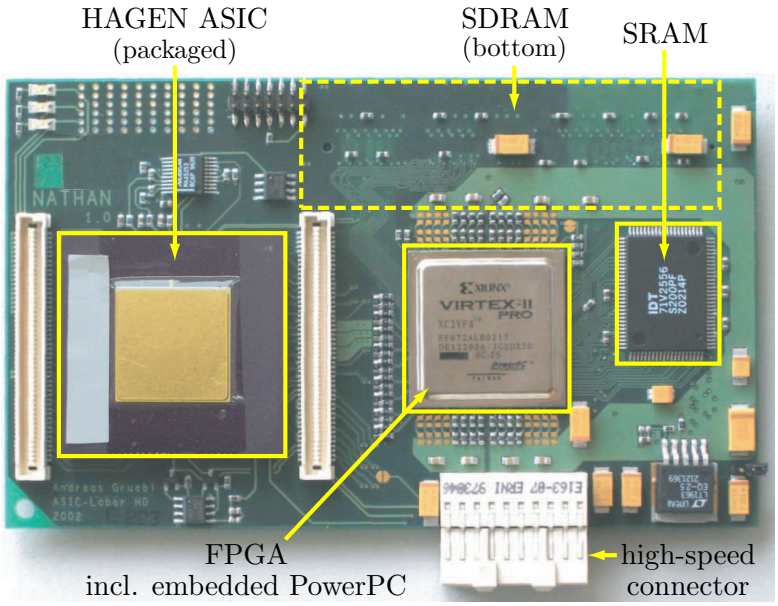


Figure 6.8: *Photograph of the top side of a NATHAN board (photograph by Andreas Grübl with kind permission). Multiple boards can be interconnected on a so-called Backplane PCB using the dedicated high-speed connectors for the multi-gigabit transceivers of the FPGA.*

ASIC prototypes. Apart from that, the NATHAN board provides connectors for eight high-speed links — so-called multi-gigabit transceivers (MGTs) — that are integrated in the used FPGA and are each capable of up to 3.125 Gbit/s.

faster HAGEN operation

One of the advantages of this new setup compared to the currently used environment is that the new FPGA model allows to operate the HAGEN chip at higher frequencies. Initial tests reveal that the interface of the network ASIC can successfully be operated at 200 MHz instead of the 100 MHz that can be realized on the Darkwing board [181] [185].

6.3.2 The Backplane System

interconnecting 16 NATHAN boards

In parallel to the NATHAN board itself, a dedicated backplane has been developed that hosts up to 16 NATHAN modules and interconnects them to form a toroidal topology using the provided high-speed links [78]. This backplane system is connected to a general purpose PC by utilizing a Darkwing card in combination with one of the LVDS interface extensions that usually interface to the carrier PCB of the HAGEN ASIC (see section 6.1.1). This way, a slow-control network between the host computer and the single network modules can be established.

enhanced resources

Various approaches are conceivable of how to fully exploit the enhanced network resources and high degree of parallelism that are offered by this new hardware environment. A possible application for large hierarchical networks (see section 2.3.2) that are distributed over multiple network chips is outlined in [52]. Apart from

implementing larger networks on several HAGEN ASICs, the use of multiple different network chips also provides the possibility to speed up the training process by evaluating several candidate solutions in parallel. If the training algorithm itself can benefit from parallelization, it could even be distributed over the multiple PowerPC processors of the single NATHAN modules itself.

operational readiness

The introduced advanced hardware environment is a conjoint work with J. Fieres, A. Grübl, S. Philipp, Dr. J. Schemmel, T. Schmitz, F. Schürmann and A. Sinsel. At the time of writing of this thesis, the single components of this system have passed the initial test stage and are demonstrated to be fully functional. The used software environment (see chapter 7) is successfully migrated to the embedded Linux that is executed on PowerPC core of the employed Xilinx FPGA. The whole framework will be fully functional in the near future. Although its operation and further development is beyond the scope of this thesis, the evolutionary training strategies presented in part III are partially designed in special consideration of the advantages and limitations of this new hardware setup.

6.3.3 Implications for Network Training

advantages

With regard to the training process, the NATHAN/Backplane environment will allow to exploit the speed of the HAGEN ASIC more efficiently than it is possible with the current Darkwing setup: The chip itself can be interfaced at higher speeds and the bottleneck of the PCI bus is efficiently avoided by executing the training software locally on the embedded PowerPC. A clever distribution of adequate training approaches over multiple NATHAN boards might even promote further acceleration due to parallelization.

limitations

It remains to be a noticeable limitation of this setup, that the featured PowerPC core operates at a maximum clock frequency of only 350 MHz[4] [232] [231] and does, e.g., not provide special floating-point units. Compared to the currently used 2.4 GHz Pentium IV, this breeds a substantial reduction in speed and is thus bound to affect the training process. Aiming to make best use of both, the accelerated network ASIC operation and the faster data transfer, the used training algorithm will be challenged to keep up pace with the implemented networks even more than in the present setup.

feasible hardware acceleration

Considering the application of evolutionary training algorithms, it will therefore become inevitable to employ the evolutionary coprocessor and thereby release the PowerPC from performing the time-consuming genetic operations in software. Ultimately, the calculation of the fitness is preferably migrated to the FPGA as well, and the software part is best reduced to only a minimum of controlling functionality.

simple training algorithms

As it has already been discussed in section 6.2 this implies to use simple genetic codings, variation operators, and fitness functions that allow for an efficient realization within a configurable logic. In summary, it can be concluded that within either of the described hardware setups, feasible evolutionary training ap-

[4] Three different speed grades are available with maximum clock frequencies of 300, 350 and 450 MHz, respectively [232] [231]. All those models are compatible with the NATHAN board. The eventual choice is determined by cost considerations.

proaches need to contend themselves with simple components and — in anticipation of the NATHAN/Backplane system — should rather aim to efficiently benefit from potential parallelization. A promising training approach will be introduced in chapter 9.

Chapter 7

The HANNEE Software

Real Programmers don't comment their code. If it was hard to write, it should be hard to understand.

Anonymous

The HAGEN neural network chip introduced in chapter 5 is optimized for highly iterative chip-in-the-loop training algorithms. This motivates the application of evolutionary strategies (chapters 3 and 4), and it has been discussed in section 6.2 that this kind of optimization procedure can greatly benefit from hardware acceleration. Nevertheless, as long as adequate training approaches remain to be a subject of investigation in their own right, the arising need for flexibility advises to implement at least part of the training in software.

Even if the training was completely realized on-chip or within a configurable logic, the user interaction with the hardware — either for mere system testing or the visualization and evaluation of training results and network responses — would eventually involve a dedicated software. Within the two neural network hardware setups presented in the preceeding chapter, the corresponding software components are executed on either the standard PC and/or the embedded PowerPC cores of the used Xilinx FPGAs.

In parallel to the HAGEN ASIC and the described hardware frameworks, a software environment has been developed that allows the user to test, analyze and train neural networks on the HAGEN chip for desired tasks. This software is called HANNEE (Heidelberg Analog Neural Network Evolution Environment). It has been designed in co-operation with Johannes Fieres and also received major contributions from Eilif Mueller, Johannes Schemmel, Tillmann Schmitz, and Felix Schürmann.

design goals

Regarding the prototype state of both, the neural network hardware as well as the appropriate training algorithms, the design of HANNEE had to fulfill at least two important requirements. First, the software has to be usable in connection with different, present and future hardware configurations (e.g., new generations of network ASICs or alternative PCB environments) with only minor additional programming effort. Second, it has to facilitate the integration of new software components contributed by different researchers with diverging research interests.

With respect to the evaluation of evolutionary algorithms as suitable training approaches, the second point also comprises the concrete desire for an easy combination of different evolutionary algorithm components (see sections 3.3.2 and 3.3.4).

platform independence

As a basic consequence of these considerations, the HANNEE software follows an object-oriented approach and is implemented in C++ [202]. At the time of writing of this thesis, it is primarily used on Unix-based systems, but special care has been taken to keep the source code largely platform-independent. Earlier versions have been used under the *Windows* operating system, and it is apprehended to constitute no major effort to port current and future versions to *Windows* as well.

The succeeding sections will describe HANNEE in more detail and outline how the main design goals formulated above are realized. Being written by a workgroup of scientists to serve as a tool for their ongoing research, the HANNEE platform is subject to constant development and improvement. Any detailed discussion of its implementation would be outdated within weeks and would in any case miss the purpose of this thesis.

In this sense, the present chapter is not intended to serve as an outright HANNEE manual or an exhaustive source code documentation (the latter is included as a *doxygen* [213] documentation within the source files). Rather, the main design concepts that establish HANNEE as a flexible and efficient software environment for the presented neural network hardware and as a valuable tool for the research on evolutionary training strategies will be investigated on a general and structural level.

Since the following sections are concerned with object-oriented software design, it is assumed that the reader is familiar with the philosophy of object-oriented programming and the concepts of classes, objects and inheritance. An introduction to this field can be found in literature [155] [202].

7.1 Overview

Figure 7.1 schematically illustrates the structure of the HANNEE software and its interaction with the neural network hardware. As it has already been discussed in section 6.1, any communication between the software and the HAGEN chip is mediated by a configurable logic. Within the software, the access to the FPGA is completely encapsulated by a set of dedicated hardware access classes. Besides providing the functionality to configure the FPGA and to control the operation of the neural network ASICs, these classes also manage the communication with the evolutionary coprocessor (see section 6.2).

7.1.1 Standardized Hardware Access

efficiency requirements

Realizing parts of the training algorithm in software gives rise to frequent data transfer between the neural network chips and the software environment (see sections 2.4.5 and 6.1.3). Therefore, an efficient implementation of the hardware access is vital if the speed benefits of using a fast neural network hardware are

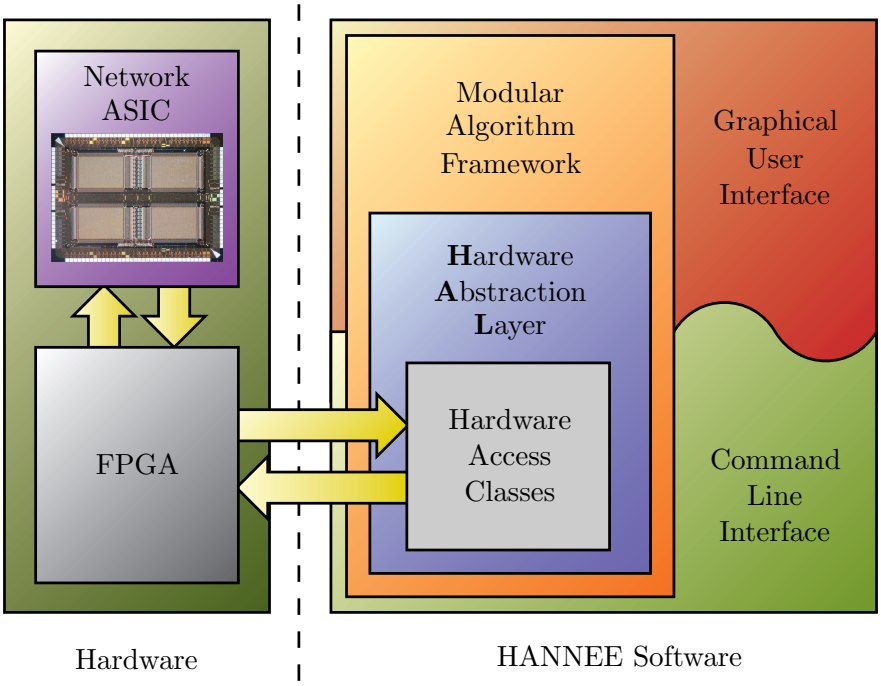


Figure 7.1: *Overview of the HANNEE software and its interaction with the hardware. The low-level hardware access is mediated by a set of dedicated hardware access classes that are encapsulated within the Hardware Abstraction Layer (HAL). The modular structure allows to easily integrate new algorithm modules. These components communicate with the network ASICs solely via the HAL interface and thus remain largely independent of the used hardware setup. HANNEE provides both, a graphical user interface and a command line interpreter.*

not to be compromised. This in turn requires the hardware access classes to be designed with regard to the peculiarities of a specific hardware configuration and initially opposes the desire to use the HANNEE software in different setups.

abstracted hardware access

On a more general level, however, the forms of interaction between the neural network chips on the one side and the user or the training algorithm on the other are anticipated to be independent of the specific setup: For the largest part, networks have to be implemented and executed on the ASICs and their results are to be read back for evaluation by the user and/or the algorithm. Neither the algorithm nor the user are concerned about — or should be forced to explicitly account for — the details of the software-hardware interaction. Similar considerations apply to the communication with the evolutionary coprocessor.

Hardware Abstraction Layer (HAL)

Therefore, the low level hardware access is separated from the remaining parts of the software by the so-called Hardware Abstraction Layer (HAL). This way, the access to the hardware is reduced to a concise and standardized set of methods and data structures that nevertheless summarizes the full functionality needed for common user interaction and the implementation of training algorithms.

Beneath the Hardware Abstraction Layer, the actual communication with the

network ASICs or the evolutionary coprocessor is delegated to an exchangeable set of hardware access objects that is specifically designed for the currently used hardware setup. Apart from selecting the present configuration, the user is not involved in the details of the hardware access. Furthermore, by exclusively addressing the network chip or the coprocessor via the HAL interface, all remaining parts of the HANNEE software retain the flexibility to be used with any present and future hardware configurations. The Hardware Abstraction Layer will be described in more detail in section 7.3.

7.1.2 Modular Structure

easy extensibility

Besides providing a standardized hardware interface, the HANNEE software exhibits additional features that facilitate the implementation of new training strategies or other experimental setups. The modular structure of the HANNEE platform allows to easily integrate new components into the project and to efficiently combine them with existing modules.

built-in functionality

Basic functionality — like the ability to store parameters and results on disk, a user friendly interface, and the possibility of automated batch scripting, etc. — is provided in the form of carefully designed base classes. Building on this foundation, the effort of developing new modules that can readily be used within the HANNEE framework and benefit from its various features is reduced to a minimum.

standardized algorithm interface

In order to promote the fast realization and testing of novel chip-in-the-loop training approaches, special care has been taken to warrant the easy interchangeability of different algorithmic components. This is achieved by introducing a dedicated standardized interface class which is used as the basis for all algorithm implementations in HANNEE. The main concepts of the modular approach are outlined in section 7.2.

evolutionary algorithm framework

Evolutionary strategies are deemed to be a particularly feasible approach for the training of mixed-signal neural network ASICs like the HAGEN chip and can gain considerable acceleration from the evolutionary coprocessor (section 6.2). Therefore, HANNEE has been equipped with an additional set of base classes that are designed to ease the realization of different evolutionary algorithm models for neural network training. The entirety of these classes forms the HANNEE Evolutionary Algorithm Framework (HEAF) that will exhaustively be discussed in section 7.4.

7.1.3 Automatically Generated User Interfaces

GUI and command-line interpreter

HANNEE can be controlled via two user interfaces, a graphical one and a command line interpreter. The latter proves to be convenient when automated procedures are processed in batch mode or when only a terminal connection can be opened to the executing machine. The graphical interface is implemented using the Qt library [210] which is available for multiple platforms.

user interface generation

While an intuitive user interface is important for a software to be of reasonable use in practice, research on new algorithmic concepts which requires the

frequent implementation and modification of new software components is not to be decelerated by the cumbersome details of creating corresponding graphical representations. It is one of the outstanding features of the HANNEE software that the user interface for new modules is to a large extent generated automatically. This refers to the graphical as well as the command line interface.

The creation of a solid software infrastructure that can serve as a convenient basis for development and as an easily expandable tool for scientific research is a demanding and time-consuming exercise by itself. This particularly includes, but is not limited to, the automatization of user interface generation and the implementation of scripting functionality. Nevertheless, although a lot of work has been dedicated to these subjects, they are not of direct relevance for the scientific work presented in this thesis. The elaborate details of how these convenience features are implemented will therefore not be described here.

7.1.4 Platform Independence

compatibility considerations

Within the currently used hardware setup (see section 6.1), HANNEE is executed on the IBM compatible general purpose computer that hosts the Darkwing board. At present, the primary development platform is Linux with a 2.4.x kernel and a GCC 3.3 compiler [67]. But apart from the usage of the QT library that can be compiled for both, Unix and *Windows* operating systems, several additional precautions have been taken in order to warrant an easy migration of the HANNEE software also to other compilers and platforms:

- All used libraries are openly available and can be compiled for any platform that is supported by the GCC.

- It has been abstained from using any compiler specific extensions to the C++ standard. This readily allows to employ a variety of compilers that can automatically optimize the code for the targeted CPU. For example, regarding the used Intel Pentium IV processor, HANNEE can alternatively be compiled by the dedicated Intel Compiler [104] (see also [185]).

- The low-level hardware access is mediated by the WinDriver 6.x product by Jungo, Inc. [114] which is available for all major platforms. Under Unix and *Windows*, it encapsulates the hardware device by a kernel module or a device driver, respectively.

HANNEE on NATHAN

Most notably, HANNEE can readily be compiled for the embedded Linux that is executed on the PowerPC 405 microprocessor of the NATHAN board. Currently, only the command-line interface is available when executing the program on NATHAN, but if desired, the graphical user interface can be included in future versions. The MacOs X operating system is supported as well and apart from some minor bugfixes that might potentially be required when the code is actually tested on other platforms, it is expected that HANNEE will immediately be operable also under *Windows NT/XP*.

7.2 HObjects and the HAlgorithm Concept

powerful base classes

HANNEE, as an extendible software framework, provides a set of powerful base classes with general functionality and a common interface. By inheriting these classes, the developer of new modules can readily use this basic functionality, and the such created new components are seamlessly integrated and interoperable within the overall HANNEE software.

Two examples of these base classes will be introduced here in more detail. First, the `HObject` class, since it represents the least specialized case and is in fact the common basis for all other classes that are to be inherited by new modules. Second, the `HAlgorithm` class, since it serves as the appropriate base class for all components that implement some kind of executable procedure like, e.g., neural network training algorithms.

7.2.1 The HObject Framework

general requirements

Algorithms in general and neural network training strategies in particular usually include numerous free parameters (see chapters 2, 3, and 4). One of the main challenges of successful training is posed by the appropriate tuning of these variables. Hence in practice, a neural network training software is required to allow for an easy adjusting of parameters via a suitable user interface and to provide means for storing previously found settings on disk. This can be extended to other parts of the HANNEE software as well. For example, the smooth operation of the hardware requires the specification of several parameters like clock frequencies, reference currents, etc. (see section 5.4.3).

Parameter Management via HElements and HValues

`HObject` functionality

Being derived from `HObject`, a new class automatically inherits the functionality to manage internal variables such that their contents can be stored on disk in XML (Extensible Markup Language [167]) format and are also automatically recognized as parameters that require an adequate representation in the user interface. Figure 7.2 shows a simplified UML (Unified Modeling Language [155]) diagram of the `HObject` class. Like most of the UML schematics shown in this chapter, figure 7.2 omits parts of the class specification for clarity. Only those attributes and methods are shown that are relevant for this discussion. For a detailed account of the presented classes, the interested reader is referred to the source code documentation.

`HElement`s as subelements

Any internal variables of an `HObject` that need to appear in the user interface and have to be included in the XML description are each encapsulated within a so-called `HElement`. A given `HObject` can contain multiple `HElement`s as subelements that are also referred to as its children. New children are added to and removed from the object via different overloaded forms of `addElement(·)` and `removeElement(·)`.

`HElement` functionality

Some basic features of `HElement` are shown in figure 7.3. Most notably, objects of this class exhibit the ability to convert their contents to valid XML strings. The settings of a specific `HElement` can be identified in an XML document via the

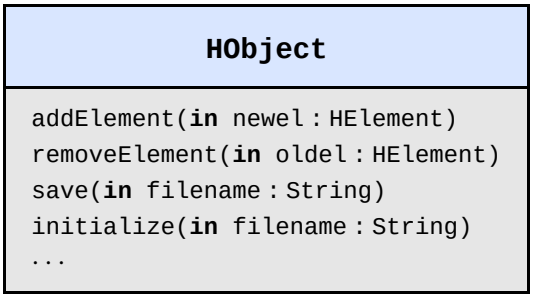


Figure 7.2: *Simplified UML representation of the* `HObject` *class. The diagram does not show the complete class interface. Several overloaded versions of some of the shown functions as well as attributes and methods that are not relevant for the discussion have been omitted.*

object's individual name. In addition to the name, each `HElement` is assigned one of a predefined set of XML tags and may contain a brief description of its purpose. Name, tag and description are typically specified upon construction of the object but can also be changed afterwards.

inheritance hierarchy, `HValue`*, and* `HGroup`

Various classes are derived from `HElement` and figure 7.4 illustrates part of the inheritance hierarchy. While `HValue`s generally encapsulate parameters of some kind (Booleans, integers, floating point numbers, arrays, etc.), `HGroup` objects contain and manage multiple `HElement`s as subelements. In fact, `HObject` inherits the functionality to host `HElement` objects from its `HGroup` base class. The ability to save its contents in XML format is owed to the circumstance that each `HObject` is ultimately an `HElement`.

complex element hierarchies

XML functionality

As a general result of this inheritance hierarchy, the children of an `HObject` are not restricted to be simple parameters (represented by suitable `HValue`s and possibly grouped within several `HGroup` subelements), but can themselves be arbitrarily complex `HObject`s with their own specific functionality. When called to convert its settings to an XML string, an `HObject` recursively includes the set-

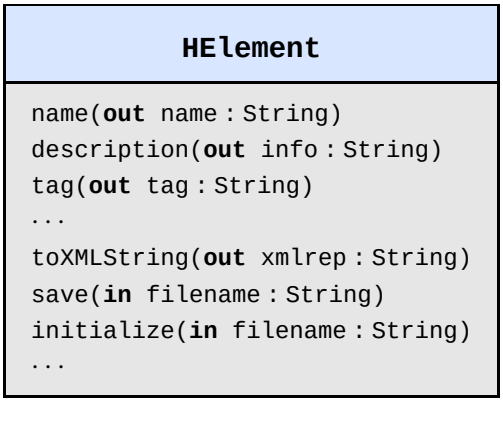


Figure 7.3: *Abbreviated UML diagram of the* `HElement` *class. Each* `HElement` *object has a name, a short description and an XML tag. The encapsulated data can be converted to resp. be reconstructed from a valid XML representation.*

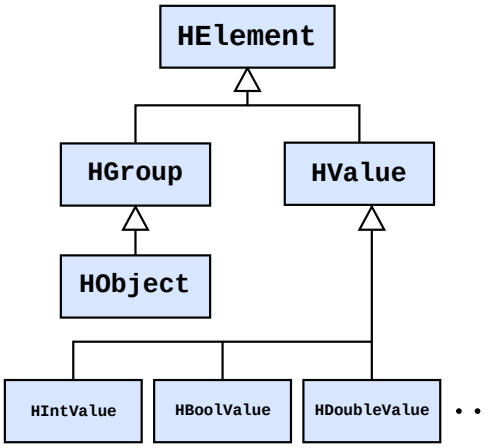


Figure 7.4: *Part of the* `HElement` *class hierarchy. As every* `HObject` *is also an* `HGroup`*, it can host other* `HElement` *objects such as the various* `HValue`*s. All of the shown classes inherit their XML functionality from* `HElement`*.*

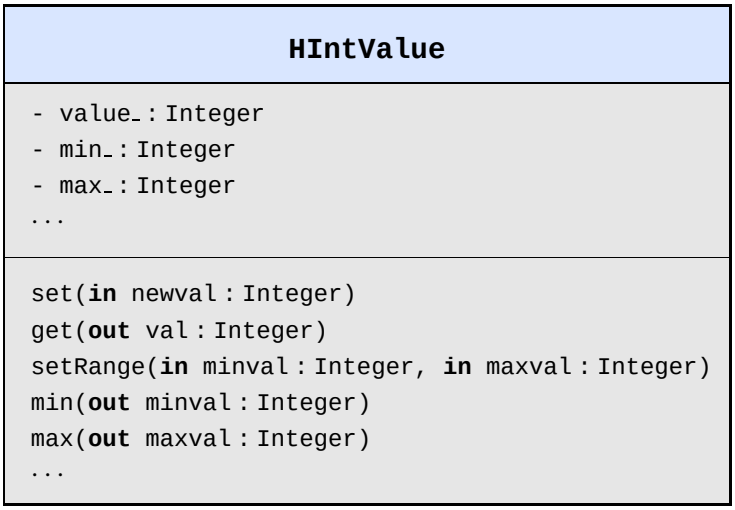


Figure 7.5: *Parts of the* `HIntValue` *class specifications. The internal integer variable can be accessed via the* `set(·)` *and* `get()` *methods. The valid range of the represented parameter — given by the integer values* `min_` *and* `max_` *— can be specified as well. Other* `HValue` *subclasses exhibit similar class interfaces.*

tings of all its subelements. Likewise, when being initialized from an appropriate XML document, it accounts for the initialization of all its children as well.

HIntValue

In order to provide an example for a specific `HValue` subclass, figure 7.5 shows the abbreviated UML diagram of the `HIntValue` class that is used to represent integer numbers. The actual value of the encapsulated variable can be accessed via the `set(·)` and `get()` methods and besides the value itself, `HIntValue` also allows to specify the valid range for the represented parameter. Similar `HValue` subclasses are provided for floating point numbers, Boolean variables, strings, arrays, etc. It is an important feature of the various `HValue` derivatives that they are thread[1] safe, i.e., the integrity of the internal data is warranted even when it is being accessed simultaneously from different threads of the application[2].

Easy Integration of New Functionality

HANNEE object nomenclature

According to the agreed notation of the HANNEE project, all classes that inherit from `HObject` begin with a capital `H`. In the following, it is understood that all such classes exhibit various attributes that are attached as corresponding `HElement` objects. In the shown UML diagrams, the common subelements of a class are usually omitted if not otherwise stated.

[1] In programming, a thread is a subprocess of a program that executes independently of its other parts but operates on the same address space. Threads are managed by the operating system and if the latter supports multithreading, the different threads of a program are executed concurrently [192].

[2] The thread safety is implemented via mutual exclusion. All multi-thread functionality within HANNEE is realized using the ACE library [1] that is openly available for all major platforms.

```
HMyClass::HMyClass()
  : HObject("MyObject", "This is my object.")
{
  addElement(new HIntValue("Integer Parameter", 2));
  addElement(new HDoubleValue("Float Parameter", 1.5));
  addElement(new HBoolValue("Boolean Parameter", false));

  addElement(new HGroup("Empty Group 1"));
  addElement(new HGroup("Empty Group 2"));
}

HObjectMapEntryImpl<HMyClass> mycl("HMyClass", "HObject");
```

Figure 7.6: *C++ constructor and registration entry of an exemplary new* `HObject` *subclass. Objects of this class are given three parameters, an integer, a float (with double precision) and a Boolean. The default value of each parameter is passed as the second argument to its respective constructor. In this simple example, the two subgroups remain empty. The last line registers the new class with the HANNEE application.*

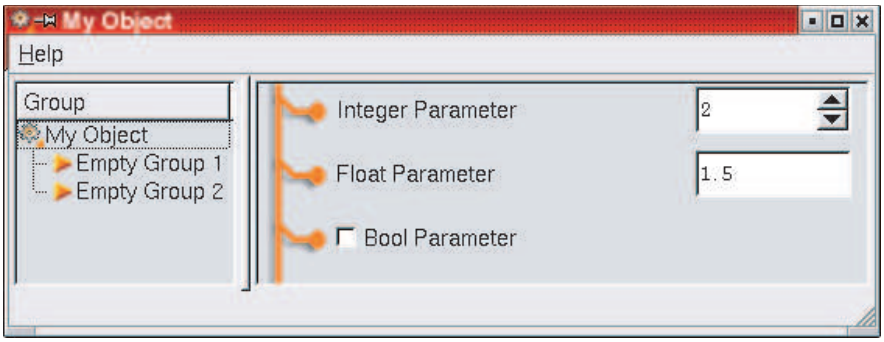


Figure 7.7: *Automatically generated user interface for an object of type* `HMyClass`*. The shown image is a screenshot of the HANNEE application being executed on a Linux system.*

introducing new `HObject`*s*

When additional source files with new `HObject` subclasses are compiled and linked to the HANNEE project, the relinked application will automatically recognize these new classes and allow the instantiation of corresponding objects by the user. It is an important feature of the `HObject` framework that the newly introduced functionality will be readily available to all parts of the program without the need to modify existing code. The new source files merely have to contain a single additional line of code that registers the new class with the HANNEE application. The C++ constructor of a simple exemplary `HObject` subclass together with the appropriate registering entry is presented in figure 7.6. The class declaration and the implementation of the destructor of `HMyClass` are straight forward and have therefore been left out. The complete code example can be found in figures A.3 and A.4 in the appendix.

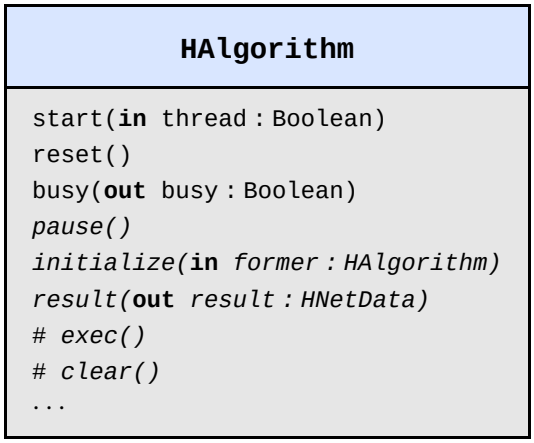


Figure 7.8: *Abridged interface of the* `HAlgorithm` *class. Abstract methods that have to be implemented by subclasses are set in a slanted font. The last two functions are protected, i.e., they are only visible inside the class and in subclasses but are not accessible from outside.*

automatic GUI generation

The automatically generated user interface of an `HObject` accounts for the hierarchical structure of its children and contains controls for its various `HValue` subelements. Figure 7.7 shows the resulting interface for an object of the new `HMyClass` introduced in figure 7.6. The labeling is generated on the basis of the various `HElement`s' names and it shall be repeated that apart from the code shown in 7.6, no further development is necessary to create this user interface.

7.2.2 Implementing New Algorithms as HAlgorithm subclasses

New neural network training algorithms are implemented on the basis of the `HAlgorithm` class which itself is a subclass of `HObject`[3]. `HAlgorithm` extends the functionality of its base class by adding several methods that define the common interface for any algorithm module to be used in HANNEE. The most important extensions are summarized in figure 7.8. Again, the class interface is abbreviated for simplicity.

Automated Execution Management

basic `HAlgorithm` *functionality*

Most prominently, every `HAlgorithm` can be started, paused and reset. The methods `start(·)` and `reset()` are predefined and are not to be reimplemented by inheriting classes since the provided implementations manage several internal procedures that are important for the smooth integration of an `HAlgorithm` into the HANNEE framework. In particular, the Boolean argument to the `start(·)` method allows to specify whether the algorithm is to be executed in its own separate thread or not. In any case, the provided implementations ensure a correct synchronization of the algorithm execution with other modules — such as potential monitoring objects — and the user interface in general. The `busy()` method, for example, automatically returns `true` for as long as the `HAlgorithm` is executing, i.e., has been started but not terminated.

[3] In reality, the situation is somewhat more complicated as `HAlgorithm` inherits from `HObject` via several intermediate classes, but that does not affect the present discussion.

implementing new subclasses

While the predefined versions of `start(·)` and `reset()` ensure the frictionless operation of any `HAlgorithm` within the HANNEE software, the actual functionality that is specific for a new `HAlgorithm` subclass is to be provided by implementing the two protected, virtual abstract methods `exec()` and `clear()`. Internally, these functions are called by `start(·)` and `reset()`, respectively.

Similar considerations apply to the `pause()` function that can be filled with a desired implementation by the developer and is expected to enable the user to manually terminate the algorithm execution. After calling `pause()`, a succeeding invocation of `exec()` is assumed to continue the execution at the point where it has previously been halted; a call to `clear()` is to restore the internal state of the algorithm to its initial conditions.

Figure 7.9 presents a simple but sensible implementation of an exemplary `HMyAlgo` class and figure 7.10 shows a screenshot of `HMyAlgo` in operation. The complete code for this example is provided in figures A.1 and A.2 in the appendix.

HAlgorithm Interaction

`HAlgorithm` results

The desired outcome of a neural network training algorithm is a network that performs a given task with a reasonable accuracy. Once an `HAlgorithm` has finished executing, the resulting network can be obtained via the `result()` method which returns the corresponding network specification in form of an `HNetData` object. The `HNetData` class summarizes all information that is necessary to implement a given neural network on the used network chip and will be described in more detail in section 7.3.1.

The `HAlgorithm` class is designed to suit as a practical base class not only for neural network training algorithms, but also for algorithmic modules that implement other kinds of processes. For an algorithm of the latter type, the result may possibly not be given by a neural network. The `result()` method is in this case expected to return 0.

combining different algorithms

Instead of starting from a default or random initial state, an algorithm might reasonably be initialized with the result of a former process. For this reason, the `HAlgorithm` interface includes the `initialize(·)` method that accepts another `HAlgorithm` as input parameter and can be implemented in subclasses as desired. In the case of neural network training strategies, the `initialize(·)` method of the new `HAlgorithm` will most probably call `result()` on the passed object and use the obtained network as starting point for its own execution. This readily allows to combine different training approaches like, e.g., traditional gradient-based algorithms and evolutionary strategies (see sections 4.1.3 and 4.2.2). At the same time, each new algorithm retains the standardized `HAlgorithm` interface and the modularity and reusability of the implemented concepts are warranted.

7.3 The Hardware Abstraction Layer

overview

The Hardware Abstraction Layer HAL consists mainly of three classes: `HNetMan`, `HNetData`, and `EvoCop`. The `HNetMan` offers access to the neural network ASIC and the programmable logic. Networks can be implemented and executed on the

```
HMyAlgo::HMyAlgo()
  : HAlgorithm("HMyAlgo", "This is my algorithm.") {
  maxiter_ = new HIntValue("Maximum Iteration", 10);
  addElement(maxiter_);
  paused_ = false;
  curriter_ = 0;
}

void HMyAlgo::exec() {
  paused_ = false;
  while (curriter_ < maxiter_->get() && !paused_) {
    log << "Current Iteration:  " << curriter_ << hlog;
    curriter_++;
  }
}

void HMyAlgo::clear() {   curriter_ = 0;   }

void HMyAlgo::pause() {   paused_ = true;   }
```

Figure 7.9: *C++ constructor and simple implementations of the* `exec()`, `clear()` *and* `pause()` *methods for an exemplary* `HMyAlgo` *class. Note how the internal integer variable of* `maxiter_` *is accessed in the termination condition of the* `while` *loop in the* `exec()` *method. The* `log` *stream is a HANNEE peculiarity. Effectively, it conveys the passed text to the output stream and also writes it into a special log file.*

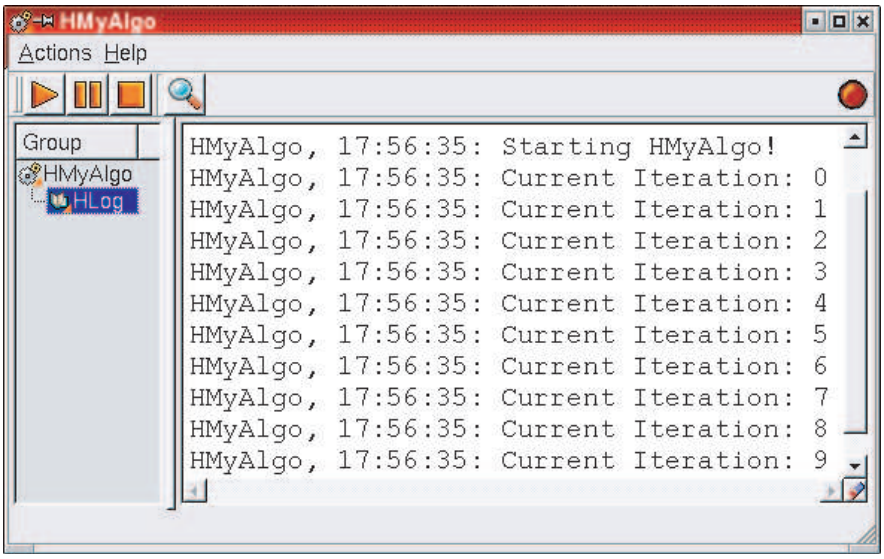


Figure 7.10: *Automatically generated user interface for an algorithm object of type* `HMyAlgo`*. The shown image is a screenshot of the HANNEE application being executed on a Linux system. Note the three buttons in the upper left corner of the window that allow to invoke the respective methods* `start(·)`, `pause()`, *and* `reset()` *manually. The output that is generated during execution is displayed in the* `HMyAlgo`*'s log window.*

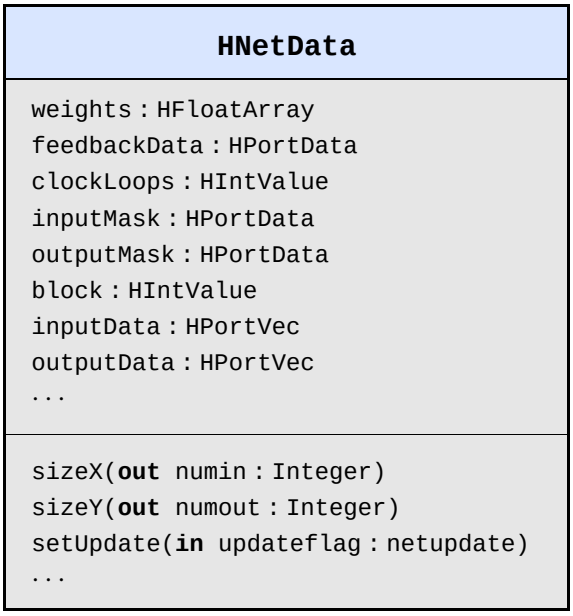


Figure 7.11: *Simplified UML description of the* `HNetData` *class. The desired network configuration is coded in the form of multiple dedicated* `HValue` *objects. (The* `HPortData` *class basically encapsulates a binary mask, and an* `HPortVec` *is essentially a vector of* `HPortData` *objects.) The applied input patterns and the resulting network output data are stored in* `HNetData` *as well. The interface is kept as general as possible to hide the peculiarities of the actually used network ASIC from the rest of the software.*

hardware by passing the desired chip configuration together with the desired input patterns to the `HNetMan` in form of one or more `HNetData` objects. The `EvoCop` provides an interface to the evolutionary coprocessor.

7.3.1 The HNetData class

When executing a network on the HAGEN ASIC, several parameters need to be specified in addition to the mere weight values in order to unambiguously define the desired topology (see section 5.2). Besides the required number of network cycles and the activated feedback connections, it is, e.g., to be appointed which of the output neurons in the various blocks are regarded as actual outputs of the network.

multiple network blocks

The `HNetData` class exhibits various publicly accessible data fields to store this configuration data, to hold the desired input patterns, and to allow for the storage of the obtained network response (see figure 7.11). In the current version of HAL, one `HNetData` object specifies one network block; the targeted block coordinate[4] is determined by the `HNetData`s `block` value. Multiple `HNetData` objects can be connected to a linked list: When an `HNetData` is passed to the `HNetMan` for implementation on the hardware, all appended objects are automatically processed as well, thereby allowing to configure a complete HAGEN chip by passing a list of four appropriately prepared `HNetData` objects.

Chip Peculiarities

subclasses for specific ASICs

While the `HNetData` class represents the general interface for chip configurations as it is visible from outside the Hardware Abstraction Layer, the peculiarities

[4] In the case of the HAGEN prototype, this is one of the quadrants *upper left*, *upper right*, *lower left*, or *lower right*.

of different past, present, and future ASICs are to be accounted for by deriving corresponding subclasses. For example, the configuration of one HAGEN block is given by an object of the `HHagenData` class. For the reasons discussed at the beginning of this chapter, the actual nature of the currently used network chip is to be hidden from the rest of the program, and the `HNetData` interface has to be kept as general as possible. Therefore, the dimensions of the weight matrix — being one of the specifications that is most likely to be changed for future chips — can be inferred via `sizeX()` and `sizeY()` that return the number of input nodes and output neurons, respectively.

analog parameters

In addition to the network configuration, each type of network ASIC is anticipated to require the specification of several analog parameters (reference currents, bias voltages, etc. — see also section 5.4.3) that are likely to differ between the individual chip models. These settings do therefore not appear in the `HNetData` interface, but are defined in the chip specific subclasses where they are coded via adequate `HValue`s. As such, these parameters are not initially visible from outside HAL. But due to the functionality of the `HObject` framework, they can, e.g., be manually adjusted via the user interface.

Network Evaluation

executing networks on the hardware

Implementing and evaluating a neural network on the hardware proceeds as follows: First, the desired network configuration is translated into a linked list of `HNetData` objects — including the feedback connections, the weight values, and the desired input patterns. Second, the list is sent to the hardware for execution. Third, the resulting response of the network can be obtained from the `outputData` fields of the respective `HNetData` objects and can be evaluated as required. The second step is the only part that involves the hardware and is accomplished by using the `HNetMan` class (section 7.3.2).

repeated execution

During the execution of chip-in-the-loop training algorithms, multiple networks are iteratively evaluated on the chip that usually only differ in a few aspects, e.g., the weight values. It is thus undesirable to retransfer the whole set of unchanged settings to the hardware over and over again. In fact, weights, feedback information, and input patterns are stored in the local memory of the FPGA and potentially get updated whenever a new `HNetData` is sent (see section 6.1.3). A dedicated set of update flags in `HNetData` allows for a differentiated specification of the information that actually needs to be retransmitted. In the case of the HAGEN/Darkwing setup, this can particularly speed up the network evaluation if the used input patterns are the same for all tested networks and only the weight values have to be updated for each run (see section 6.1.3).

7.3.2 The HNetMan class

For all classes outside the hardware abstraction layer, the access to the used neural network ASIC is mediated by one or more objects of the `HNetMan` class. Figure 7.12 shows a simplified UML representation of its interface. Besides several functions for the configuration of the FPGA, the testing of the hardware setup,

and the calibration of the used neural network ASIC (see section 5.5), `HNetMan` provides several overloaded versions of the `run(·)` method. `HNetData` objects (see above) are sent to the hardware via this function, and consequently, every version of `run(·)` expects at least an `HNetData` object as input argument.

Forms of Network Evaluation

evolutionary coprocessor support

In its simplest form, `run(·)` does not require additional information beyond the desired network configuration. The remaining examples, however, show two important variations of how a network can be evaluated on the ASIC. The first of these alternative versions is to be used in conjunction with the evolutionary coprocessor (see section 6.2): The additional argument is interpreted as an address in the coprocessor's local memory that points to an individual in the current population. Instead of using the `weights` array of the passed `HNetData` object, the synaptic weight values are read from the specified address. At the time of writing of this thesis, the coprocessor is only applied to the evolution of the network weights. All other parameters of the network specification are obtained from the passed `HNetData` object (see also section 6.2.2).

coprocessor benefits

As the weights represent by far the largest part of the actual network configuration data (not considering potential input patterns), taking their values directly from the coprocessor's RAM is considerably more efficient than transferring them from the software. For the Darkwing/HAGEN setup described in section 6.1, using the evolutionary coprocessor in connection with this alternative version of the `run(·)` method speeds up the network evaluation by approximately a factor of 2 (see also section 6.1.3).

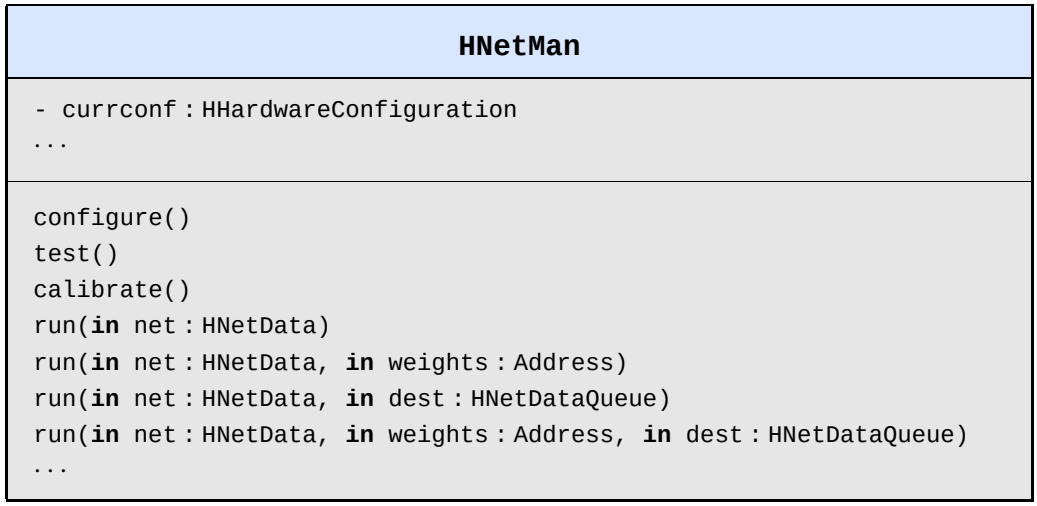


Figure 7.12: *Abridged UML schematic of the `HNetMan` class. Besides various methods for the FPGA configuration, hardware testing and chip calibration, `HNetMan` allows to execute networks on the neural network ASICs via several overloaded `run(·)` functions. Internally, the interaction with the hardware is managed by an exchangeable `HHardwareConfiguration` object. Each `HHardwareConfiguration` subclass represents a given combination of network ASIC and PCB environment.*

Both of the above functions do not return before the evaluation of the network on the hardware is finished, and the software thus remains inactive during the whole network execution. This is acceptable, as long as the process that initiated the hardware operation cannot reasonably proceed without knowledge of the network's response. On the other hand, various scenarios are conceivable where the time of the network evaluation can be used more efficiently. For example, in the course of an evolutionary algorithm, the fitness of a former network could be calculated while in the meantime, the next individual is already being executed on the hardware.

threaded hardware access

The last two of the shown `run(·)` methods allow to perform the network evaluations in the background. Instead of waiting for the hardware to be finished, these version return immediately after the respective `HNetData` object has been passed to the `HNetMan`. Internally, the `HNetMan` enqueues the received object at the end of a designated first-in, first-out queue (FIFO). The network specifications in this queue are sent to the hardware sequentially, and this is performed in a separate, dedicated thread.

`HNetDataQueue`

The last argument of the respective method specifies the destination FIFO where the resulting `HNetData` (including the obtained network output) is to be appended after its execution is completed. For such purposes, HANNEE provides the thread safe `HNetDataQueue` class. From this queue, the processed `HNetData` objects can be accessed by the calling processes in their own threads. Returning to the above example, the algorithm could first send all individuals of the current population to the `HNetMan` to be enqueued for evaluation. After that, it can immediately proceed with calculating the fitness of the sequentially arriving network results while the hardware is operating in parallel.

Managing Hardware Configurations via HHardwareConfiguration

Internally, the `HNetMan` delegates all hardware requests to its `HHardwareConfiguration` subelement. The interface of the `HHardwareConfiguration` class is essentially equal to that of the `HNetMan` discussed above. Specific hardware setups are to be represented by corresponding subclasses, like, e.g., `HHardwareConfigurationHagenDarkwing`[5].

exchangeable hardware functionality

Wrapping the peculiarities of specific hardware environments into exchangeable subelements of the `HNetMan` has various benefits. For example, it allows to implement new control software components for novel hardware configurations without the need to change or inherit `HNetMan` itself. The `HNetMan` class thus persists to serve as the exclusive, unchanged, and reliable hardware interface for the rest of the program. Furthermore, as `HHardwareConfiguration` is derived from `HObject`, new versions that work with different combinations of ASICs and PCB environments can easily be integrated: The `HNetMan` allows the user to select from all currently implemented variants.

integrating hardware-related code

At the same time, the actual functionality of a specific `HHardwareConfiguration` subclass will itself be provided by a given set of internal hardware access objects. These classes need not fit into the `HObject` framework or provide

[5] Abbreviated names are conceivable.

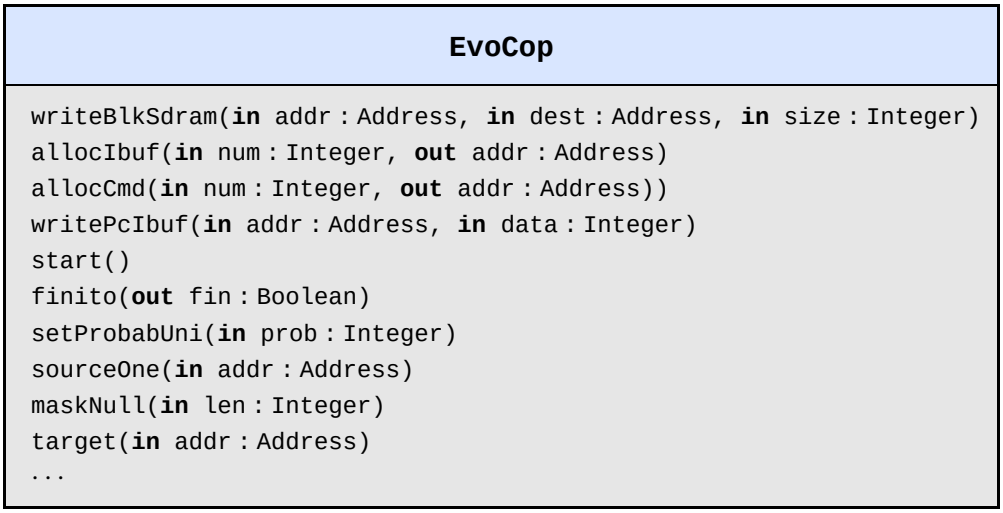


Figure 7.13: *Abbreviated UML interface of the* `EvoCop` *class. Apart from dedicated methods that implement the transfer of genetic data between the memory of the software and the local RAM of the coprocessor, the* `EvoCop` *class provides several functions to manage and fill the coprocessor's data and instruction buffer. A call to* `start()` *transmits the contents of this buffer to the local memory of the FPGA and initiates the execution of the defined instruction sequence.*

any functionality beyond the interaction with the hardware. This way, the developer of new hardware access software can concentrate on the optimization of the hardware related code. The resulting functionality can then easily be wrapped into an `HHardwareConfiguration` subclass that provides all the convenient features of the `HObject` framework, can readily be integrated into the HANNEE software, and is therefore immediately available for use.

7.3.3 The EvoCop class

As it has been discussed in section 6.2.1, the evolutionary coprocessor that is implemented in the configurable logic of the used hardware environment can be controlled via an extensive set of instructions. Within HANNEE, the access to the coprocessor is encapsulated by an object of the `EvoCop` class which provides the functionality to read and write genetic data to and from the local memory of the FPGA, to define a desired sequence of commands, and to initiate the successive execution of these instructions by the coprocessor itself. The abridged interface of the `EvoCop` class is shown in figure 7.13.

types of commands

According to what has been stated in section 6.2.5, the provided set of instructions can roughly be divided into four groups: instructions that define the configuration for each stage of the pipeline, data transfer commands, instructions that specify the global evolution parameters, and those that control a single recombination process, e.g., by specifying the source and target addresses of the processed genetic material in the local memory of the FPGA.

fixed configurations

In the current stage of the used neural network framework, neither the coprocessor nor the HANNEE software allow to modify the basic operation of the single stages of the coprocessor pipeline. In other words, the first type of instructions is not yet supported and the lookup table (see section 6.2.4) for each of the stages shown in figure 6.5 is predefined (see also section 6.2.5).

data transfer

The transfer of genetic data from the software's RAM space to the local memory of the coprocessor is implemented by the `writeBlkSdram(·)` method. This function requires the specification of the source address in the local memory of the CPU, the target address within the SDRAM of the coprocessor, and the number of bytes to be transferred. A corresponding `readBlkSdram(·)` method (not shown in figure 7.13) transmits a defined number of bytes from the memory of the coprocessor into the memory of the CPU .

Instruction Buffer Management

pooling instructions

While the above commands are issued to the coprocessor directly and executed immediately, the majority of instructions that control the processing of the hosted genetic material are temporarily pooled inside an instruction buffer that resides

coprocessor execution

in the RAM space of the software (see section 6.2.5). Once this buffer has been filled with the desired set of instructions, an invocation of the `start()` command transfers the whole sequence to the coprocessor for execution. The `start()` method returns as soon as the respective data has been transmitted, i.e., during the processing of the genetic data, the software can in principle address other tasks. The state of the coprocessor can be inferred at any time via the `finito()` function which returns true as soon as the coprocessor has finished executing.

buffer management

Apart from the actual instructions, dedicated parts of the buffer can also store frequently used data like, e.g., an available set of valid chromosome addresses and lengths (see section 6.2.5). The buffer space for the desired number of commands and data entries is allocated via the corresponding `allocCmd(·)` and `allocIbuf(·)` method, respectively (buffer space that is not required any longer can be released by according `freeCmd(·)` and `freeIbuf(·)` methods not shown in figure 7.13). While any desired data can then be written to a given address in the buffer using the `writePcIbuf(·)` function, all specified instructions are consecutively stored in the reserved command buffer in the order in which they have been issued. After a call to `start()`, the thus defined sequence of commands is be executed by the coprocessor in the same order.

Evolution Commands

global parameters

Three global evolution parameters can be specified externally: the respective probabilities for each single gene to be subject to uniform and Gaussian mutation as well as the width of the Gaussian distribution that is used to generate the random numbers for the latter. As an example, the probability for uniform mutation can be set by the `setProbabUni(·)` method. Corresponding functions are provided for the remaining two parameters. Typically, these methods are only executed

once at the beginning of the training run[6].

recombination instructions

For each recombination process, the two source addresses of the parent chromosomes in the local memory of the coprocessor, the target address of the offspring chromosome, as well as the desired crossover mask need to be specified (see sections 6.2.3 and 6.2.4). The required address information can be provided by the `sourceOne(·)`, `sourceTwo(·)` (not shown), and `target(·)` methods. The passed values do not directly spcify the adresses of the chromosomes in the RAM but always refer to entries in the data buffer.

crossover masks

Crossover masks can be defined in a runlength encoding using the `maskNull(·)` and `maskOne(·)` functions (the latter method is not included in figure 7.13 for simplicity). Initially, the mask has length 0. Using the above methods, it can successively be extended by adding corresponding sequences of ones and zeros with the respectively specified lengths.

During evolution, the software-implemented training algorithm will typically pool a whole sequence of instructions that defines the recombination of multiple chromosome pairs — e.g., those that form two entire genomes or even several mating pairs — before the actual processing of the genetic data is initiated by a call to `start()`. A simple code example for how the methods of the `EvoCop` class can be used to program the coprocessor in practice is given in figure A.5 in the appendix.

additional features

At the time of writing of this thesis, the evolutionary coprocessor is still under development and subject to ongoing improvement. A variety of novel features has been added since its original version that allow for a more efficient control of the recombination process (see section 6.2.5). While this additional functionality is already supported by the HANNEE software, it has not yet been thoroughly tested and is not used for the experiments presented in this thesis. A thorough discussion of these features is therefore deferred to later publications that will describe the coprocessor in more detail[7].

7.4 HEAF: An Object-Oriented Framework for Evolutionary Algorithms

It has been motivated in section 7.2.2 that implementing new network training algorithms in the form of individual `HAlgorithm` subclasses greatly simplifies the process of integrating and testing new algorithmic concepts and allows to easily combine different training strategies. Regarding the special case of evolutionary algorithms, the modularity concept can even be taken further.

evolutionary algorithms and object orientation

As outlined in sections 3.3.2 and 3.3.4, the different parts of an evolutionary algorithm — the fitness calculation, the selection scheme and the genetic representation in combination with the variation operators — are inherently indepen-

[6] Note, however, that the specification of the global parameters nevertheless requires to allocate space for a corresponding number of commands in the instruction buffer and is to be executed with a call to `start()` (see also figure A.5 in the appendix).

[7] This primarily refers to the aforementioned PhD thesis of Tillmann Schmitz that has not been finalized at the time of this writing.

dent and interact through well-defined interfaces. For this reason, evolutionary algorithms lend themselves particularly well to an object-oriented and modular implementation.

HEAF overview

The HANNEE Evolutionary Algorithm Framework, henceforth referred to as HEAF, is designed in special consideration of this modularity. It is founded by the following classes: `HFitnessFunction`, `Genome`, `HGenome`, `HPopulation`, and `HSelectionScheme`.

design goal: coprocessor integration

Aiming for an extensive flexibility in the combination of different realizations of the individual evolutionary algorithm components, it has been a major design goal to smoothly integrate the evolutionary coprocessor. Selection schemes, population models and fitness functions are to be combined with the coprocessor in an equally straight forward manner as they can be connected to pure software solutions of the genetic representations and variation operators.

7.4.1 The HFitnessFunction Class

efficiency vs. flexibility

The evaluation of the fitness constitutes a time critical aspect of every evolutionary algorithm since it has to be repeated for each new individual in every generation. A desirably fast fitness estimation process demands an efficient implementation of the involved calculations. Unnecessary overhead like it may potentially be introduced by an enhanced flexibility and configurability of the fitness function is best avoided. On the other hand, the investigation of new evolutionary approaches for the solution of a given task particularly relies on the option to vary the details of the fitness calculation as well.

implementation implications

Rather than implementing one function that allows to chose between various slightly different fitness estimation approaches — e.g., by incorporating multiple internal conditional statements and calling alternative subroutines — it is preferred to implement several slightly different, specialized, and self-contained versions. This approach is likely to lead to the duplication of large code fragments and may thus be deemed inferior to more modular and elegant programming paradigms. But in this case, it is justified by the sheer need for efficiency. Since the HAGEN ASIC is optimally suited for the fast processing of large numbers of input patterns, this particularly applies to those parts of the fitness calculation that are to be repeated for each single output pattern.

At the same time, a given set of several, slightly different fitness functions may still be regarded as modified versions of one and the same fitness estimation process. From the user's point of view, these methods should appear as the same function with merely different parameter settings. The way in which the fitness evaluation process is realized within HEAF tries to reconcile the desire for flexibility with the need for efficiency.

Fitness Functions

globally visible fitness functions

The various forms of fitness calculation are implemented as globally visible methods that are not member functions of any class. These functions expect two arguments: First, the `HNetData` object that contains the network response to

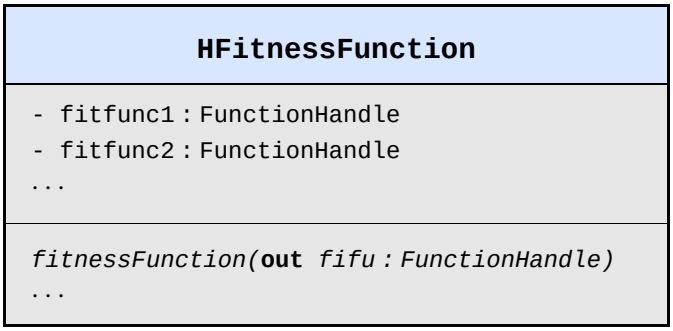


Figure 7.14: *Simplified UML diagram of the* `HFitnessFunction` *class. Since the actual fitness calculation is delegated to global, specialized, non-class functions,* `HFitnessFunction` *merely provides the handle to the fitness function implementation that suits the currently chosen parameters. The respective methods are standardized to accept two* `HNetData` *objects as inputs and return the calculation result as a* `Fitness` *object.*

be evaluated, and second, an additional `HNetData` object that includes the respective correct output for comparison. The result of the calculation is to be returned in the form of an object of a `Fitness` subclass. Most commonly, a given derivative of `Fitness` merely encapsulates a single numerical value (e.g., a floating-point number). The use of an abstracted `Fitness` class is aimed to promote an easy extension to multi-objective optimization [37] in the future.

`Fitness` class

While using a whole `HNetData` object to merely store the correct network output might seem lavish, it allows to incorporate additional specifics of the networks, e.g., the resemblance to a given solution, into the fitness estimation. Furthermore, it has to be taken into account that given the usual size of typical training data sets, the superfluous information contained in an `HNetData` object is most often negligible compared to the amount of processed input patterns and the corresponding responses.

Management By HFitnessFunction objects

providing fitness functions

New fitness function components are integrated into the HANNEE framework in form of an `HFitnessFunction` subclass (see figure 7.14). Since the actual fitness calculations are implemented by different global non-class functions, an object of a given `HFitnessFunction` does not provide methods to directly estimate the fitness of a network. Instead, it can be called to return a handle (also called pointer [115]) to an appropriate C-function and thereby introduce the chosen fitness measure to the remaining components of the evolutionary algorithm.

managing fitness functions

While the fitness functions themselves are specialized and preferably depend on as few variable parameters as possible, a dedicated `HFitnessFunction` subclass is able to manage a whole set of different but related fitness calculation methods. Depending on the current values of several suitable, user accessible parameters, a call to `fitnessFunction()` will return the handle to the corresponding version

among the internally known set of functions. By storing the respective pointer for as long as it is needed, the remaining parts of the software can directly call the associated function whenever required.

The `HFitnessFunction` class thus allows to manage potentially large and unhandy sets of functions and easily integrate them into the HANNEE platform via the `HObject` framework. This way, the drawbacks of abandoning a modular and elegant approach for the actual implementation of the fitness calculations are at least partly compensated.

7.4.2 The Genome and HGenome Classes

genetic coding and variation operators

The genetic representation of a candidate solution and the associated set of adequate variation operators can be seen to form a closed unit (section 3.3.2). While they must carefully be chosen to ideally complement each other, their implementation is initially unaffected by those of the selection scheme or the fitness function and vice versa. Therefore, it suggests itself to wrap both, the coding and the variation operators into one class.

The Abstract Genome Interface

general interface

In HANNEE, the genotypes of single individuals are represented by objects of appropriate classes that are to be derived from the common `Genome` base class (see figure 7.15). In fulfillment of the essential requirements of evolutionary algorithms, each `Genome` can be duplicated, mutated, recombined with another `Genome`, translated into an individual (i.e., a neural network), and assigned a fitness value.

The details of how the parameters of a candidate solution are stored and organized internally are completely hidden behind the general `Genome` class interface and the same applies to the variation operators. In other words, the remaining components of the evolutionary algorithm will merely require the various subclasses to implement the methods generically provided by every `Genome`.

coprocessor integration

In this context, it is important to emphasize that different versions of the genotypic representation can either realize the coding and the operators purely in software or utilize the evolutionary coprocessor in the form of a dedicated `EvoCop` object. As long as the corresponding classes retain the common `Genome` interface, they can equally well be combined with any implementation of the selection scheme or fitness function.

The HGenome Concept

In order to allow for an easy exchange of different `Genome` derivatives, its interface is deliberately kept as general as possible. Regarding the actual algorithm execution, the provided methods should suffice to warrant the usability of the `Genome` class in manifold different evolutionary algorithm setups.

`Genome` configuration

On the other hand, if it is to be applied to multiple tasks that are likely to require networks of varying size and architecture, any implementation of a genotypic representation is best made configurable in the number and partitioning of the contained genes (see section 3.4.2). In that case, the way in which a genome

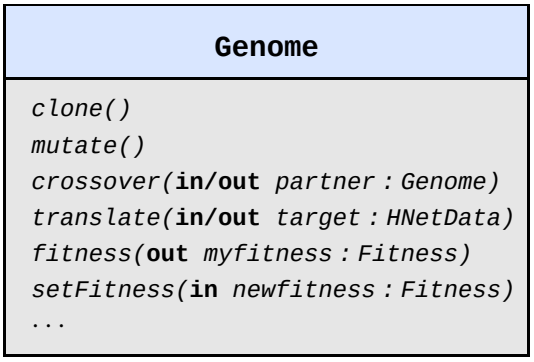


Figure 7.15: *Abbreviated UML schematic of the* `Genome` *interface. From the point of view of the other evolutionary algorithm components, the functionality of a genome can be summarized by a concise set of methods. A genome can be duplicated (cloned), mutated, recombined with another genome, and translated into a phenotype. The performance of this phenotype determines the genome's fitness.*

is to be translated into a neural network needs to be variable as well. Finally, the used variation operators will in general allow for the tuning of several settings, like mutation probabilities and crossover rates, etc. For some genetic representations, even several alternative mutation and crossover operators might be available (see sections 6.2 and 8.3.1).

desired `HObject` *functionality*

In practice, the realizations of different genetic codings should preferably allow to manipulate their adjustable parameters via an adequate user interface and provide the possibility to save these settings to disk. In so far, it would stand to reason to employ the features of the `HObject` framework and derive new `Genome` subclasses from `HObject`. But in addition to an enhanced functionality, inheriting `HObject` also breeds a measurable overhead in terms of an object's RAM consumption as well as in the complexity of its construction and destruction. As long as an object either exhibits a sufficiently large complexity by itself or has a reasonably long life expectancy, the introduced overhead is usually outbalanced by the practical features of `HObject`.

`HObject` drawbacks

During the execution of an evolutionary algorithm, multiple candidate solutions are in fact repeatedly created, tested and discarded. At the same time, the chosen evolution parameters and the desired structure of the genome typically apply to the population as a whole and are not specific to individual genomes. Therefore, representing each single genotype by a corresponding `HObject` is deemed impracticable.

Rather, each `Genome` subclass is to be accompanied by a corresponding derivative of the `HGenome` class (see figure 7.16). As indicated by the capital `H`, `HGenome` inherits `HObject` and the management and accessibility of the internal variables is thus automatically accounted for (see section 7.2.1).

`HGenome` factory class

Apart from that, a given `HGenome` derivative serves as a factory class for the associated `Genome` subclass in that it can create objects of the latter in consideration of its various parameters as they have been specified by the user. This functionality is provided by `newGenome()`. When the relevant settings are changed during an evolution run, any existing `Genome` can be modified to account for these changes by calling `HGenome::updateGenome(·)` with the respective object as parameter.

`Genome` management via `HGenome`

The most frequently used internal variables of a genotype might still be stored within every single `Genome` object of the population to speed up their access by

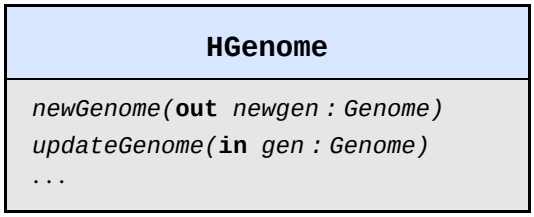


Figure 7.16: *Abridged UML representation of the* `HGenome` *class.* `HGenome` *allows to create new* `Genome` *objects according to the current values of its internal parameters and to update already existing* `Genome`*s to suit the specified settings.*

the objects' own methods. At the same time, their values can be managed centrally by only a single `HGenome` instance. This allows to implement elaborate ways of specifying the genome structure and/or to chose between different evolution settings and variation operators without introducing intolerable overhead to the single `Genome` instances. While this approach promotes an efficient handling of the genetic material during the execution of the evolutionary algorithm, it obviously leads to additional organizational effort during the setup phase: Whenever the genotypic parameters change, it is to be warranted that every `Genome` in the population gets updated by the responsible `HGenome` object. This task is adopted by the `HPopulation` class.

7.4.3 The HPopulation Class

Although initially, a population is essentially nothing more than a group of individuals, representing it by its own class is motivated by various considerations.

Motivations for an Extra Population Class

the population as the unit of evolution

The population is regarded as the unit of evolution in that the repeated variation and selection of its single individuals gradually improves the average quality of the contained candidate solutions (see section 3.1.1). As such, the selection schemes of an evolutionary algorithm can be seen to transform the population as a whole (see also figure 3.3). Transferred to an object-oriented implementation, this insight motivates to complement the concept of a dedicated selection scheme component with a corresponding population class.

managing genomes

In combination with the dual `Genome`/`HGenome` setup introduced in the preceeding section, a population object is also practical in so far as it can internally account for the synchronization of the single `Genome` objects with the settings of the responsible `HGenome` instance. Also, when using the evolutionary coprocessor, several issues like the RAM management for the gene data, the efficient handling of instruction sequences, and the general configuration of the coprocessor can be hidden from the remaining evolutionary algorithm components without the need to intricately distribute their solution across the multiple `Genome` objects.

handling the coprocessor

HPopulation Class Interface

Like in the case of the `Genome` class, the interface of `HPopulation` as it is visible to the remaining parts of the algorithm is kept preferably simple and general (see figure 7.17). During execution, an evolutionary algorithm is most likely

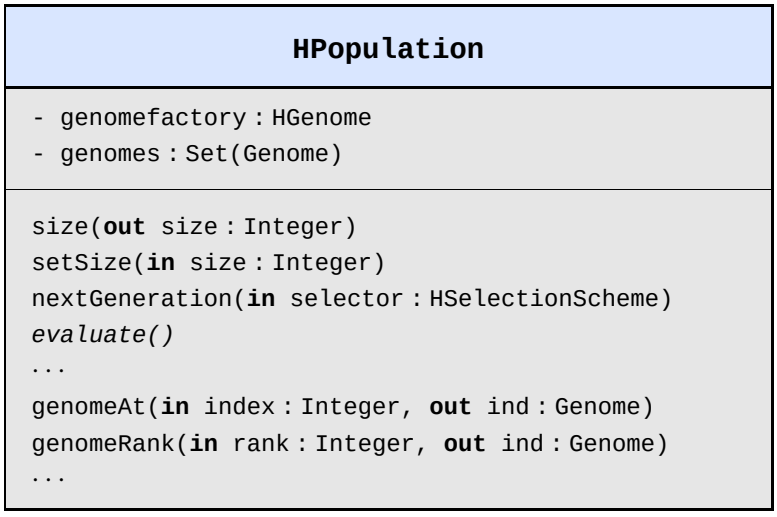


Figure 7.17: *Simplified UML representation of the* `HPopulation` *class. Every* `HPopulation` *hosts a group of* `Genome` *objects and automatically accounts for their synchronization with the settings of the dedicated* `HGenome` *instance. During an evolutionary algorithm, the population is repeatedly transformed by the desired selection scheme and the new individuals are evaluated for their fitness. Single individuals can be accessed from outside.*

to alternately and repeatedly call `evaluate()` and `nextGeneration(·)` on its population object. The first method sequentially translates all hosted `Genome`s to individual networks and uses the associated fitness function to calculate their fitness values. The second method applies the selection procedure that is implemented by the passed `HSelectionScheme` (see section 7.4.4).

genome access

The selection scheme and other components of the software can access the single `Genome` objects of the population via the `genomeAt(·)` and `genomeRank(·)` methods. The former expects the (arbitrary) index of the desired individual in the internal set of genomes as parameter. In the latter case, the demanded individual is specified via its current rank as it is determined by its fitness value. Using the functionality provided by the general `Genome` interface, members of the population can then be duplicated, replaced, recombined and mutated as required in order to realize multiple different selection procedures.

implementing subclasses

The details of `nextGeneration(·)` and `evaluate()` are to be supplied by specialized subclasses and depend, e.g., on the internal organization of the population and whether it is dedicated for use with the evolutionary coprocessor or not. The `HGenome` subelement of an `HPopulation` is exchangeable and can be determined by the user. The interfaces of both classes are designed to allow for a combination of various genotypic representations with different population models. The only restriction is that genetic codings that utilize the coprocessor cannot be combined with population implementations that expect a pure software realization. It is to be repeated that apart from that, a potential specialization to the evolutionary coprocessor is not visible from outside, particularly not to the applied fitness functions or selection schemes.

coprocessor integration

7.4.4 The HSelectionScheme Class

The `HSelectionScheme` class (figure 7.18) serves as the basis for potential subclasses that each encapsulate a different model of how to perform a parent or survivor selection process on a given `HPopulation`. The actual selection is realized by the `newGeneration(·)` method that accepts the population to be transformed as its argument.

parent and survivor selection

The name `newGeneration` has historical origins and might be misleading. In fact, for some evolutionary algorithms, both, parent and survivor selection might have to be applied to yield the next generation (see sections 3.2 and 3.4.1). In that case, the two selection procedures are likely to be performed by different `HSelectionScheme` objects and `HPopulation::evaluate()` is to be called in between to evaluate the new individuals of the intermediate population (see also figure 3.3).

selection and offspring creation

In principle, the selection of potential parents from among a given population is distinct from the actual process of mating and reproduction. Nevertheless, for practical reasons, any selection scheme that is to be used for parent selection also assumes the task of managing the creation of offspring like it is implemented by the `Genome`s `crossover(·)` and `mutate()` functions. After calling the corresponding `newGeneration(·)` method, the population will already contain the resulting offspring. A survivor selection process, on the other hand, will merely discard superfluous individuals from the population depending on their fitness values.

In both cases, an `HSelectionScheme` might change the size of the passed `HPopulation` object and the combination of parent and survivor selection scheme is to be tuned to effectively leave the size of the population constant. For generational evolutionary algorithm models (see section 3.4.1), it can be practical to implement `HSelectionScheme`s that combine both selection steps to one `newGeneration(·)` method.

efficiency considerations

In contrast to the fitness estimation process and the variation operators, selection procedures are only applied once or twice per generation. In so far, they can be regarded as the least time critical part of an evolutionary algorithm.

For that reason, the realization of a new `HSelectionScheme` subclass is straight forward in that the selection process can directly benefit from the functionality of the `HObject` framework. All variable parameters of a selection scheme are to be represented by adequate `HValue` objects and the `newGeneration(·)` method can be implemented to account for enhanced flexibility without giving rise to an intolerable loss of efficiency.

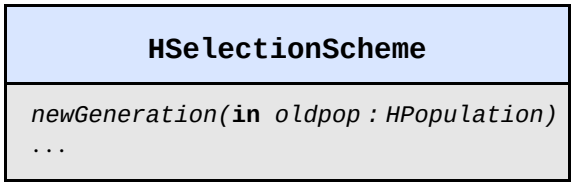


Figure 7.18: *Short UML description of the `HSelectionScheme` class. The actual selection scheme is implemented by the `newGeneration(·)` method.*

```
void HSimplePop::evaluate() {
  Fitness thefitness;
  for (int i=1;i<size();++i)
    genomeAt(i)->translate(netdat_);
    netman_->run(*netdat_);
    thefitness = fitfunc_(netdat_, compdat_);
    genomeAt(i)->setFitness(thefitness);
  }
  sortPopulation();
}

void HSimplePop::nextGeneration(HSelectionScheme* sel) {
  sel->newGeneration(*this);
}
```

Figure 7.19: *C++ implementations for the* `evaluate()` *and* `nextGeneration(·)` *methods of the exemplary* `HSimplePop` *class. In* `evaluate()`*, each genome in the population is translated into a network description, executed on the hardware and evaluated for its fitness.* `nextGeneration(·)` *uses the passed selection scheme to transform the population. Especially with regard to this second method, more sophisticated evolutionary algorithms are likely to require more complex implementations. Note, however, that in spite of their simplicity, the provided examples are in fact fully functional.*

7.4.5 Evolutionary Algorithm Practice

The preceeding sections have described HEAF by providing isolated views on its single base classes. The following paragraphs shall illustrate more explicitly how these components can be combined. For this purpose, the relevant parts of a simple exemplary evolutionary algorithm implementation will be presented. Finally, the chapter will be concluded with a brief discussion of how the introduced classes can cope with extensions to the basic evolutionary concept like they have been introduced in section 3.3.4.

A Simple Evolutionary Algorithm Example

Consider the classes `HMyFitFunc`, `MyGenome`, `HMyGenome`, `HSimplePop`, and `HSimpleSelect` that are each exemplary subclasses of `HFitnessFunction`, `Genome`, `HGenome`, `HPopulation`, and `HSelectionScheme`, respectively. The implementation details of the genetic coding and the actual form of the fitness function shall in the following be ignored. It suffices to ensure that each class completely and reasonably implements the interface provided by its corresponding base class.

simple `HPopulation` *example*

Figure 7.19 shows simple versions of the `evaluate()` and `nextGeneration(·)` methods as they could be implemented by the `HSimplePop` class. Every `HSimplePop` object is assumed to have access to an `HNetMan` instance via a pointer named `netman_` as well as to a pair of `HNetData` objects — `netdat_` and `compdat_` — that are used to temporary hold the actual network descriptions of the single individuals and to store the desired target outputs, respectively. Be-

```
void HSimpleSelect::newGeneration(HPopulation& pop) {
  int last = pop.size() - 1;

  Genome*& worst  = pop.genomeRank(last);
  Genome*& bad    = pop.genomeRank(last-1);
  Genome*& better = pop.genomeRank(1);
  Genome*& best   = pop.genomeRank(0);

  delete worst; worst = best->clone();
  delete bad;   bad   = better->clone();
  worst->crossover(bad);
  worst->mutate();
  bad->mutate();
}
```

Figure 7.20: *Simple C++ implementation for the* `newGeneration(·)` *method of the exemplary* `HSimpleSelect` *class. Effectively, the worst two individuals are replaced by offspring of the best and second best. Although it is clearly basic, this example already demonstrates how arbitrary selection schemes can easily be implemented. Note how the creation of offspring is realized merely by using the methods of the* `Genome` *interface.*

sides that, it can call the designated fitness function via the `fitfunc_` function pointer[8].

exemplary `evaluate()` *method*

Employing the functionality provided by these components, the implementation of `evaluate()` is strikingly simple. The single genomes in the population are sequentially translated into network descriptions that can be executed on the neural hardware via the HAL interface, and the resulting network responses are used to calculate the individuals' fitness values.

Besides noting that the interaction with the hardware is essentially reduced to one line of code, it is also worth emphasizing that the shown implementation is unaffected by the peculiarities of the utilized genotypic representation and the applied fitness function. It could readily be used with various different `HGenome`, `HGenome` and `HFitnessFunction` subclasses without modification.

For convenience, the population is sorted according to the fitness values of the individuals once the evaluation is completed which is done mainly to simplify the implementation of the `genomeRank(·)` method. Both functions are realized in a straight forward fashion and shall therefore not to be discussed here in detail.

simple version of `nextGeneration(·)`

In comparison to `evaluate()`, the code of `nextGeneration(·)` is even more concise. The population is transformed via a single call to the `newGeneration(·)` method of the passed `HSelectionScheme`. This is sensible as long as the parent selection, the creation of offspring, and the survivor selection can be implemented in one method. A simple example for such a selection scheme is given in figure 7.20.

The shown procedure effectively replaces the worst two individuals of the pop-

[8]The actual access to these objects and functions is provided via several built-in methods that have so far been omitted for simplicity. Exhaustive information about this kind of organizational details can be inferred from the HANNEE source code and its documentation.

```
void HMyEvoAlgo::exec() {
  paused_ = false;
  population_->evaluate();
  while (curriter_ < maxiter_->get() && !paused_) {
    population_->nextGeneration(selector_);
    population_->evaluate();
    curriter_++;
  }
}
```

Figure 7.21: *The* `exec()` *method of a simple exemplary* `HMyEvoAlgo` *algorithm. During execution, the hosted population is repeatedly and alternately transformed by the used selection scheme and evaluated on the hardware. Since the necessary functionality is completely encapsulated by the used HEAF classes, the implementation of* `exec()` *retains a remarkable generality.*

ulation with offspring of the two best individuals (it is assumed that the fitness decreases with increasing rank). Again, this scheme can be implemented solely on the basis of the general `Genome` and `HPopulation` interfaces regardless of the actually used subclasses. While this example can hardly claim to represent a feasible selection scheme suitable for practical applications, it does indicate how more sophisticated procedures can be realized.

basic implementation of `exec()`

Figure 7.21, finally, presents the `exec()` method of an exemplary `HMyEvoAlgo` class that illustrates how `HSimplePop` and `HSimpleSelect` can be combined to form a complete evolutionary algorithm. The algorithm has access to the corresponding objects via the `population_` and `selector_` pointers. In every iteration, a new generation is created from the previous population and the individual networks are tested for their fitness. Given the shown realizations of `HPopulation::evaluate()` and `HSimpleSelect::newGeneration(·)`, this approach does in fact leave room for improvement as a large unchanged fraction of the population is unnecessarily retested on the hardware in each iteration.

On the other hand, the code presented in figure 7.21 retains a remarkable generality and could even be viewed as a formal notation of the basic generational evolutionary approach itself (see section 3.4.1). It can therefore be anticipated that, despite of its simplicity, this version of the `exec()` method serves as a suitable basis for a large variety of evolutionary algorithms simply by exchanging the used `HPopulation` and `HSelectionScheme` subclasses.

outlook

Although the shown methods have been simplified to omit several important organizational aspects that are undoubtedly essential for an evolutionary training algorithm to be usable in practice (e.g., defining the used input patterns, creating the initial population, monitoring the populations fitness distribution during execution, defining an appropriate termination condition, etc.), they demonstrate the basic methodology to create functioning algorithms in HEAF. The devil is said to be in the details and this clearly applies also to software engineering. The code of a complete and usable evolutionary algorithm implementation is likely to be much more elaborate than the presented example. Nevertheless, the main concept

of exploiting the inherent modularity of evolutionary computation for the realization of a flexible evolutionary algorithm software framework holds. It remains to be investigated whether the proposed design withstands a confrontation with the extensions to the basic evolutionary approach that have been brought forward in section 3.3.4.

Extensions to the Basic Concept

island-model algorithms

Among other things, encapsulating the details of the population management into a dedicated `HPopulation` class yields the advantage that multiple populations can easily be combined to implement various forms of island-model approaches (section 3.3.4).

diffusion-model algorithms

In contrast, the realization of diffusion-model algorithms requires to extend the functionality of the `HPopulation` and `HSelectionScheme` classes by deriving appropriate subclasses. A corresponding `HPopulation` derivative needs to account for the desired topology such that the hereby defined neighborhood of a given individual is well-defined and can readily inferred by the `newGeneration(·)` method of the used `HSelectionScheme` object.

One possible way to introduce this functionality is to expand the interface of the `HPopulation` subclass by a dedicated method, e.g.,

```
neighborhood(in ind : Integer, out neigh : Set(Integer)),
```

which returns the set of indices `neigh` of all population members that form the neighborhood of a given individual `ind`. This is probably the most intuitive approach but it is limited in so far as an accompanying `HSelectionScheme` subclass has to be implemented as well that actually employs this function during the selection process. Hence, although the new `HPopulation` class would provide the necessary topology information, the actual implementation of the diffusion-model concept would be deferred to the new `HSelectionScheme` derivative.

An alternative approach exploits the fact that the `HPopulation::nextGeneration(·)` function is designed as a callback routine (see also figure 7.19): Instead of calling `HSelectionScheme::newGeneration(·)` on itself, a specialized diffusion-model population can repeatedly apply the desired selection procedure to appropriate subpopulations. These neighborhoods can be represented by internal objects of another specialized `HPopulation` subclass that do not actually host their own genomes. Instead, their `genomeAt(·)` and `genomeRank(·)` methods are to return handles to the corresponding genotypes of the original population. This second solution is advantageous in so far as it allows to employ existing `HSelectionScheme` classes unchanged.

retained modularity and generality

Note that for both variants, the peculiarities of the diffusion-model concept are completely hidden from the actual algorithm implementation. For example, the code presented in figure 7.21 would not need to be changed at all. This insight can be regarded as an additional support for the claim that the inherent modularity of evolutionary algorithms (section 3.3.2) is not compromised by the extensions introduced in section 3.3.4.

similarity measures and speciation

Analog considerations apply to the concept of a similarity metric used for speciation schemes. As suggested in section 3.3.4, the task of quantifying the resemblance between two individuals can be assigned to the genetic representation. In other words, subclasses of `Genome` can provide the necessary information in form of an additional method, e.g.,

```
similarity(in comp : Genome, out sim : Float),
```

which returns the similarity `sim` between the regarded object and the specified partner `comp` in adequate units. Alternatively, a special `HFitnessFunction` object could be used to provide functions that do not actually evaluate the performance of a network in consideration of the target output but calculate some measure for the resemblance between the two passed `HNetData` objects. This second variant can conveniently be used with already existing `Genome` classes, but an estimation of the similarity based on the final networks runs the risk of being less efficient.

again: retained modularity and generality

While the first method measures the distance in the genotype space, the second quantifies the similarity in the phenotype space. In both cases, the thus provided information can be used by dedicated `HSelectionScheme` subclasses and again, the algorithm implementation remains unaffected on the higher level.

The above reflections do not only underline the modularity of the evolutionary computation concept itself. They also demonstrate that HEAF has the potential to serve as a practical and flexible foundation for the efficient implementation of evolutionary training algorithms for mixed-signal neural networks. This functionality constitutes the basis for the development of the efficient evolutionary training strategies introduced in the following chapters.

Part III

Experiments and Results

Chapter 8

A Simple Evolutionary Approach

Whenever you are asked if you can do a job, tell 'em, 'Certainly I can!' Then get busy and find out how to do it.

Theodore Roosevelt

The hardware neural network framework presented in the preceeding chapters allows to implement networks with in the order of 10^4 synapses that can efficiently exploit the inherent parallelism of neural information processing. Networks of the achievable scale and performance constitute a promising approach to tackle challenging real-world problems that involve the processing of large amounts of data. But in order to fully benefit from the potential of the provided hardware resources, suitable training algorithms have to be devised first that can cope with the size of the implemented networks and the complexity of the investigated tasks.

Several strategies have been considered for the design of neural systems that aim to make best use of the available resources and the featured parallelism. Both, the concept of computing without stable states (section 2.3.4) as well as the hierarchical approach (section 2.3.2) have been demonstrated to be successfully applicable to the construction of networks on the presented hardware platform [52] [184].

The experiments described in this and the following chapters investigate the potential of the more traditional chip-in-the-loop training approach. As it has been stated before, highly iterative generate-and-test algorithms ideally complement the realization of neural networks on the HAGEN chip since they can on the one hand account for the peculiarities of the CMOS substrate and the inevitable noise in the analog circuits (sections 2.4.5 and 5.5.1) and can on the other hand take full advantage of HAGEN's high configurational speed (section 5.4.4). Beyond that, the utilization of evolutionary algorithms or any other black-box approaches (see section 4.3) readily avoids the credit assignment problem (see section 2.2.2) that usually impedes the training of multilayer networks of threshold neurons.

It remains that in order to keep up pace with the speed of the networks, the training algorithm itself has to be implemented efficiently. This insight drove the

design of the evolutionary coprocessor that provides flexible means of accelerating a large diversity of evolutionary strategies (see section 6.2). In its current state, the coprocessor supports codings and operators that are primarily suited for the evolution of the mere synaptic weights. More sophisticated representations — like they have been discussed in section 4.2.2 — that allow to simultaneously optimize also the network architecture can, as yet, not fully benefit from hardware acceleration. In general, it can be said that the desire for fast algorithm implementations motivates the use of simple algorithms. Elaborate speciation schemes, computationally expensive fitness measures, and/or ingenious genetic representations run the risk of compromising the speed advantage of the neural network hardware during training.

In turn, the desire for simple algorithms interferes with the ultimate goal of training complex networks for demanding real-world tasks. Besides describing the general training setup that is to serve as the basis for all presented experiments, this chapter will investigate in how far the restriction to simple evolutionary algorithms limits the achievable training success. The obtained results will motivate the development of the improved training strategies introduced in the next chapter.

8.1 Classification Benchmarks

benchmark problems

Pattern classification (see section 1.2.4) is one of the most important types of applications for neural networks [20] [163]. There are a large number of real-world classification problems originating from a variety of application areas that are accepted throughout the neural network community as suitable benchmarks for evaluating the feasibility of network models and novel training approaches.

Within the experiments described in this thesis, evolutionary training algorithms are tested for their ability to train neural networks on the HAGEN chip for a selected set of these common classification benchmarks. This readily allows for a direct comparison of the achieved results with other neural network implementations, training algorithms, or classification procedures. The selected tasks all represent real-world data and are therefore anticipated to capture the main features that characterize realistic pattern categorization applications in practice. The data sets have been obtained from the UCI KDD online archive [90], but some are also included in the "Proben1" collection compiled by Lutz Prechelt [163].

8.1.1 The Classification Tasks

the selected tasks

A total of nine well-established classification benchmarks are used within the described experiments: the breast cancer, diabetes, heart disease, liver disorder, iris plant, wine, glass, *E.coli*, and yeast problems. Each task is represented by a finite set $\mathbb{E}$ of N_e exemplary instances e^α, $1 \leq \alpha \leq N_\mathrm{e}$, that each belong to one of N_c classes C_k, $1 \leq k \leq N_\mathrm{c}$. Every instance e^α of the respective problem is described by a set of N_a attributes whose values are used to form the corresponding input vectors $\mathbf{I}^\alpha$ for the network (see also section 1.2.4).

benchmark	N_c	N_e	N_a
breast cancer	2	683	9
diabetes	2	768	8
heart disease	2	297	13
liver disorder	2	345	6
iris plant	3	150	4
wine	3	178	13
glass	6	214	9
E.coli	8	336	7
yeast	10	1462	8

Table 8.1: *The numbers of classes N_c, instances N_e, and attributes N_a for the nine different data sets used in the described experiments.*

The numbers of classes, instances, and attributes for the different data sets are summarized in table 8.1. More detailed information about the single data sets can be found in appendix B.

8.1.2 Measuring the Generalization Performance

training set and test set

The ultimate goal of neural network training is to yield a network with satisfactory generalization performance (see section 1.2.4). In order to evaluate the ability of a specific network model and/or a training algorithm to produce systems with favorable generalization on a given task, the corresponding data set $\mathbb{E}$ is usually divided into at least two partitions: the training data $\mathbb{E}_t$ and the test data $\mathbb{E}_g$. As the name suggests, the former is used for training. In contrast, the test data set $\mathbb{E}_g$ — also called the generalization data set — is only presented to the network after the training is complete and is used to quantify its ability to correctly classify previously unseen examples.

Fixed Partitionings

classification accuracy

Once the training has been completed, let the achieved classification accuracy of the network on the training data, more specifically, the fraction of correctly classified instances in percent, be denoted with a_t. The corresponding performance on the test set will then be denoted as a_g. Since in practice, the used data sets contain a finite number of examples, the obtained results are observed to measurably depend on the used partitioning of the data into training and test set.

disadvantageous partitionings

This is intuitively clear. Consider the exemplary case of a four-class problem where the training set $\mathbb{E}_t$ contains only instances of class C_1 and C_2 while the test data $\mathbb{E}_g$includes only examples of C_3 and C_4. Even if the network exhibits a satisfactory accuracy on the training data, it is unlikely to perform well on the test set since it has never been taught to differentiate between members of C_3 and C_4. An alternative partitioning that includes a representative number of examples from all four classes in each respective subset is not only likely to yield

networks with better generalization ability, but also permits a more meaningful interpretation of the results.

But even then, the actual compositions of the training and test set remain to influence the observed classification and generalization accuracies. Some of the instances are usually harder to classify than others, e.g., if the classes partly overlap in the input space (compare figures 2.3 a) – c)). Including the majority of these problematic cases in the generalization data is likely to lead to a better performance on the training data and thus to higher values of a_t but will also yield a lower observed generalization accuracy a_g and vice versa.

reproducible partitionings

In summary, it can be said that a given partitioning of the data into training and test set inevitably imposes an unwanted bias to the evaluation of the generalization performance. Hence, for a quantitative comparison with the results of other investigations, it is of vital importance to use exactly the same partitioning [163]. In fact, to achieve representative and sufficiently unbiased estimates, multiple different partitionings should be used. The "Proben1" collection accounts for this issue by specifying three randomly created but fixed partitionings for each task [163].

Cross-Validation

multiple random partitionings

A better estimate for the generalization ability of a classifier system is provided by a more elaborate method commonly known as N-fold cross-validation or rotation estimation. First, the whole data set is randomly divided into N_p mutually exclusive and equally sized subsets $\mathbb{E}_r$, $1 \leq r \leq N_\mathrm{p}$. Second, based on these subsets, N_p different partitionings are constructed that divide the whole data set $\mathbb{E}$ into training and test data: According to the rth partitioning, subset $\mathbb{E}_r$ represents the test patterns while $\mathbb{E} \setminus \mathbb{E}_r$ is used as the training set. The classification and generalization accuracies a_t^r and a_g^r of the investigated classifier system are then evaluated once for each partitioning r and the results are averaged to yield the mean classification and generalization accuracies $\overline{a}_\mathrm{t}$ and $\overline{a}_\mathrm{g}$.

stratified subsets

Each single instance of the task is contained in the generalization set of exactly one partitioning and appears in the training data of the remaining $N_\mathrm{p} - 1$ cases. This way, all instances are treated equally. Taking the average over all obtained generalization accuracies effectively compensates for the unfavorable bias that is likely to be introduced by each partitioning individually. The reliability of the estimate can be improved further by ensuring that the proportion of classes in all N_p subsets is approximately equal. In each subset, the fraction of instances belonging to class C_k then roughly corresponds to C_k's share of the whole data set. This is called stratified N-fold cross-validation.

leave-one-out validation

In the extreme case, the number N_p of subsets equals the number N_e of examples in $\mathbb{E}$ which is commonly known as leave-one-out validation or jackknifing. Depending on the size of the used data set, it can be very expensive to train N_e different networks. While it might be apprehended that the quality of the estimation grows with an increasing number N_p of partitions, it can actually be shown that leave-one-out validation does not necessarily yield the most accurate estimate for the generalization ability of the investigated system [119]. Choosing a number

of partitionings in the order of ten is generally supposed to yield a sufficient accuracy of the obtained results and also constitutes a compromise with regard to the necessary computational effort.

In particular, the results of a random, stratified 10-fold cross-validation can be expected to be sufficiently accurate as to compare them with other investigations that also employ the same scheme but with a different set of random partitionings. Therefore, most of the presented experiments follow a stratified 10-fold cross validation scheme. Only if the obtained results are to be directly compared with previously published results, the experiments are conducted with the same partitionings as reported in the respective publications.

Indeterministic Algorithms and Repeated Validation

the random nature of evolution

The averages $\overline{a}_\mathrm{t}$ and $\overline{a}_\mathrm{g}$ obtained by a stratified 10-fold cross-validation are deemed to be largely independent of the actually used random subsets, and the respective standard deviations Δa_t and Δa_g can be interpreted as to reflect the variance in difficulty between the individual partitionings. However, for highly indeterministic algorithms such as evolutionary strategies which involve a large number of random decisions, even the results of multiple training runs on one and the same training and test data configuration are observed to show a measurable variance. No pair of evolutionary optimization processes yield the same network. Even common gradient-based approaches suffer from a similar problem since the outcome of the training can significantly depend on the random initializations of the weight values. When applying neural networks to real-world applications, it is therefore common practice to train multiple networks on the same data and keep only the best one.

modified cross-validation

In order to evaluate the expected generalization performance of the resulting networks, the used stratified 10-fold cross-validation scheme is easily modified such that on each of the 10 partitionings, a number of N_n networks are trained and only the ones with the best performance on the respective training set are used for the calculation of $\overline{a}_\mathrm{t}$ and $\overline{a}_\mathrm{g}$. For a realistic estimate, it is evidently necessary that the choice of the best network on each combination of training and test data is exclusively based on the respective training set.

But it remains that given a specific network model and training algorithm, it is not initially known in how far a high performance on the training data also implies a high generalization accuracy. Especially in the case of highly indeterministic training algorithms, the latter might therefore show a measurable variance between two experiments, even if only the best of N_n networks is considered in each case. Neither does the used cross-validation scheme account for this variance, nor can the measured standard deviations Δa_t and Δa_g serve to quantify its magnitude.

repeated cross-validation

For this reason, the described cross-validation scheme is applied multiple times. Such a procedure is frequently encountered in literature and widely known as repeated cross-validation. Usually, it is intended to further minimize the impact of the partition bias and therefore, different random divisions of the data are employed for each repetition. In the case of the presented experiments, repeating the measurement rather aims to quantify the effects of the inherently indeterministic nature of the training algorithms. Hence, the single repetitions of the

cross-validation scheme all use the same set of fixed but random partitionings.

the used evaluation scheme

In summary, the performance on a specific task that is to be expected from a given combination of network model and evolutionary training strategy is chosen to be evaluated as follows: A stratified 10-fold cross-validation is performed, such that for each of the 10 partitionings, $N_\mathrm{n} = 10$ networks are trained and only the one with the best performance on the training data is taken to contribute to the averaged accuracies $\overline{a}_\mathrm{t}$ and $\overline{a}_\mathrm{g}$. Using the same set of partitionings, the whole process is repeated $N_\mathrm{r} = 5$ times. The respectively achieved accuracies $\overline{a}_\mathrm{t}$ and $\overline{a}_\mathrm{g}$ are averaged once more to yield the final results A_t and A_g together with the respective standard errors of the mean ΔA_t and ΔA_g. Note that in order to obtain these values, a total of 500 networks is trained. It is therefore expected that they do not only represent a sufficiently unbiased estimate of the generalization performance of the investigated system, but also reasonably quantify the reliability of the used evolutionary training algorithms.

8.2 Network Setup

The HAGEN chip supports a wide range of neural network architectures (see section 5.2). Several experimental investigations have shown that highly recurrent networks [184], strictly layered feedforward topologies with multiple layers [52] [94], as well as architectures with shortcuts [93] or feedback connections [95] can feasibly be implemented. Among other things, the reported results demonstrate that large networks can successfully be distributed over multiple network blocks (see section 5.4.2), thereby paving the way for future experiments that are to spread even larger networks over several chips (see also section 6.3).

8.2.1 General Architecture and Number of Hidden Nodes

fixed architecture

It has repeatedly been stated in the preceeding chapters that choosing the optimal topology or size of a neural network for a given task is anything but trivial. Rather than striving for a thorough optimization of the network architecture, the investigations presented in this thesis aim to provide a proof of principle for the claim that simple and fast algorithms can successfully train networks implemented on the HAGEN chip for realistic pattern recognition task. Therefore, the architecture of the trained network is fixed in advance and the training algorithm merely optimizes the synaptic weights. It is to be emphasized, though, that this is by no means due to any principle restriction of the used hardware neural network framework.

two-layer networks

Strictly layered, feedforward neural networks with one hidden layer are proven to be universal approximators (see section 2.2.1), and it has therefore been chosen to employ feedforward architectures with at most one hidden layer for all presented experiments. Given one of the classification benchmarks introduced in the foregoing section, the necessary number of input nodes N_in is related to the number of attributes N_a, while the size of the output layer N_out depends on the number of classes N_c. The exact values are eventually determined by the way in which the individual input attributes are presented to the network and by how

the network is desired to code its output. These topics will be addresses in the following sections.

number of hidden neurons

The number of hidden nodes in each network is decided to be scaled according to the number of classes in the investigated benchmark. For simplicity, all networks contain a fixed number of $N_{\text{hid}}^{\text{c}} = 6$ inner neurons per category, such that the total size of the hidden layer is in each case given by $N_{\text{c}} \cdot N_{\text{hid}}^{\text{c}}$. This is intended to account for the fact that with an increasing number of classes, more complicated partitionings of the input space need to be performed which in turn implies a larger required number of hidden neurons.

The particular choice of $N_{\text{hid}}^{\text{c}} = 6$ neurons per class is motivated by practical considerations rather than theoretical predictions: Given a maximum of ten classes (like they are included in the yeast problem), the hidden layers of all networks fit on one network block of the HAGEN chip (see section 8.2.4). Apart from that, it is understood that the employed network sizes do not necessarily represent the respective optimal choice (see also section 8.4.3).

8.2.2 Input Representation

types of attributes

With only a few exceptions, all attributes that characterize the individual instances of the investigated classification tasks fall in one of three categories: binary, nominal, or (quasi-)continuous numerical variables. Attributes of the first kind assume only two different states and are therefore adequately represented by one bit. Nominal attributes can take one of $m > 2$ possible values that do not necessarily obey an ordering relation (like, e.g., *north*, *south*, *east*, and *west*). The values of numerical attributes, finally, are specified by either integer or floating-point numbers.

There do appear some ordinal attributes within the data sets that assume one of $m > 2$ alternative values which actually show a qualitative order (like, e.g., *small*, *medium*, and *large*). Attributes of this type could reasonably be mapped onto corresponding numeric variables, but in all encountered cases, the respective number m of potential values turns out to be lower than 4. For simplicity and in agreement with other investigations [163] it has therefore been chosen to treat these parameters like nominal attributes.

If the values of numerical or nominal attributes are to be presented to a network on the HAGEN chip, they first have to be mapped onto the binary inputs of the used network blocks. Nominal and continuous attributes are coded in different ways.

Nominal attributes

1-of-m encoding

Although a nominal parameter with m possible values can in principle be coded as an integer number, e.g., between 0 and $m - 1$, it is common practice to use a different representation known as 1-of-m encoding. According to this approach, m binary inputs are reserved for the attribute. In order to specify the jth value, the jth input node is activated, while the remaining nodes are kept deactivated.

Effectively, this procedure could be seen as to replace a nominal attribute of

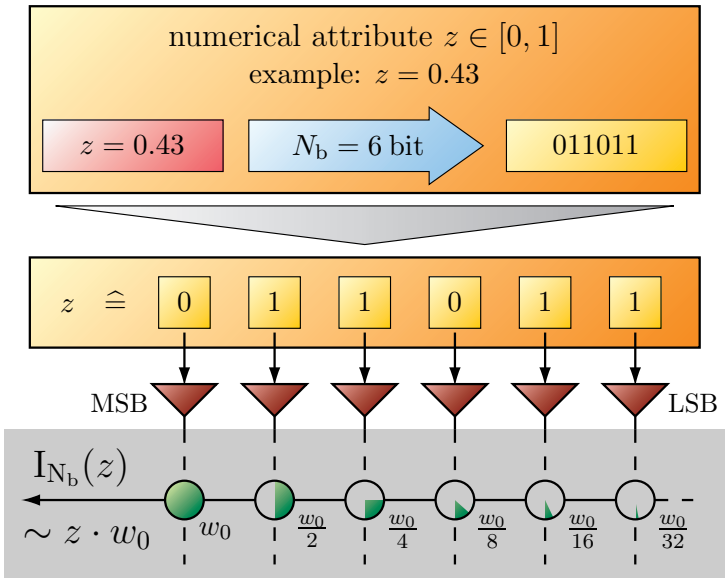


Figure 8.1: *The value z of a continuous attribute is transformed into an N_b-bit integer number that can be applied to a group of N_b input neurons. The corresponding synaptic weights are coupled such that if the weight which corresponds to the most significant bit is assigned a desired value w_0, the succeeding synapses are set to $w_2 = w_0/2$, $w_3 = w_0/4$, etc. The hereby generated current $I_{\mathrm{N_b}}(z)$ is proportional to the original value z.*

m possible values by m binary attributes. Within the presented experiments, it has been chosen to adopt the 1-of-m encoding for all encountered nominal variables. This is not only conform to other reported investigations [163], but is also particularly well suited for the binary inputs of the HAGEN network blocks.

Numerical attributes

n-bit integer numbers

In neural network experiments, numerical input values are commonly scaled to lie within $[0, 1]$ using a linear function [163] [235] [236]. For many of the investigated tasks, the numerical attributes are already provided in this format. In order to be applied to the binary inputs of the HAGEN chip, the resulting values are coded as N_b-bit integers that can be applied to a group of N_b binary input nodes. This is illustrated in figure 8.1.

weight coupling

The application of an integer number $z \in 0, \ldots, 2^{N_\mathrm{b}}$ to the corresponding group of N_b input nodes is desired to yield an input current $\mathrm{I}_{\mathrm{N_b}}(z)$ to the respective neuron which is proportional to the original attribute value. This can be achieved by coupling the weights of the involved synapses such that the connection leading to the most significant bit (MSB) is assigned a given weight $w_1 = w_0$ and the succeeding synapses are set to $w_2 = w_0/2$, $w_3 = w_0/4$, etc. The input node that represents the least significant bit then contributes with a strength of $w_{N_\mathrm{b}} = w_0 \cdot 2^{-(N_\mathrm{b}-1)}$. Note that the absolute values of the individually programmed weights w_i remain to be restricted to integer numbers between -1023 and 1023 and might have to be rounded accordingly (see section 5.4.3).

benefits

Coupling the N_b synapses of a multi-bit input in the proposed fashion considerably decreases the number of free parameters to be optimized by the training algorithm. The resulting connection is completely specified by one single value w_0. In order to warrant the linearity of the hereby implemented digital to analog converters, it is necessary to at least calibrate the row-wise averages of the synapse offsets within the used HAGEN chip (see section 5.5.1). Previous investigations have revealed that such N_b-bit input nodes with a precision of up to $N_\mathrm{b} = 6$ can reliably be implemented on a HAGEN ASIC and retain a satisfactory linearity of the produced postsynaptic current $\mathrm{I}_{\mathrm{N_b}}(z)$ [186]. The reported measurements use $w_0 = 400$ and a maximum postsynaptic current $\mathrm{I}_\mathrm{ps}^\mathrm{max}$ of $45\,\mu$A (see section 5.4.3).

known issues

Within the experiments discussed in this work, $\mathrm{I}_\mathrm{ps}^\mathrm{max}$ is set to $22\,\mu$A in order to better exploit the dynamic range of the neuron (see section 5.4.3) which leads to a precision of the individual weights of about 1 % of the synaptic dynamic range. Compared to the above results, this can be expected to impair the accuracy and linearity of the generated currents, especially for small values of w_0. On the other hand, a comparably small synaptic weight also indicates that the corresponding input attribute is not of significant importance for the decision of the receiving neuron. Therefore, it can be anticipated that a decreased accuracy of the effectively contributed current $\mathrm{I}_{\mathrm{N_b}}(w_0, z)$ does not necessarily compromise the neuron's functionality. At last, chip-in-the-loop training algorithms can readily account for these issues and tune the remaining free parameters to yield an optimal balance between the exploitation of the available dynamic range and the required accuracy of the generated currents.

used coding scheme

Against the background of these considerations, it has been decided to code all numerical attributes as follows: First, if necessary, the values are scaled to lie between 0 and 1. More specifically, given the attribute's minimum and maximum values x_min and x_max in the data set, every value x is transformed according to

$$x' = \frac{x - x_\mathrm{min}}{x_\mathrm{max} - x_\mathrm{min}}. \tag{8.1}$$

Second, the obtained numbers x' are rescaled and rounded to be coded as 6-bit integers. The weights of the corresponding input nodes are coupled to form 6-bit analog to digital converters in the fashion described above, such that the resulting synaptic connection can completely be characterized by only one weight value w.

exceptions

There are two exceptions to this rule:

- The data that is obtained from the UCI KDD archive specifies all input attributes of the breast cancer problem as integer numbers between 0 and 10. Instead of rescaling these values to $[0, 1]$ and then representing the results by 6-bit integer numbers, the original attribute values are directly coded as 4-bit integers. Hence, the corresponding networks utilize multi-bit integer inputs with a precision of only 4 bit.

- The attributes of the *E.coli* and yeast data sets are already provided as numerical values between 0 and 1. A closer investigation reveals that two of the attributes in the *E.coli* data set assume only two different values each, and the same applies to one attribute of the yeast problem. These three

benchmark	No. of attributes			No. of binary inputs
	binary	nominal	numerical	
breast cancer	—	—	10	40
diabetes	—	—	8	48
heart disease	3	4	6	52
liver disorder	—	—	6	36
iris	—	—	4	24
wine	—	—	13	78
glass	—	—	9	54
E.coli	2	—	5	32
yeast	1	1	6	40

Table 8.2: *The numbers of binary, nominal, and numerical attributes for each task, as well as the total resulting numbers of required binary input nodes to the respective networks. Note that the breast cancer,* E.coli*, and yeast problems require special treatment. The details are discussed in the text.*

parameters are therefore chosen to be treated as binary variables. Furthermore, one attribute of the yeast problem has only three different values in the whole data set. It is coded as a nominal attribute.

binary input strings

The resulting numbers of binary, nominal, and numerical attributes for each task are listed in table 8.2. The binary representations of all input attributes that specify a given instance e^α of the investigated task are concatenated to form a linear string of bits s^α that serves as the input to the network. The resulting lengths of the binary input strings for each task correspond to the respectively required number of input nodes and are also included in table 8.2.

8.2.3 Output Representation

integer coding

After the application of an input pattern s^α, it is the purpose of the network to respond with the correct class label k, $e^\alpha \in C_k$. In principle, the class label k is an integer number $1 \leq k \leq N_\mathrm{c}$ where N_c is the number of classes in the investigated problem. Hence, the network could reasonably code its response in the form of a single numerical value. In the case of the HAGEN chip, the N_out binary outputs of the implemented network might be interpreted as an N_out-bit integer number that is then assumed to be the label of the predicted class.

1-of-m output coding

This kind of output encoding is problematic in so far as it forces the network to map its internal representation of the prediction onto an arbitrary labeling that has been fixed externally. In analogy to the input encoding of nominal attributes, many investigations on pattern classification rather employ the 1-of-m representation to also code the network response, i.e., each of the possible classes is assigned one output neuron [163] [235] [236]. It is common to consider neurons with continuous activation functions and in response to an input pattern s^α, the output with the highest activation is then interpreted as to specify the network's prediction for the correct class. This output representation suits the operation

of neural networks far better than an encoding of the class label by an integer number (compare figures 2.2, 2.4, 2.5, 2.6, 2.7, and 2.8).

Using neurons with sigmoidal activation functions (see section 1.2.3), the individual output values O_k can be interpreted as the predicted probabilities $P_k^\alpha = P(C_k \mid e^\alpha)$ that the applied instance belongs to the respective class C_k [20] (see also section 2.1.3). It is a notable advantage of this approach that it allows to infer the confidence of the network's prediction. If multiple output neurons exhibit nearly equal values or if even the one with the highest activation yields only a negligible response, the classification of the respective instance is evidently difficult (see also figure 2.5). For practical applications, this information can be of vital importance for the user.

multiple outputs per class

In the case of binary outputs like they are implemented on the HAGEN chip, the response of the network can only unambiguously be interpreted if exactly one output neuron is activated and all others remain silent. In order to promote a more differentiated network response, it stands to reason to assign a number $N_{\text{out}}^{\text{c}} > 1$ of outputs to each class. It then remains to be specified, how these output neurons are desired to quantify the individual probabilities P_k^α.

used coding scheme

Similar to the combination of input nodes for multi-bit integer inputs, it is possible to group $N_{\text{out}}^{\text{c}}$ binary output neurons to act as an integer output. Using $N_{\text{out}}^{\text{c}}-1$ feedback connections and $(N_{\text{out}}^{\text{c}}-1)N_{\text{out}}^{\text{c}}$ appropriately tuned weights, such a group of output neurons can measure its analog input activity and represent it as an integer number between 0 and $2^{(N_{\text{out}}^{\text{c}}-1)}$ [186]. For the investigations presented in this work, a simpler approach has been chosen: After the application of an input pattern s^α, the class with the highest number of activated output neurons is considered to be the prediction of the network. Given a fixed number $N_{\text{out}}^{\text{c}}$ of outputs per class, this encoding does not achieve the same resolution as the representation by an $N_{\text{out}}^{\text{c}}$-bit integer, but it yields the advantage of not binding additional resources and being straight-forward to implement.

The number of outputs per class is fixed to $N_{\text{out}}^{\text{c}} = 4$. It is expected that this number constitutes a good compromise between the achievable resolution and the required resources. Still, this initial choice retains a certain arbitrariness. A thorough optimization of the network parameters might yield another value and this topic will returned to in section 10.3.1.

8.2.4 Implementation on the HAGEN ASIC

overall network setup

Summarizing the considerations of the preceeding sections, figure 8.2 a) schematically illustrates the chosen network setup. For a given categorization problem with N_{c} classes, the network contains a single hidden layer of $6 \cdot N_{\text{c}}$ inner neurons — $N_{\text{hid}}^{\text{c}} = 6$ per class — that are homogeneously connected to all input nodes as well as to all outputs. While each class is assigned its own group of $N_{\text{out}}^{\text{c}} = 4$ output neurons, the exact number of binary input nodes is determined by the numbers and types of input attributes that characterize an individual instance of the investigated task (see table 8.2). As described in section 8.2.2, some or all of the input nodes are associated in groups to form N_{b}-bit integer inputs for the corresponding numerical attribute values. By convention, N_{b} is set to 6 for all

tasks, except for the breast cancer problem where $N_\mathrm{b} = 4$ (see section 8.2.2).

multi-block setup

In order to be implemented on the HAGEN chip, the resulting networks are distributed over multiple blocks as it is shown in figure 8.2 b). The first layer is realized on block a. The output states of the involved neurons after the first network cycle are fed into the input nodes of block b which implements the output layer. Hence, the actual network response is available after the second cycle. For tasks with only a few classes, the resulting networks could easily be fit on single blocks and use appropriate feedback connections. But in anticipation of the extensions introduced in the next chapter and in order to treat all tasks in a similar fashion, the shown multi-block setup is used for all investigated problems.

The choice of $N_\mathrm{hid}^\mathrm{c} = 6$ hidden neurons per class allows to fit all involved networks on a maximum of two blocks. For the yeast problem with its ten classes, nearly all outputs of block a have to be connected to the corresponding inputs of block b. Regarding the limited, hard-coded inter-block connections of the HAGEN prototype (see section 5.4.2), this implies to use a block of the left side for block a and its counterpart on the right side for block b (compare figure 5.7).

unused resources

Not all input nodes and neurons of the two blocks are needed to implement the desired networks. Output neurons of block b that do not contribute to the response of the implemented network are ignored. Any redundant feedback or inter-block connections are switched off and the weights of all unused synapses are set to 0. The unused inputs of both blocks are permanently deactivated[1].

internal neuron bias

In this context, it is to be noted that apart from those input nodes that are used to code the actual attributes of the respective task, one input per block is reserved for the implementation of internal bias values for the neurons (see section 1.2.3 and figure 1.8). These nodes are permanently set to an input of one and the weights of the corresponding synapses are allowed to be adjusted by the training algorithm.

8.3 The Evolutionary Training Algorithm

With the architecture of the networks being fixed, it remains the task of the training algorithm to find a suitable set of weight values that yields a satisfactory classification performance on the training data. It has repeatedly been argued in the preceeding sections that in order to best exploit the speed of the used neural network hardware, the training is preferably implemented as fast as possible.

use evolutionary coprocessor

Within the used training setup, the synaptic weights are optimized by an evolutionary chip-in-the-loop algorithm that is partly implemented in software but employs the evolutionary coprocessor (see section6.2) for the genetic representation of the individuals and the application of suitable variation operators.

[1]Note that this does not apply to the respective input nodes with indices 126 and 127 that are used for the neuron offset calibration (see section 5.5.4).

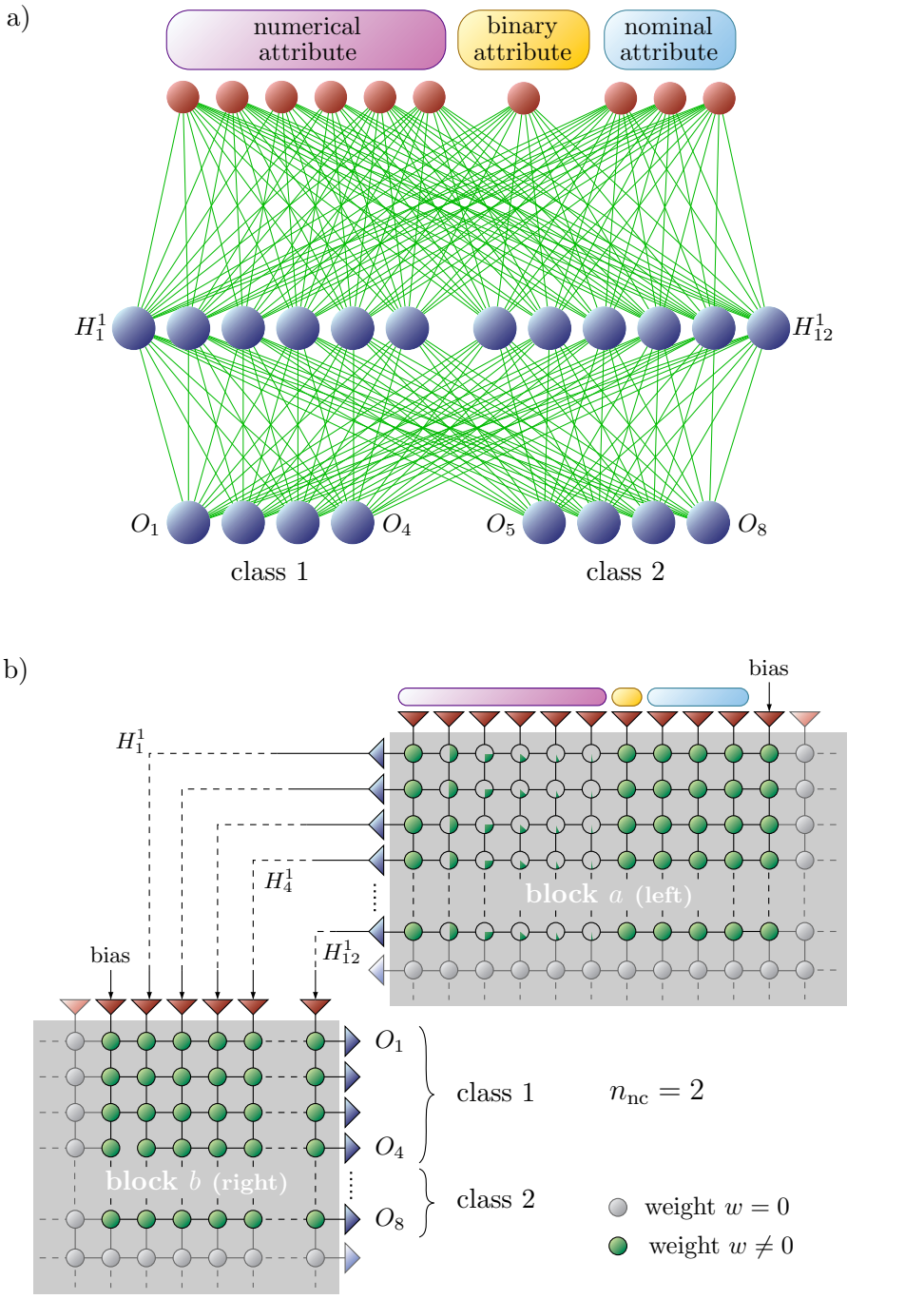


Figure 8.2: ***a)*** *Schematic of the chosen network setup for an exemplary problem with* $N_c = 2$ *classes and* $N_a = 3$ *input attributes: One numerical, one binary, and one nominal parameter. The network contains* $N_{hid}^c = 6$ *hidden neurons and* $N_{out}^c = 4$ *outputs per class.* ***b)*** *Implemented on the HAGEN chip, each network employs two blocks. The first layer is implemented on a left block, while the second layer is hosted by the corresponding block on the right side. Unused synapses are set to zero and unnecessary feedback connections are deactivated. The chosen architecture requires* $n_{nc} = 2$ *network cycles.*

8.3.1 Genetic Representation and Operators

genomes, chromosomes, and genes

Every individual of the processed population defines a set of weight values for all synapses that are utilized to implement the corresponding network on the HAGEN chip. The used genetic representation follows a direct encoding scheme (see section 3.4.2). The genomes are hosted by the evolutionary coprocessor (see section 6.2) and are divided into multiple chromosomes, such that each chromosome codes the weights of all used synapses that lead to one single neuron of the HAGEN chip (see figure 8.3). Within one chromosome, the single weight values are represented by genes that are linearly arranged to form a one-dimensional array. Each gene is an integer number between -1023 and 1023 that directly specifies the weight value of the targeted synapse.

coding and noncoding genes

For technical reasons (see section 6.2.2), each chromosome contains genes for all 128 weight values of the coded neuron, even if not all synapses are used for the implemented network. As a side effect, the position of a gene within its chromosome immediately and uniquely specifies the coordinate of the corresponding synapse. All unused genes are set to zero and are marked as deactivated to exclude them from being affected by the mutation operators (see also section 6.2.2).

coupled genes

As it has been discussed in section 8.2.2, some of the weight values need to be coupled appropriately in order to form multi-bit integer inputs. Apart from fixing unused genes to zero, a given gene g_i can be specified to always assume half of the value of the previous gene: $g_i = g_{i-1}/2$ (see section 6.2.3 and compare figures 6.5 and 8.3). For an N_b-bit integer input, N_b consecutive genes of the corresponding chromosome are associated to form a group where only the first position is actually subject to the genetic operators and the remaining $N_\mathrm{b} - 1$ genes are successively determined by the value of their respective predecessor. These genes are then referred to as being coupled. Since they all eventually and exclusively depend on the value of the first gene, the $N_\mathrm{b} - 1$ associated positions are affected by mutation and crossover only indirectly.

Genetic Operators

uniform and Gaussian mutation

Two different single-gene mutation operators are employed: A uniform version and a nonuniform mutation with Gaussian distribution (see section 3.4.3) that are each applied with the respective probability $\rho_\mathrm{m}^\mathrm{u}$ and $\rho_\mathrm{m}^\mathrm{g}$. The two forms of mutation occur independently, i.e., a given gene can in principle be subject to both mutations with a probability of $\rho_\mathrm{m}^\mathrm{u} \cdot \rho_\mathrm{m}^\mathrm{g}$. In general, the uniform mutation is always applied first and any potential further Gaussian mutation is performed afterwards (see figure 6.5).

used crossover operators

The evolutionary coprocessor supports the implementation of various forms of crossover and for the experiments presented in this thesis, three different alternative operators are investigated. All of the employed recombination procedures operate on chromosome pairs and are applied with probability ρ_c. In other words, for each pair of corresponding chromosomes that are supplied by the two mating individuals, it is decided separately whether crossover is applied or not. Besides common one-point and two-point crossover, a simple chromosome exchange op-

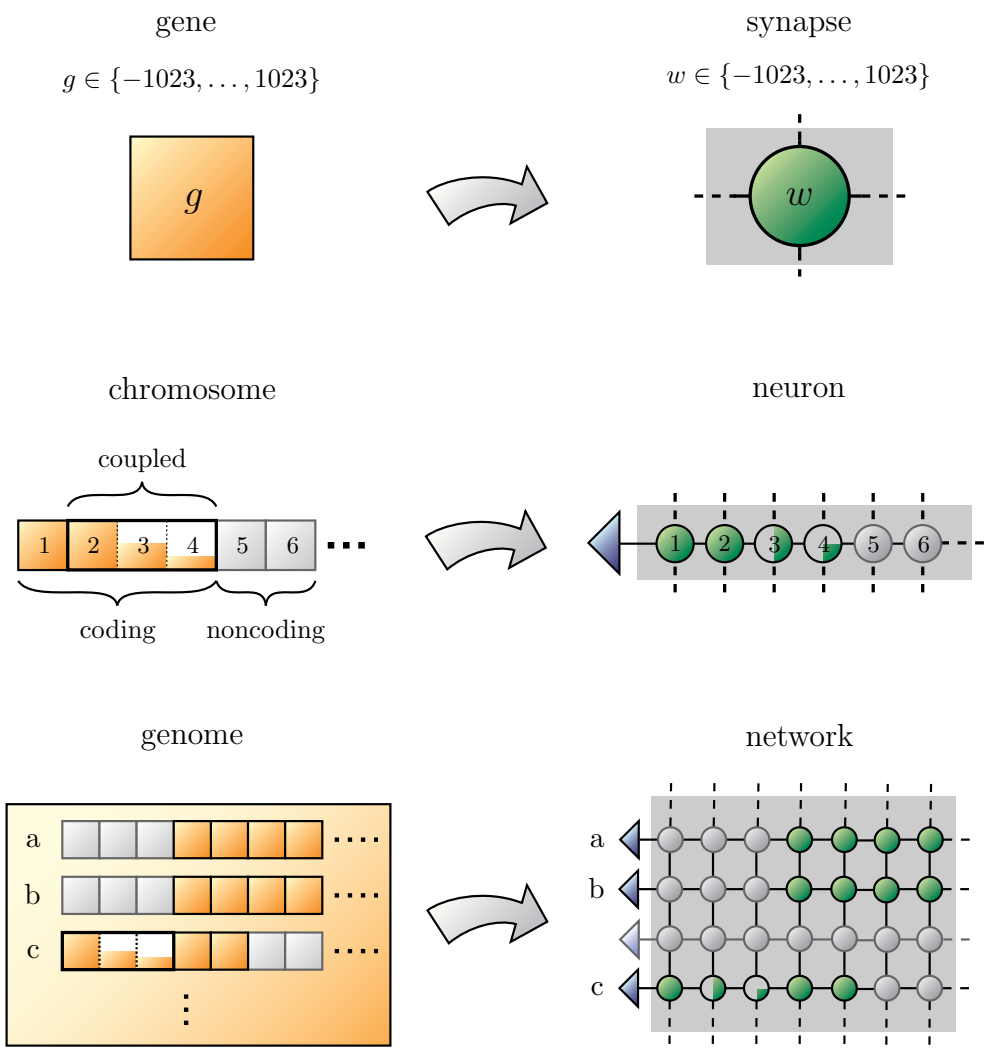


Figure 8.3: *The basic indivisible unit of the genome is the gene (top). Each gene directly codes the weight of one synapse in the form of an integer number between -1023 and 1023. A chromosome (middle) summarizes all genes that code the weights of all synapses that lead to one output neuron of one network block on the HAGEN ASIC. Within a chromosome, the single genes are arranged linearly. Since not all synapses of every neuron are actually used by the implemented network architecture, some of the contained genes are fixed to zero and are not modified by the mutation operators. In contrast to the coding genes that represent adjustable, non-zero parameters of the evolved network, they are referred to as noncoding genes. In order to implement the desired multi-bit integer inputs (see section 8.2.2), a gene g_i can be linked to its predecessor g_{i-1} as to always obey $g_i = g_{i-1}/2$ (compare figure 6.5). For an N_b-bit integer input, N_b genes are associated to form a group where only the first position is actually subject to the genetic operators and the remaining $N_\mathrm{b} - 1$ genes are successively determined by the value of the respective previous gene. These N_b genes are also denoted as being coupled. The whole network, finally, is coded by a genome (bottom) that contains one chromosome for each neuron in any of the four blocks of the HAGEN ASIC that is used by the actual network architecture.*

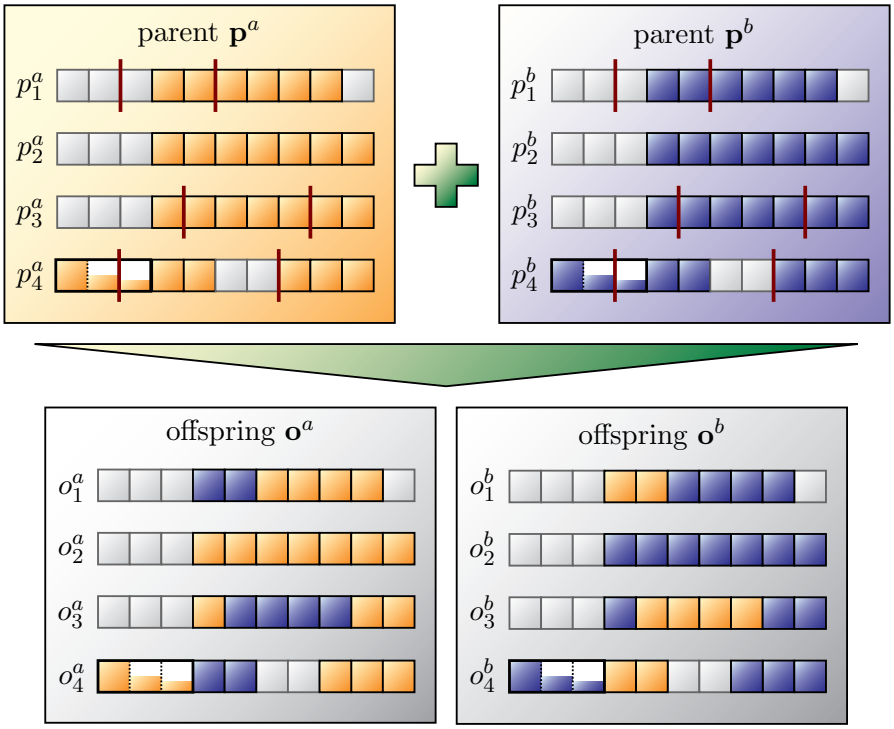


Figure 8.4: *The used recombination operators process one pair of chromosomes at a time and modify it with a fixed probability ρ_c. This figure schematically illustrates a simplified example with reduced chromosome length. A two-point crossover is shown, i.e., for each pair of chromosomes that is actually recombined, two new random cut points are generated. Here, it is assumed that the second chromosome pair turns out to be not crossed: The corresponding chromosomes remain unchanged. None of the used crossover operators distinguishes between coding and noncoding genes (see also figure 8.3). The exchange of genes that are linked to their predecessor (see text) does not have any effect, as long as the first gene of the group is not transferred as well (see the last shown chromosome pair).*

erator is considered where each pair of chromosomes is swapped among the two involved genomes (see section 3.4.4).

During a given evolution run, only one single operator is consistently used for all recombination processes, i.e., the different forms of crossover are not mixed. Furthermore, it is to be noted that none of the applied crossover operators distinguishes between coding and noncoding genes, i.e., those that code the weight of a synapse that is actually used by the represented network and those that are fixed to zero.

default: two-point crossover

In general, the recombination of two genomes yields two new offspring. For the example of the used two-point crossover operator, the basic overall procedure is schematically illustrated in figure 8.4. By default, the two-point crossover is utilized for the majority of the following experiments and it will explicitly be noted if another operator is used.

8.3.2 Selection Scheme and Evolution Parameters

While the actual genetic material of the population is hosted and processed by the evolutionary coprocessor, the hereby defined genetic representation is encapsulated into a dedicated `HGenome` subclass of the HEAF software framework (see section 7.4). Within the used setup, both, the fitness calculation and the selection scheme are realized purely in software and are wrapped into corresponding `HFitnessFunction` and `HSelectionScheme` subclasses.

generational replacement

The implemented algorithm processes a population of μ individuals and follows a generational replacement scheme (see section 3.4.1). Since several theoretical investigations suggest that algorithms with elitist selection exhibit more favorable convergence properties than those without [50] [174], the best n_{el} genomes of the current population can be taken over to the next generation unchanged. For the described experiments, n_{el} is fixed to 1.

tournament selection

The rest of the new generation is generated as follows: First, a common tournament selection process with a tournament size of $\tau = 2$ is used to fill a mating pool of $\mu - n_{\mathrm{el}}$ genomes drawn from the preceeding population. The members of the mating pool are then randomly grouped into pairs and each mating pair is recombined with a probability $\rho_{\mathrm{r}} = 90\,\%$ using the selected crossover operator. Finally, the respective offspring are made subject to mutation[2]. Together with the n_{el} unmodified genomes from the previous population, the resulting genotypes form the new generation.

small populations and strong mutation

Against the background of what has been discussed in section 4.1.2 and given the simplicity of the utilized genetic coding, it is to be anticipated that the described evolutionary approach will potentially suffer from the negative impact of the permutation problem. Aiming for a minimization of these effects, a comparably small population size of $\mu = 20$ is chosen (see also section 4.1.2). In order to compensate for the emergent increased susceptibility to premature convergence (see section 3.3.1), the probabilities of uniform and nonuniform mutation are set to relatively high values of $\rho_{\mathrm{m}}^{\mathrm{u}} = 3\,\%$ and $\rho_{\mathrm{m}}^{\mathrm{g}} = 7\,\%$, respectively. Choosing small population sizes in combination with comparably high mutation rates has also been motivated by previous investigations with a preceeding neural network ASIC prototype [93].

8.3.3 Fitness Estimation

The calculation of the fitness needs to be repeated for each new individual in every generation. Besides the genetic variation operators, the fitness estimation process therefore constitutes one of the most time critical aspects of an evolutionary algorithm. This particularly applies to the training of neural networks for the selected benchmark problems, since quantifying the performance of a network on one of these tasks involves the evaluation of its response to a potentially large number of input patterns (see table 8.1).

planned hardware implementation

On the other hand, common measures for the network performance reasonably treat all input instances — and the corresponding network outputs — in a uniform

[2] If $\mu - n_{\mathrm{el}}$ is an odd number, the remaining genome is not recombined but only mutated.

fashion (see equations 2.15 and 2.16). With regard to a fast implementation of the fitness calculation, this eventually motivates a pipelined and possibly even parallelized processing of the involved data within a configurable logic (see section 6.2.6). In the future, it is therefore planned to transfer the calculation of the fitness to the FPGAs on the Darkwing or NATHAN boards of the used hardware setup (see sections 6.1 and 6.3). For this approach to be feasible, the necessary computations for each single pattern need to be simple enough as to warrant their efficient realization with the available logic blocks of the used configurable substrate.

A Basic Approach

From what has been said above, it can be concluded that regardless of whether the fitness calculation is implemented in software or within an FPGA, it is preferably kept simple. Initially, this restriction seems to be easily reconciled with the chosen output representation. In response to an input pattern s^α of class C_k, the network is desired to activate the corresponding group of $N^{\mathrm{c}}_{\mathrm{out}} = 4$ output neurons and keep the remaining outputs deactivated. A simple bitwise comparison of all neuron outputs O^α_i, $1 \leq i \leq N_{\mathrm{c}} \cdot N^{\mathrm{c}}_{\mathrm{out}}$ with the corresponding target responses T^α_i yields the number of agreeing positions $\lambda(e^\alpha)$. After the individual scores for all processed inputs have been added up, the resulting total value $\Lambda = \sum_\alpha \lambda(e^\alpha)$ readily fulfills the general requirements to a suitable fitness function: First, it assumes its maximum value only for a network that always responds with the correct output. Second, due to the large numbers of instances and output neurons, Λ is expected to be a sufficiently smooth measure as to allow for a differentiated comparison between two competing individuals.

simple scoring scheme

repeated application of patterns

In practice, the continuity of the hereby defined fitness function is further improved by duplicating each of the training patterns $m_p > 1$ times and applying the total resulting total data set in a random order. Originally, this procedure is intended to account for the analog noise in the system: Due to the present temporal fluctuations, the repeated application of the same input can cause different network responses. Considering each pattern multiple times does not only yield a smoother fitness measure but also allows to quantify the stability of the respective candidate solution. Within the described set of experiments, m_p is set to 5.

A Refined Version

It is an outstanding feature of the proposed fitness calculation that it confines itself with simple bitwise comparisons and integer summations and can therefore be implemented efficiently either in software or within a configurable logic. Unfortunately, it turns out that in its original, simple form, the described procedure cannot serve as a suitable fitness estimation procedure in practice. The reasons for this are twofold.

Weighted Outputs A considerable fraction of the used classification benchmarks includes a number of classes N_{c} that is larger than two (see table 8.1). As

a consequence, the correct output $\mathbf{O}^\alpha$ in response to an applied input pattern s^α contains more zeros than ones. When the described fitness measure is employed for the evolutionary training of networks for such tasks, it can be observed that after only a few generations, the population has evolved to be comprised of networks that always respond with $O_i = 0$, $1 \leq i \leq N_{\text{out}}$, regardless of the presented input. This especially affects the glass, *E.coli*, and yeast problems (see table 8.1).

problem: unbalanced output

Although evidently undesired, this behavior can easily be understood. Consider the *E.coli* problem with its $N_{\text{c}} = 8$ classes. Each target output $\mathbf{T}^\alpha$ contains only $N_{\text{out}}^{\text{c}} = 4$ ones but $7 \cdot 4$ zeros. A network that permanently produces a response of zero with all its output neurons achieves a fitness that is 7/8 of the possible maximum. This breeds a critical evolutionary advantage over networks that exhibit a large number of false activations, even although they might actually constitute more promising candidate solutions in terms of the original classification task.

modified scoring scheme

For this reason, the fitness calculation is modified as follows: After a bitwise comparison with the target output $\mathbf{T}^\alpha$, the single outputs are weighted according to whether they are required to be activated or not. Every neuron that correctly shows an output of one is rewarded with a score of $N_{\text{c}} - 1$, those that correctly respond with zero contribute with a score of 1. As before, neurons that produce an incorrect output do not increase $\lambda(e^\alpha)$ at all. This scoring scheme ensures that the activation of the correct neurons and the deactivation of the remaining outputs are effectively weighted equally.

Finally, if no other class exhibits a larger number of activated output neurons than the correct one, the network is awarded an overall bonus of $2(N_{\text{c}} - 1)N_{\text{out}}^{\text{c}}$ which corresponds to the maximally achievable value of $\lambda(e^\alpha)$ without bonus. In total, the resulting single pattern scores thus range between 0 and $4(N_{\text{c}} - 1)N_{\text{out}}^{\text{c}}$.

Normalized Class Distributions In the majority of the investigated data sets, the different classes are not represented by equal numbers of instances (see appendix B). For example, the most common classes of the *E.coli* and yeast problems contribute 42.6 % and 31.2 % of the data, respectively. Even after incorporating the above modifications into the fitness function, the evolutionary algorithm turns out to produce networks that always predict the most frequently occurring class of the current problem, independently of the presented instance.

weighted class-specific scores

In analogy to the considerations of the preceeding paragraph, this problem can be overcome by an appropriate weighting of the different classes. The single pattern scores $\lambda(e^\alpha)$ are added up separately for each class C_k to obtain the class-specific fitness values Λ_k which are then divided by the respective square root of the number of instances N_{e}^{k} in each category. The sum of these normalized class scores forms the fitness of the individual.

final fitness function

In terms of the notations introduced above, the fitness function F that is used for the experiments presented in this chapter assumes the final explicit form:

$$F = \sum_{k=1}^{N_\mathrm{c}} \left(\frac{\Lambda_k}{\sqrt{N_\mathrm{e}^\mathrm{k}}} \right) = \sum_{k=1}^{N_\mathrm{c}} \left(\frac{1}{\sqrt{N_\mathrm{e}^\mathrm{k}}} \sum_{e^\alpha \in C_k} \lambda(e^\alpha) \right) \tag{8.2}$$

If desired, the resulting quantity can be normalized by $\left(\sum_k \sqrt{N_e^k} \right) 4(N_\mathrm{c} - 1) N_\mathrm{out}^\mathrm{c}$ to yield a fitness measure between 0 and 1.

preserved hardware suitability

Although this refined version does not retain the same simplicity as the original form, it is still suited for an implementation in a configurable logic. Apart from the normalization of the single class scores Λ_k by the square roots $\sqrt{N_\mathrm{e}^\mathrm{k}}$, the calculation persists to merely involve comparisons and summations. The divisions have to be performed only once per individual, and the required values $\sqrt{N_\mathrm{e}^\mathrm{k}}$ are fixed throughout the training process. They can be calculated in advance and, e.g., be stored in a lookup table. Since contemporary configurable logic substrates do not commonly incorporate floating-point units, the used weighting scheme might have to be modified to exclusively involve integer numbers, but the basic concept persists to be fit for realization within an FPGA.

8.4 First Training Experiments

It remains to be investigated whether the described evolutionary algorithm in combination with the predefined feedforward architecture can succeed in producing networks that achieve a satisfactory generalization performance on demanding real-world pattern categorization problems. To this end, a set of stratified cross-validation measurements (see section 8.1.2) is performed on each of the nine selected tasks.

8.4.1 Experimental Setup

chip parameters

All trained neural networks are implemented on a HAGEN chip and employ the architecture defined in section 8.2.4. The used ASICs are operated in calibrated mode such that both, the row-wise averages of the synapse offsets and the neuron offsets are compensated for (see section 5.5.1). The synaptic dynamic range is chosen to be limited to a maximum postsynaptic current of $\mathrm{I}_\mathrm{ps}^\mathrm{max} = 22\,\mu\mathrm{A}$ (see sections 5.4.3, 5.5.4, and 8.2.2).

hardware setup

The experiments utilize the hardware configuration described in section 6.1. The software part of the evolutionary algorithm is implemented within the HEAF framework of the HANNEE software (section 7.4) and is executed on a 2.4 GHz Pentium IV. The software communicates with a single HAGEN chip via one Darkwing board. Three identical setups are used, but all networks for one specific benchmark are always trained on the same chip. In all setups, the interface of the ASIC is operated at a frequency of 84 MHz (see section 6.1.3) which determines the the effective network frequency f_net (see section 5.2) to be 14 MHz [185] [181].

parameter	value
No. of partitionings N_p	10
networks per partitioning N_n	10
No. of repetitions N_r	5
output neurons per class $N_\mathrm{out}^\mathrm{c}$	4
hidden neurons per class $N_\mathrm{hid}^\mathrm{c}$	6
resolution of multi-bit inputs N_b	6(4)
population size μ	20
tournament size τ	2
recombination probability ρ_r in %	90
crossover probability ρ_c in %	90
uniform mutation rate $\rho_\mathrm{m}^\mathrm{u}$ in %	3
Gaussian mutation rate ρ_m^g in %	7
Gaussian mutation width in LSB	102
used crossover operator	two-point
elitist selection range n_el	1
maximum generation (per class)	1000
neuron offset calibration	on
row-wise offset calibration	on
maximum postsynaptic current $\mathrm{I}_\mathrm{ps}^\mathrm{max}$	22 μA
pattern multiplier during training m_p	5(2)
pattern multiplier for training set m_p^t	5(2)
pattern multiplier for test set m_p^g	50(20)

Table 8.3: *Summary of the relevant parameters of the employed experimental setup. The numbers in parenthesis represent the individual exceptions for two of the used benchmarks: The resolution of the used multi-bit integer inputs is reduced to 4 for the breast cancer problem (see section 8.2.2), and due to the sheer size of the yeast data, smaller pattern multipliers are used for this benchmark (see text). Apart from that, all tasks are treated equally.*

algorithm setup

The weights of all individuals in the initial population are set to uniformly distributed random values between -1023 and 1023. Since the size of the network depends on the number of classes N_c in the current task, the maximum allowed number of generations for the evolutionary training algorithm is adapted as well and is set to $1000 \cdot N_\mathrm{c}$. As noted in section 8.3.2, a chromosome-wise two-point crossover operator is employed for all experiments.

termination condition

If the fitness of the best individual in the current population reaches the highest possible value and if this condition is maintained over a period of 5 succeeding generations, the training is considered successful and the evolutionary algorithm is terminated immediately. The best individual in the last generation is taken as the result.

pattern repeat during evaluation

It has already been said that during training, each pattern is presented to the network $m_p = 5$ times in order to account for analog noise in the ASIC. The same scheme is adopted during the estimation of the final classification performance a_t

of the trained network. For the accurate evaluation of the generalization performance a_g, each pattern in the test set is even applied $m_p^g = 50$ times. This is intended to account for the observation that the network's performance on previously unseen data shows a higher sensitivity to noise than in the case of the training patterns. This is plausible, since for the latter, the increased robustness against temporal variations in the internal signals is automatically promoted by the employed chip-in-the-loop training approach. This does not equally apply to the generalization data. With the size of the test set being only 1/9 of the training data, the resulting numbers of patterns for the evaluation of the classification and generalization performance are approximately equal.

correct classification

When determining the categorization accuracy of the network on the training or generalization data, a given input pattern is only regarded as being classified correctly if no other class exhibits a larger or equal number of activated output neurons than the desired one. Ambiguous responses of the network where two or more classes show the same amount of activation are counted as false classifications.

Apart from that, the conducted experiments follow the setup described in the preceeding sections. The values of all relevant parameters are summarized in table 8.3.

8.4.2 Results and Discussion

comparison to previous results

The resulting averaged classification accuracies on the training and test sets A_t and A_g together with their respective standard errors of the mean are listed in table 8.4. It is well known that the selected tasks show a considerable variance in difficulty which manifests itself mainly in the form of varying maximum values for the achievable generalization rates. Therefore, the results for the individual tasks cannot directly be compared with each other but have to be judged in comparison to the performance that can realistically be achieved on the respective benchmark. The last column of table 8.4 presents some generalization rates that have previously been reported in literature. Note that only the results obtained by Prechelt [163] actually involve neural networks. The remaining investigations employ support vector machines [65], approximate distance classifiers [30], statistical methods [3], or k nearest neighbor classifiers [101].

differing evaluation procedures

It is to be emphasized that the values shown in this last column are primarily intended to convey a general feeling for the generalization rates that can in principle be achieved on the individual tasks. At this stage, a direct quantitative comparison with the presented experimental results is problematic since the cited investigations do not all employ the same stratified 10-fold cross-validation scheme. Some of the reported results are obtained on single random partitionings of the data [163], while others are the outcome of 4-fold [101], 5-fold [30], or leave-one-out [3] cross validation experiments. Although the provided value for the liver-disorder problem is indeed the result of a 10-fold measurement [65], the used subsets are not stratified. The same stratified 10-fold validation scheme that is also used for the experiments described in this work is only applied in the case of the yeast problem [101]. None of the publications reports error bounds for the

benchmark	No. of classes N_c	classification accuracy in %			
		training set A_t	test set A_g	reported A_g	taken from
breast cancer	2	98.26 ± 0.01	96.27 ± 0.13	98.85	[163]
diabetes	2	78.27 ± 0.02	73.17 ± 0.58	78.5	[163]
heart disease	2	89.06 ± 0.09	81.94 ± 0.48	96.8	[163]
liver disorder	2	75.94 ± 0.20	67.09 ± 0.78	73.7	[65]
iris plant	3	98.80 ± 0.05	94.30 ± 0.26	96–98	[30]
wine	3	91.10 ± 0.45	87.36 ± 1.37	100	[3]
glass	6	52.66 ± 0.68	46.07 ± 1.15	67.92	[163]
E.coli	8	57.13 ± 0.70	54.55 ± 0.45	86	[101]
yeast	10	8.52 ± 0.59	8.70 ± 0.68	60	[101]

Table 8.4: *Training results obtained with the described combination of network setup and evolutionary training algorithm. For problems with two or three classes, the results are promising. With an increasing number of classes, the gap between the best reported results and the accuracies that are obtained by networks on the HAGEN chip widens significantly. The yeast problem is not reasonably learned at all (see text).*

cited values. For a reasonable quantitative comparison with the measurements discussed in this thesis, the different validation schemes of the other investigations would have to be reproduced. A corresponding set of experiments will be discussed in section 9.4.5.

promising results

Against the background of these considerations, it can be inferred from table 8.4 that — for most of the used benchmarks — the results are promising. In the case of the breast cancer and iris plant problems, the obtained averaged accuracies on the test sets are approximately in the order of those reported by other authors. As expected, the performance on the test set is generally lower than the respective accuracy on the training data (except for the yeast problem, see below) but is high enough as to imply a satisfactory generalization ability of the trained networks. Nevertheless, the absolute values of the achieved accuracies clearly leave room for improvement: Apart from the liver disorder and iris plant data, not even the respective performance on the training set can exceed the generalization accuracy of other systems.

comparably low accuracies

remaining deficiencies

The gap between the achieved generalization performance and the accuracy that can approximately be expected in the ideal case turns out to be considerably wide for problems that incorporate larger numbers of classes. It is an important result that being trained on the glass and *E.coli* data sets, the final networks do not simply predict the most common class as it has been encountered in preliminary experiments (in the case of the glass problem, the largest class represents 35.5 % of the whole data). But compared to the results that have previously been reported by other authors, the performance of the trained networks is not satisfactory.

yeast problem: clear failure

In the case of the yeast problem, finally, the networks completely fail to produce any reasonable response at all. A closer investigation reveals that they persist to

yield an output which is largely independent of the actually applied input pattern, and that the single networks each tend to specialize on only one of the three most common classes (which contain 31.2 %, 28.9 %, and 16.4 % of the whole data, respectively). But even this specific class rarely exhibits a number of activated outputs that exceeds those of all others. Against the background of these observations and given the low absolute accuracies, it is not surprising that the averaged performance on the test data does not differ significantly from the classification rate on the training set.

8.4.3 Modified Training Setups

For categorization tasks with three, six, or up to eight classes, the normalization of the class scores that has been incorporated into the used fitness function (see section 8.3.3) seems to at least partially yield the desired improvements. Nevertheless, the eventually achieved generalization accuracies do not live up to the expectations and this particularly applies to tasks with more than three classes. The training of networks for the yeast problem evidently exceeds the capability of the used evolutionary algorithm setup. With respect to the initial goal of training networks on the HAGEN ASIC for demanding real-world applications — that are likely to involve large numbers of classes — it might thus be concluded that simple evolutionary algorithms and networks with a fixed architecture do not appear to be a feasible approach.

possible limitations

Before this conclusion can be drawn with certainty, it is to be ensured that the achieved training success is not in fact limited by the chosen particular evolutionary algorithm setup, i.e., the used mutation and crossover operators. In the light of the permutation problem, the mere application of crossover might itself impede a satisfactory outcome of the evolutionary training. Furthermore, since the experiments presented in this thesis are the first to consistently utilize the evolutionary coprocessor, it is to be verified that no hitherto unobserved deficiencies of this device are the reason for the achieved results.

modified evolution parameters

Therefore, a set of experiments similar to the ones discussed above is conducted where some of the parameters of the described setup are modified. In the first experiment, the Gaussian mutation is disabled completely and the probability for uniform mutation is raised to 4 %. The next three experiments use the original mutation probabilities but employ different crossover strategies: one-point crossover, no crossover, and chromosome exchange. In the latter case, the recombination operator simply swaps each pair of corresponding chromosomes between the two mating genomes with a modified probability of $\rho_c = 50\,\%$ (see sections 3.4.4 and 8.3.2).

varying crossover operators

pure software algorithm

Another experiment is performed that uses the unmodified evolution settings but utilizes a software implementation of the genetic coding and the variance operators instead of the coprocessor.

Finally, it is understood that the used network architecture, most notably the predetermined size of the respective hidden layer, can generally be suspected of constituting an unfavorable choice that eventually limits the achievable generalization performance. Indeed, it is to be expected that a systematic examination of multiple network sizes will yield an optimal number of hidden units per class

that is different from the chosen value $N_{\text{out}}^{\text{c}} = 6$ and depends on the particular task at hand. But although a corresponding set of experiments can easily be conducted, this would ultimately contradict the original intention of abandoning the optimization of the network architecture in favor of high training speeds. The sole existence of an optimal network size does not support the feasibility of the simple evolutionary approach, even if it resulted in an improved generalization capability of the trained networks. Moreover, the following chapter will present a training approach that does not require a thorough optimization of the network architectures in order to achieve competitive classification rates.

decoupled weight values

For these reasons, an exploration of different network sizes is deferred to section 10.3.1. The last experiment in this campaign rather investigates whether the enforced coupling of the weights to predefined multi-bit integer inputs (see section 8.2.2) might in fact prevent the training algorithm from discovering better solutions. In this experimental setup, all synaptic weights of the network are allowed to be optimized by the algorithm independently of each other. While this considerably increases the number of free parameters, it also permits to explore hitherto inaccessible regions of the search space.

8.4.4 Results Obtained with the Modified Setups

The results are listed in tables 8.5 and 8.6. Only the obtained generalization accuracies are shown, since the classification rates on the training sets are only of minor interest in practice. A complete summary of the results can be found in table C.1 in the appendix. Repeatedly training 500 networks for the *E.coli* and yeast problems is particularly time-consuming due to the large number of classes and — in case of the yeast benchmark — the sheer size of the data set. An investigation of single, exemplary evolution runs indicates that neither of the above modifications to the original setup can actually compensate for the general incapability of the used evolutionary approach to train networks for the yeast problem. Preliminary measurements with the *E.coli* data yield results that are similar to those obtained with the other tasks. Therefore, it is decided to exclude the last two benchmarks from this second set of experiments.

unchanged results

A comparison with table 8.4 immediately reveals that also in the cases of all other data sets, no significant improvement of the classification success can be achieved. For the breast cancer, liver disorder, and iris plant problems, the abandonment of the Gaussian mutation leads to a slight increase of the averaged generalization accuracy, but the respective difference to the original result is in no case larger than the estimated error. For all other benchmarks, the use of a Gaussian mutation has lead to a measurably better generalization.

crossover operators: equivalent

Considering the various forms of recombination, none of the tested alternatives seems to exhibit a distinct superiority over the others. Using no crossover at all only yields a better performance on the glass problem. For all other tasks, it either deteriorates the generalization rate or does not lead to an improvement that exceeds the estimated uncertainty. Similar conclusions are to be drawn for the one-point crossover and the chromosome exchange operator. Each of the investigated recombination methods shows favorable results on some of the benchmarks but

benchmark	generalization accuracy A_g in %		
	no Gaussian mutation	no crossover	one-point crossover
breast cancer	95.90 ± 0.19	96.18 ± 0.10	96.33 ± 0.18
diabetes	73.61 ± 0.60	73.33 ± 0.76	73.62 ± 0.36
heart disease	80.17 ± 0.85	80.76 ± 0.17	82.47 ± 0.63
liver disorder	67.71 ± 0.83	67.85 ± 0.83	68.21 ± 0.73
iris plant	93.51 ± 0.97	94.03 ± 0.46	94.87 ± 0.74
wine	83.61 ± 0.91	80.25 ± 1.60	85.03 ± 1.10
glass	43.25 ± 1.27	49.91 ± 0.47	46.36 ± 1.22

Table 8.5: *Training results obtained with slightly modified evolutionary algorithm setups. For the first experiment, the Gaussian mutation is disabled and the probability for uniform mutation is raised to 4%. The second and last columns refer to experiments without any crossover and with one-point crossover, respectively.*

benchmark	generalization accuracy A_g in %		
	chromosome exchange	software algorithm	no coupled weights
breast cancer	96.07 ± 0.09	96.40 ± 0.10	94.71 ± 0.15
diabetes	73.32 ± 0.67	72.68 ± 0.62	61.94 ± 0.65
heart disease	82.26 ± 0.53	81.49 ± 0.72	76.82 ± 0.32
liver disorder	67.15 ± 0.66	67.31 ± 0.74	51.83 ± 1.13
iris plant	94.86 ± 0.62	94.88 ± 0.35	58.06 ± 2.50
wine	81.83 ± 1.59	82.23 ± 0.90	46.33 ± 1.71
glass	46.30 ± 1.56	44.44 ± 1.37	25.00 ± 1.21

Table 8.6: *Training results obtained with slightly modified training resp. network setups. During the experiments that are represented by the first column, the applied recombination operator simply swaps each pair of corresponding chromosomes in the two mating individuals. The results in the second column are obtained with a pure software implementation of the genetic coding and variation operators. For the last investigation, the enforced coupling of the synaptic weights to form predefined multi-bit integer inputs is relieved and all weights can be adjusted independently of each other.*

yields inferior generalization rates on others. On most problems, the differences between the various crossover operators are within the estimated error bounds. The only exception is given by the wine benchmark where the initial two-point crossover is observed to lead to significantly better results than the alternative operators.

Especially with regard to the performance that is achieved without any recombination, these results can be seen as to support previous suggestions that the permutation problem might not in fact be as severe as widely supposed [80] [81] (see section 4.1.2). But they could also be accredited to the circumstance that in the presence of the used high mutation rates and small population sizes, the applied recombination operator might not have a significant impact on the optimization process at all.

It can be argued that especially for the larger networks, stronger selection in combination with even further increased mutation rates could potentially improve the performance of the evolutionary algorithm [93]. But similar to an exploration of the optimal architecture, the need to tune the evolution parameters to optimally suit the task at hand would rather militate in favor of an automatic adaptation of the free parameters during training than support the employed simple evolutionary approach.

coprocessor: functional

Beyond that, it is important to note that compared to the training performance of the software algorithm, the evolutionary coprocessor does not show any sign of a general insufficiency. While on the iris problem, the pure software algorithm yields a slightly better performance of the final networks, the utilization of the coprocessor leads to better results on the wine and glass benchmarks. Regarding all investigated data sets and within the remaining statistical uncertainties of the presented generalization rates, it can be said that both, the evolutionary coprocessor and the pure software implementation of the genetic coding and variation operators tend to perform approximately equally well. This is good news in so far as it supports the claim that the introduced coprocessor implementation allows to efficiently accelerate the evolutionary training algorithm without loss in training success.

weight coupling: necessary

Finally, it turns out that the original strategy to couple the weights of a multi-bit integer input to suit the used binary coding of the attribute values does indeed give rise to a remarkable improvement of the training performance compared to the case where all weights need to be adjusted independently. Using the latter approach, the achieved generalization accuracies are in some cases (e.g., the iris plant, wine and glass data sets) dramatically reduced compared to those obtained with all other investigated training setups. When appropriately coupling the weight values, the resulting reduction of the search space evidently yields such a significant simplification of the eventual optimization problem that it more than compensates for the reduced flexibility in adjusting the weight values.

8.4.5 Concluding Remarks

It is to be emphasized that regardless of the dissatisfying results that are obtained on benchmarks with more than three classes, the presented experiments success-

fully demonstrate the general functionality of the used hardware neural network framework, most notably the evolutionary coprocessor. The latter is demonstrated to achieve the same training performance as a slower software implementation. The generalization rates that are eventually obtained by comparably small networks for classification task with two or three classes are promising.

simple approach: insufficient

The observation that extensive modifications to the applied variation operators or their probabilities do not on average yield a significant change in the achieved results remains to indicate that the inherent robustness of the simple evolutionary approach might also give rise to a partial insensitivity against small changes of the evolution parameters, especially the used recombination operators. Initially, this can be seen as a favorable feature of the heuristic evolutionary approach, but it also indicates that the observed deficiency of the training setup is of a rather general nature: The used simple evolutionary algorithm is not fit for optimizing the weight values of large networks for demanding classification tasks with more than two or three classes.

This is also problematic in so far as even the largest of the trained networks — dedicated for the classification of yeast proteins — utilizes only a fraction of the resources offered by one HAGEN chip (see section 5.4.1). Aiming for an efficient exploitation of the parallelism that is provided by the implemented network model, it will ultimately be required to train even larger and more complex networks.

required: improved training

It is evident that more sophisticated evolutionary training strategies need to be devised that can readily benefit from the functionality of the coprocessor and at the same time succeed in training complex networks for realistic applications. A feasible approach will be introduced in the following chapter.

Chapter 9

Stepwise Evolutionary Training Strategies

Prediction is very difficult, especially about the future.

Niels Bohr

The experimental results presented in the foregoing chapter indicate that simple evolutionary algorithms do not suffice to successfully train large and complex neural networks on the HAGEN chip for challenging classification tasks. Regarding the sheer dimensionality of the resulting search spaces, the ubiquitous threat of premature convergence, and the various issues that are known to especially affect the evolutionary training of neural networks — like the permutation problem brought forward in section 4.1.2 — it is not surprising that the capability of simple evolutionary algorithms to train large neural networks is ultimately limited.

Common ways to face up to these challenges are to extend the basic evolutionary approach by incorporating sophisticated speciation schemes and/or elaborate genetic encodings. Several potential extensions to the described algorithm, such as an automatic adaptation of the evolution parameters, the optimization of the network architecture during training, or a further improved fitness function have already been suggested in the previous chapter. Numerous feasible extensions are also reported in literature and some of the more recent and most promising approaches to train neural networks by simulated evolution like the NEAT and EPNet algorithms have been discussed in section 4.2.2.

It has been motivated in section 7.4.5 that, in principle, such refined and improved algorithms can readily be implemented within the HEAF framework for evolutionary algorithms. Nevertheless, increasingly complex training algorithms necessarily introduce additional computational effort (see section 4.2.3). It persists that aiming for an efficient exploitation of the speed of the used network ASIC, an appropriate hardware neural network framework requires simple and fast training algorithms that can in the ideal case benefit from hardware acceleration and/or parallelization themselves (see sections 6.2.6 and 6.3.3).

In pursuit of a training strategy that can fulfill these demands, the following chapter introduces and evaluates a novel approach that aims to solve the com-

plex task of training large networks for difficult problems without introducing additional, time-consuming calculations. Instead, it follows an intuitive divide-and-conquer heuristic.

9.1 The Divide-and-Conquer Approach

When trying to solve a large and complex task by simple means, it is a common approach to divide the whole problem into several, easier to solve subproblems that can be dealt with independently. If the available tools suffice to cope with the smaller problems, the original task can easily be accomplished by completing the individual parts separately, i.e., either sequentially or in parallel. It remains that depending on the problem at hand, finding a reasonable partitioning into independent subproblems and/or determining an adequate combination of the obtained partial solutions to successfully solve the entire problem often turns out to be highly nontrivial.

9.1.1 Stepwise Network Training

dividing classification tasks

Regarding the training of neural networks for a pattern recognition task with N_c classes, it is comparably straight forward to split the original problem into N_c problems with two classes each: For every class C_k, one separate network is trained to distinguish its instances $e^\alpha \in C_k$ from those that belong to any other class $e^\alpha \in \mathbb{E} \setminus C_k$. This is illustrated schematically in figure 9.1 for an exemplary task with three classes. The shown networks are simplified to only exhibit $N_\mathrm{hid}^\mathrm{sn} = 4$ hidden neurons and $N_\mathrm{out}^\mathrm{sn} = 2$ outputs.

training stages and training phases

The first stage of the training is then divided into N_c phases such that in the kth phase, $1 \leq k \leq N_\mathrm{c}$, one dedicated network is trained for class C_k. In response to the binary representation s^α of an arbitrary instance of the problem, this network is demanded to activate all of its outputs if the presented instance belongs to class C_k and to produce an output of zero otherwise.

simple training of subnetworks

Two important observations can be made. First, the training of the single networks can be performed independently. In each case, the entire available training data can be used. Only the desired target outputs differ between the networks. Second, since they expect the same kind of input patterns, all networks can be connected to the same set of input nodes and can thus in fact be regarded as subnetworks of one large network. If every subnetwork can successfully be trained to fulfill its individual requirements, the whole network readily solves the original task.

uncompleted architecture

Compared to the architecture that has been chosen for the experiments in the preceeding chapter, the second layer of the final network does not exhibit the full connectivity. The outputs of one specific class C_k are not connected to the hidden neurons of the other subnetworks. Being trained to each distinguish a different class $l \neq k$ from the respective rest, the other subnetworks do in particular provide means of differentiating between instances of their own class l and the considered class k. In so far, the information that is coded by the inner neurons of those remaining subnetworks might be useful also for the output neurons of this class.

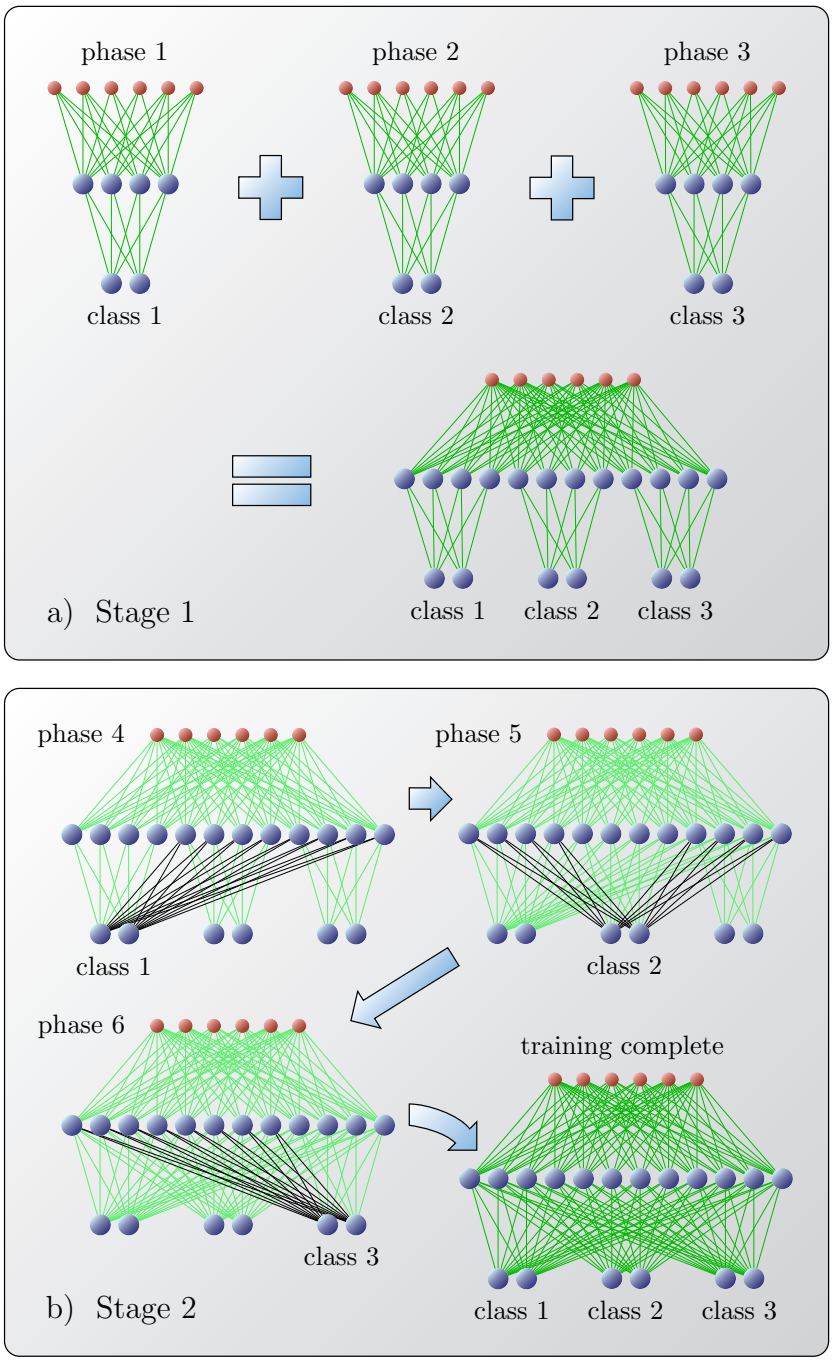


Figure 9.1: *During the first stage of the stepwise training procedure, one separate network is trained for each class (a). The shown example involves three classes and the numbers of hidden neurons and outputs per network are set to* $N_{\text{hid}}^{\text{sn}} = 4$ *and* $N_{\text{out}}^{\text{sn}} = 2$*, respectively. The individual networks can be regarded as subnetworks of one large network. During the second stage (b), the interconnections between these subnetworks are trained. Similar to the first stage, this can be done separately for each subnetwork such that in each phase, only the newly introduced connections (dark brown) are trained while the remaining synapses (light green) remain untouched. At the end of the second stage, the network is a fully connected two-layer perceptron.*

interconnecting the subnetworks

Therefore, it suggests itself to interconnect the single subnetworks in order to yield a homogeneously connected two-layer architecture. This is done in the second training stage shown in the lower half of figure 9.1. Similar to the first stage, the process can be divided into several independent phases: In each step, the outputs of one specific subnetwork k are connected to the hidden neurons of all others. While the synaptic weights that have been optimized in the preceeding phases remain fixed, the weight values of the new connections are optimized to improve the recognition accuracy of the respective class k.

independent training phases

As before, the single phases can be performed independently. During the training of the synapses that lead to the outputs of subnetwork k, the connections to all other outputs remain untouched. Furthermore, the iterative modifications of the optimized weight values are based solely on the response of the respective subnetwork — the outputs of the other subnetworks can readily be ignored. In this respect, the phases of the second stage are similar to those of the first stage which particularly allows to apply the same training algorithm.

9.1.2 Implications for Training

output encoding

Once the second stage has been completed, the final network exhibits the same connectivity and is trained to perform the same task as the network proposed in section 8.2. When being presented an instance of the respective problem, it is expected to activate the outputs that indicate the correct class and deactivate all others. In practice, the class with the largest number of activated neurons in response to an input pattern s^α is regarded as the prediction of the network.

smaller search spaces

simple subproblems

However, compared to the training process described in the foregoing chapter, the single phases of the stepwise approach involve optimization problems that are considerably simpler in several respects. First, in every phase, the amount of free parameters, i.e., weight values, is significantly reduced. Regarding the training results that can be achieved on small networks (see table 8.4) it is reasonable to assume that the used simple evolutionary algorithm can successfully cope with search spaces of the resulting dimensionalities. Second, the task that remains to be solved by each subnetwork is a mere two-class problem. Again, the results shown in table 8.4 indicate that the capability of simple training algorithms can be expected to suffice for this kind of task. Third, the desired network response demands all output neurons to behave equally. Against the background of what has been discussed in section 8.3.3, this allows for a considerable simplification of the fitness calculation. A weighting of the output neurons according to the respective target outputs becomes obsolete, and for each pattern s^α, the score $\lambda(e^\alpha)$ can simply be calculated as the number of output neurons that show the desired state of activation.

simpler fitness function

training times and parallelization

In summary, it is anticipated that the simple evolutionary algorithm described in section 8.3 can successfully be utilized for the single optimization problems that remain to be solved in each phase. On the other hand, for a problem with N_c classes, the whole training process now involves $2N_\mathrm{c}$ single evolution runs. Depending on the number of generations that are allowed per phase, the resulting total number of iterations might exceed that of the simple approach used in the

preceeding chapter. But compared to a simultaneous optimization of the whole network, the stepwise approach inherently promotes an immediate parallelization of the training process: Within each stage, the single phases can readily be performed in parallel. This particularly suits a hardware neural network platform that allows to evaluate multiple neural networks simultaneously. This aspect will be examined more closely in section 10.1.2.

9.1.3 Stepwise Training and Mixtures of Experts

The concept of dividing a given task into smaller subproblems that are to be dealt with independently by specialized modules has already been considered in the context of neural network training in another form. In 1991, Jacobs *et al.* proposed the so-called adaptive mixture of local experts approach [108]. It is an important difference between the introduced stepwise training strategy and the mixture of experts model that the latter performs a partitioning of the investigated task in the input space.

dividing the output space

In the case of the stepwise training procedure described above, the single networks are each specialized in recognizing a different class. Hence, it can be said that they are each specialized on a different dimension of the output space. At the same time, all subnetworks are trained on the complete set of training instances. In contrast, within the mixture of experts approach, the individual networks are trained to specialize on different regions of the input space. When a new vector $\mathbf{I}^{\alpha}$ is applied to the final system, it is not processed by all networks alike but only by the expert that is responsible for the respective region of the input space that contains $\mathbf{I}^{\alpha}$.

dividing the input space

learning the input partitioning

Since a suitable corresponding partitioning of the input space will in general not be known in advance, it needs to be determined during training. Apart from the single expert networks, the system proposed by Jacobs *et al.* therefore includes an additional, dedicated gating network that individually decides for each input pattern which of the experts is applicable. As a consequence, all expert networks and the gating network have to be trained simultaneously. During training, the single networks specialize on different areas of the input space, and the gating network iteratively learns how to best allocate the different instances to the single experts. A detailed account of the used training algorithm can be found in the cited publications and shall not be given here. But it is to be noted that this strategy and related approaches are reported to be successfully applicable to demanding vowel discrimination and visual recognition tasks [107] [108].

simultaneous training required

advantages of stepwise training

In the case of a classification task with N_{c} categories, the individual networks of the mixture of experts approach are in principle required to distinguish between all N_{c} classes — even if only for a reduced set of instances. According to the introduced stepwise strategy, each network merely has to solve a simple two-class problem and can be trained on all available input vectors. The necessary partitioning of the output space is automatically implied by the formulation of the actual classification problem and does not have to be learned. Besides allowing for the application of a simple algorithm to the training of the individual subnetworks, this eventually yields the advantage that all of the single training phases within

parameter	value
output neurons per subnetwork $N_{\text{out}}^{\text{c}}$	4
hidden neurons per subnetworks $N_{\text{hid}}^{\text{c}}$	6
uniform mutation rate $\rho_{\text{m}}^{\text{u}}$ in %	3(0)
Gaussian mutation rate $\rho_{\text{m}}^{\text{g}}$ in %	7(3)
Gaussian mutation width in LSB	100
maximum generation per phase	1000

Table 9.1: *Parameter modifications for the stepwise strategy. The two stages use different mutation probabilities, the values for the second stage are set in parenthesis. All remaining parameter values equal those listed in table 8.3.*

each of the two stages can be performed independently. This, in turn, permits a potential parallelization of the training process that cannot equally be achieved by the mixture of experts approach.

Nevertheless, it still awaits demonstration that the described stepwise training strategy can in fact yield an improvement in the generalization performance of the final networks compared to the simple evolutionary approach investigated in the preceeding chapter. This will be evaluated in the following sections.

9.2 Experiments with the Stepwise Strategy

network and training setup

For each of the nine benchmarks, a set of repeated stratified 10-fold cross-validation experiments (see section 8.1.2) is performed where the networks are trained using the stepwise strategy brought forward in section 9.1.1. The numbers of hidden neurons $N_{\text{hid}}^{\text{sn}}$ and outputs $N_{\text{out}}^{\text{sn}}$ for each subnetwork are chosen to be the same as the numbers of hidden neurons and outputs per class ($N_{\text{hid}}^{\text{c}}$ and $N_{\text{out}}^{\text{c}}$) that have been used for the experiments of the foregoing chapter. With the number of subnetworks being equal to the respective number of classes, this effectively yields networks of the same architecture and size as in the previous experiments. In each training phase, a maximum number of 1000 generations is allowed. Since every subnetwork is trained for a total of two phases — one in each stage — the resulting numbers of training iterations are twice as large as in the case of the initial experiments.

fitness function

According to what has been said above, the fitness function is modified such that in response to an applied input s^α, the single pattern score $\lambda(e^\alpha)$ is simply taken to be the number of agreeing positions between the network output $\mathbf{O}^\alpha$ and the desired target output $\mathbf{T}^\alpha$. Given these individual scores $\lambda(e^\alpha)$, the fitness F persists to be calculated according to equation 8.2, but the sum now only involves the two classes C_k and $\mathbb{E} \setminus C_k$. This evaluation procedure can readily be employed during both stages of the training.

evolutionary algorithm

For all phases of the first stage, the used mutation probabilities equal those listed in table 8.3. The new connections that are introduced in the second stage are not initialized randomly as it is done in the case of the first stage. Instead, they are set to starting values of zero. The training of these additional connections is performed

benchmark	No. of classes N_c	generalization accuracy A_g in %		
		complete training	only stage 1	one-layer perceptron
breast cancer	2	96.44 ± 0.23	95.34 ± 0.19	96.17 ± 0.14
diabetes	2	73.70 ± 0.31	66.92 ± 0.67	69.39 ± 0.29
heart disease	2	80.03 ± 0.54	79.82 ± 0.67	82.21 ± 0.57
liver disorder	2	66.51 ± 0.72	60.24 ± 0.63	65.40 ± 0.45
iris plant	3	95.10 ± 0.54	92.83 ± 0.44	88.46 ± 0.95
wine	3	95.31 ± 0.28	91.07 ± 0.36	94.07 ± 0.45
glass	6	62.56 ± 1.31	54.30 ± 0.60	60.89 ± 0.33
E.coli	8	81.01 ± 0.93	74.37 ± 0.23	79.74 ± 0.33
yeast	10	51.18 ± 0.31	42.55 ± 0.28	43.98 ± 0.41

Table 9.2: *Training results obtained with the stepwise strategy described in section 9.1.1. The first column shows the accuracies that are achieved with fully connected two-layer perceptrons. For the experiments shown in the middle column, the second training stage is omitted and the subnetworks thus remain unconnected. The values in the last column are obtained with single-layer perceptrons.*

without uniform mutation and the probability for the Gaussian operator is reduced to $\rho_m^g = 3\,\%$. Apart from that, the evolutionary algorithm described in section 8.3 remains unmodified. Table 9.1 summarizes all relevant parameters that have been modified compared to the previous experiments.

abbreviated training

In order to evaluate in how far the additional connections that are trained in the second training stage actually benefit the eventual generalization performance of the final networks, a further set of measurements is performed where the training is already terminated after the first stage. The single subnetworks thus remain unconnected.

alternative: single-layer networks

Finally, a last series of experiments employs a single-layer architecture for the subnetworks, where $N_{out}^c = 4$ output neurons per class are directly connected to the input nodes. Evidently, the stepwise training strategy can immediately be applied also to this topology. But since no hidden neurons need to be interconnected, no second training stage is performed. As a side effect, each of the resulting single-layer networks fits on one single network block of the HAGEN ASIC (see sections 5.4.1, 5.4.2, and 8.2.4).

9.2.1 Results and Discussion

The results of all described experiments are presented in table 9.2. Again, only the achieved generalization accuracies are shown, a complete overview that also includes the respective classification performance on the training set can be found in table C.2 in the appendix.

measurable improvements

It can be observed that the proposed stepwise training procedure mainly improves the generalization success on tasks with more than two classes. The generalization accuracies that are achieved on the iris plant, wine, glass, *E.coli* and

yeast data sets are measurably increased compared to networks that are trained as a whole (see table 8.4). For the iris plant problem, the improvement in generalization performance just exceeds the estimated error, but in case of the wine data set, the number of misclassification is reduced by more than a half.

benefits for larger networks

For large networks and problems with large numbers of classes, the stepwise approach naturally yields a substantial reduction of the search space in comparison to a simultaneous optimization of all weights. This is observed to lead to a drastic increase in generalization accuracy for the glass, *E.coli* and yeast problems. Compared to the results that are reported by other authors (see table 8.4) the obtained generalization rates now exhibit approximately the same slight inferiority as those achieved on problems with three or two classes. This is particularly satisfactory in the case of the yeast problem for which networks could not reasonably be trained with the simple evolutionary approach at all.

results for small networks

Given the observed measurement uncertainty, the results obtained on the breast cancer, diabetes, and liver disorder data sets are not significantly different to those reported in the preceeding chapter. Splitting the respective networks into two independently trained halves and thereby reducing the number of simultaneously optimized parameters does in these cases not yield a substantial improvement of the training. As suspected earlier, it can reasonably be concluded that for these benchmarks, the achieved generalization performance is not limited by the simplicity of the used evolutionary algorithm and/or the size of the networks. The only task where the stepwise strategy yields a measurably worse generalization performance than the naive approach of the previous chapter is the heart disease problem.

favorable: two training stages

It can furthermore be inferred from the third column of table 9.2 that the additional connections between the subnetworks which are introduced during the second stage of the training measurably benefit the performance of the final networks. An abandonment of the second stage leads to a decrease of the generalization accuracy for all tasks. With the exception of the heart disease problem, the observed deterioration always exceeds the estimated statistical uncertainty.

optimal architecture: task dependent

The last column, finally, reveals that the feasibility of a single-layer architecture strongly depends on the investigated task. The single-layer perceptrons that are trained for the heart disease problem actually seem to perform better on average than the networks with one hidden layer (see also table 8.4). Against the background of the above observation that even the simple evolutionary approach achieved better generalization on this benchmark, it can be concluded that the given number of 12 hidden neurons in combination with the extended training times of the stepwise strategy does in this case lead to an unfavorable overfitting of the data (see also figure 2.3 c)).

Comparing the second last and last columns, it becomes clear that when only one training stage is performed, networks with just one single layer tend to achieve better generalization accuracies than those with two layers. The only exceptions are the iris and wine problem.

suitability of single-layer networks

Although it can generally be observed that — apart from the heart disease problem — the generalization rates of the single-layer networks are lower than those of the two-layer perceptrons, it can also be seen that the respective difference turns

out to be remarkably small for most tasks. It is well known that two of the three classes that form the iris problem are definitely not linearly separable. This is consistent with the achieved results. Given the obtained accuracies on the other benchmarks, it is to be concluded that the respective classes actually contain a large fraction of linearly separable data. This seems to particularly apply to the heart disease problem where a linear partitioning of the input space is in fact superior to more complex class separations. This has already been observed during previous investigations by other authors [163].

The dimensionalities of the investigated problems inhibit a reasonable display in two or even three dimensions. But it is plausible to assume that when considering any pair of classes from one of these problem, their arrangement in the input space might be similar to the situation illustrated in figure 2.3 c).

Implications for Further Improvements

stepwise training: feasible

In summary, it can be concluded that the introduced stepwise training strategy tends to promote a measurable improvement in the achievable classification performance compared to the simple evolutionary approach that has been described in the foregoing chapter. In each phase of the training, a simple and fast evolutionary training algorithm can be used and the individual subnetworks for each class can in principle be trained in parallel. In so far, the stepwise training approach is particularly well suited for fast and massively parallel neural network hardware platforms (a possible way of exploiting this parallelism will be discussed in section 10.1.2).

remaining issues

Still, the achieved results can, as yet, not compete with those that are obtained by other systems, and the gained improvements primarily affect problems with more than two classes. Furthermore, considering the observations that are made on the heart disease benchmark, it must be stated that the restriction to a fixed and predefined, two-layer network architecture runs the risk of yielding suboptimal results on some problems.

unused resources

Finally, it is to be considered that a number of $N^{\mathrm{c}}_{\mathrm{hid}} = 6$ hidden neurons and/or $N^{\mathrm{c}}_{\mathrm{out}} = 4$ outputs per class, does — for most of the investigated benchmarks — by far not exploit the resources that are provided by even only a single HAGEN chip. Therefore, it would be desired to extend the proposed stepwise training procedure to allow for an efficient training of larger networks that can make best use of the offered hardware resources and achieve competitive generalization rates on a variety of problems without requiring an individual adaptation of the network architecture. The following sections will present a promising approach.

9.3 The Generalized Stepwise Strategy

In the spirit of an intuitive divide-and-conquer heuristic, the proposed stepwise training strategy performs a partitioning of the whole network that is inspired by the combinatorial structure of the actual categorization task: The resulting subnetworks are each trained for a different class. The experiments described in the preceeding section demonstrate that the resulting division of the training

process into the independent optimization of several single subnetworks is indeed feasible.

In pursuit of larger networks, the size of the individual subnetworks could be increased until the capability of the training algorithm to successfully optimize the resulting set of weights reaches its limit. But given the apparent success of the divide-and-conquer approach, it also suggests itself to compensate for increased network sizes by further dividing the thus enlarged subnetworks themselves.

9.3.1 Training Multiple Networks per Class

generalizing the stepwise strategy

The original stepwise strategy shown in figure 9.1 can easily be extended by training multiple networks for each class instead of one. Figure 9.2 schematically illustrates a simple example where a number of $N_{\text{net}}^{\text{c}} = 2$ subnetworks is trained for each of the $N_{\text{c}} = 2$ possible classes. Effectively, each of the N_{c} phases of the first stage is simply repeated $N_{\text{net}}^{\text{c}}$ times. Again, the resulting set of $N_{\text{c}} \cdot N_{\text{net}}^{\text{c}}$ networks can readily be regarded as to form one large network.

effect: increased networks

The second training stage then proceeds as before: During a number of $N_{\text{c}} \cdot N_{\text{net}}^{\text{c}}$ training phases, the outputs of each subnetwork are connected to the hidden neurons of all other subnetworks. Once again, the resulting network turns out to exhibit a fully connected two-layer architecture, the only change being an increased number of hidden neurons $N_{\text{net}}^{\text{c}} \cdot N_{\text{hid}}^{\text{c}}$ and outputs $N_{\text{net}}^{\text{c}} \cdot N_{\text{out}}^{\text{c}}$.

Multiple Subnetworks vs. Increased Subnetwork Size

Initially, generalizing the original strategy to the training of multiple subnetworks per class does not automatically seem to yield an advantage over a simple increase of the subnetwork size. As long as the used training algorithm can cope with the resulting dimensionality of the search space, a further division of the enlarged subnetwork is not necessarily expected to increase the training performance. At least, this is suggested by the results that are obtained with the original stepwise approach on two-class problems (see table 9.2).

increased training times

Nevertheless, the proposed extended strategy promises to be advantageous in several respects. Both, the training of larger subnetworks and the optimization of multiple subnetworks per class eventually increase the required number of training iterations. In the latter case, the total number of performed generations of the evolutionary algorithm linearly scales with the number of subnetworks per class — if it is assumed that the maximum number of allowed generations per phase remains unchanged. Given a fixed size of the individual subnetworks, the overall number of iterations is then proportional to the total number of hidden neurons in the final network. For an alternative training of larger subnetworks, the number of training iterations in each phase is reasonably increased in approximately the same fashion.

potential parallelization

Having said that, it remains an important difference between the two approaches that the training of multiple networks per class can readily be performed in parallel. Evidently, this does not apply to the additional generations that will have to be processed for the training of the larger networks. The parallelization of the

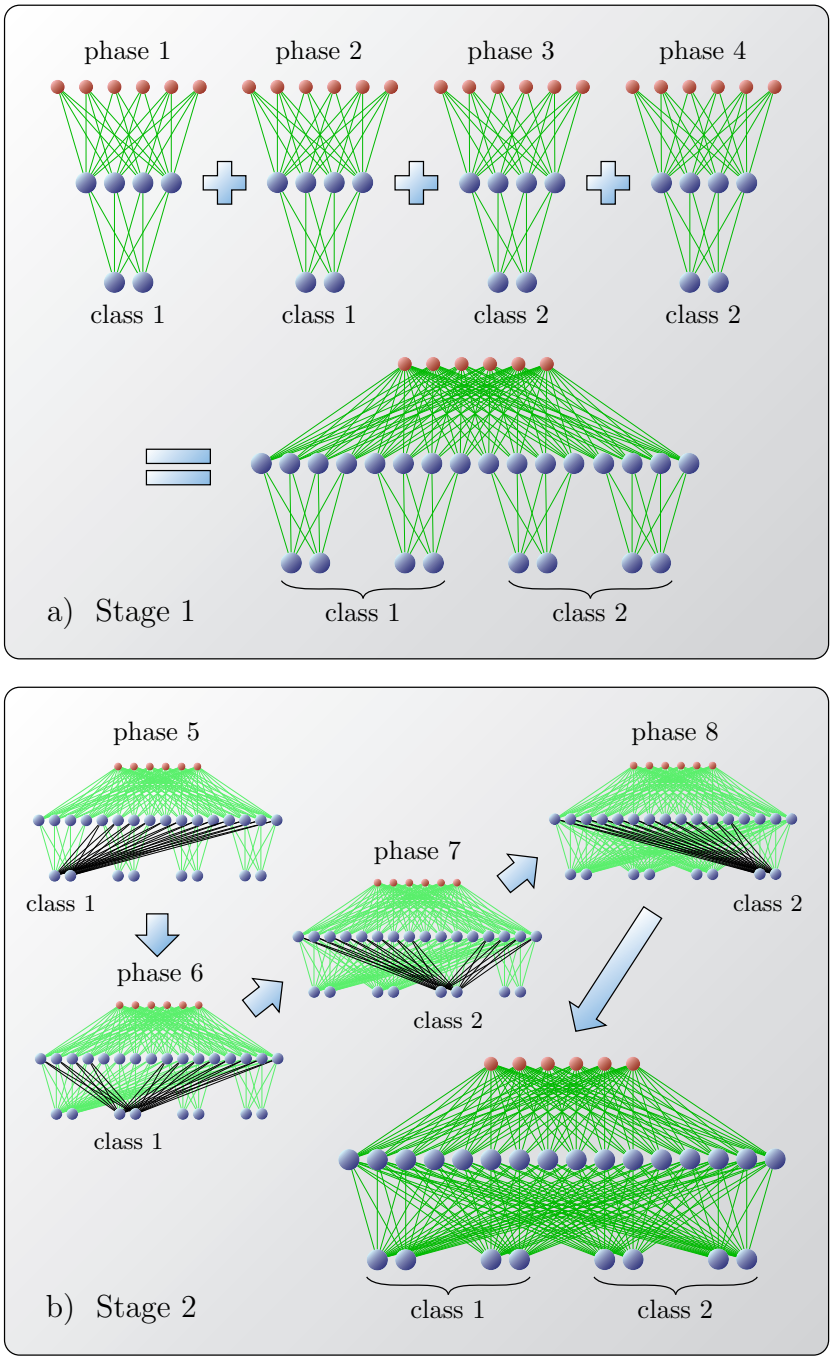


Figure 9.2: *The stepwise training strategy can easily be generalized to a training of multiple subnetworks per class instead of one. Here, an exemplary two-class problem is considered and* $N_{\text{net}}^{\text{c}} = 2$ *different subnetworks are trained for each category. The first training stage (a) then includes* $2 \cdot 2$ *phases and the resulting network features* $2 \cdot 2$ *subnetworks, two of which are sensitive to the same class, respectively. As before, the single subnetworks are interconnected during the second training stage (b). The final network is a fully connected two-layer perceptron with an effective number of* $2 \cdot N_{\text{hid}}^{\text{sn}} = 8$ *hidden neurons and* $2 \cdot N_{\text{out}}^{\text{sn}} = 4$ *output neurons for each class.*

stepwise strategy will more thoroughly be discussed in section 10.1.2.

feasibility of simple algorithms

Beyond that, increasing the size of the single subnetworks will eventually be limited by the capabilities of the used simple training algorithm. Indeed, comparing the results of the previous chapter with the performance of the stepwise strategy (tables 8.4 and 9.2), it can be expected that a measurable deterioration of the training success already occurs for networks with $3 \cdot 6$ hidden neurons (e.g., the networks for the iris and wine problems). In contrast, repeatedly training and interconnecting multiple networks of a fixed size is limited only by the available network resources, the achievable degree of parallelization, and the allowed training time.

9.3.2 Network Ensembles: Theoretical Considerations

subnetwork diversity

In addition to the above considerations, training multiple subnetworks per class instead of a single large one is also anticipated to be beneficial with respect to the achievable generalization accuracy. It has been discussed earlier that the eventual outcome of an evolutionary training run is influenced by numerous random decisions, and the multiple subnetworks i, $1 \leq i \leq N_{\text{net}}^{\text{c}}$ that are trained for a given class C_k are therefore expected to be slightly different. Due to their individual partitionings of the input space, the number of outputs $\nu_k^i(s^\alpha) \in \{1, \ldots, N_{\text{out}}^{\text{sn}}\}$ that are activated in response to the application of the same input pattern s^α is likely to vary between the single networks — at least for some of the instances (compare figure 2.3 c)).

averaged outputs

Following the proposed strategy, the binary output patterns of these subnetworks are simply concatenated such that the total number of activated output neurons per class immediately becomes

$$\nu_k(s^\alpha) = \sum_{i=1}^{N_{\text{net}}^{\text{c}}} \nu_k^i(s^\alpha) \qquad k \in \{1, \ldots, N_{\text{c}}\} \tag{9.1}$$

and eventually obeys $\nu_k \in \{1, \ldots, N_{\text{out}}^{\text{sn}} \cdot N_{\text{net}}^{\text{c}}\}$, $1 \leq k \leq N_{\text{c}}$. As long as the same number of subnetworks $N_{\text{net}}^{\text{c}}$ is trained for each category, this procedure effectively yields the same response of the entire network as when the averaged numbers of outputs $\overline{\nu_k} = \nu_k / N_{\text{net}}^{\text{c}}$ for each class were compared.

network ensembles

So-called ensembles of networks whose outputs are appropriately combined to yield a collective prediction of the whole committee have already been examined repeatedly in literature [105] [111] [124] [154] [153] [160] [189] [190].

function approximation

Perrone and Cooper [160] consider a group of N networks with the respective output functions $y_i(\mathbf{I})$, $1 \leq i \leq N$ that each approximate a given target function $f(\mathbf{I})$ with a specific individual error $\epsilon_i(\mathbf{I})$, $f(\mathbf{I}) = y_i(\mathbf{I}) + \epsilon_i(\mathbf{I})$. The expected sum-of-squares error of network $y_i(\mathbf{I})$ can then be written as

$$E_i = \langle\, (y_i(\mathbf{I}) - f(\mathbf{I}))^2 \,\rangle = \langle\, \epsilon_i^2 \,\rangle \tag{9.2}$$

(compare equations 2.15 and 2.16) and the on average expected error of any single

network in the committee simply becomes

$$\overline{E} = \frac{1}{N} \sum_{i=1}^{N} \langle\, \epsilon_i^2 \,\rangle \tag{9.3}$$

where $\langle \cdot \rangle$ denotes the expectation value on the input space $\mathbf{I} \in \mathbb{I}$.

ensemble prediction

Instead of considering only the output of a single network, one can regard the collective prediction of the whole ensemble $\overline{y}(\mathbf{I})$ to be the average over all individual network responses

$$\overline{y}(\mathbf{I}) = \frac{1}{N} \sum_{i=1}^{N} y_i(\mathbf{I}). \tag{9.4}$$

The expected sum-of-squares error of this ensemble output is then given by

$$E_{\text{ens}} = \left\langle \left(\frac{1}{N} \sum_{i=1}^{N} y_i(\mathbf{I}) - f(\mathbf{I}) \right)^2 \right\rangle = \left\langle \left(\frac{1}{N} \sum_{i=1}^{N} \epsilon_i \right)^2 \right\rangle. \tag{9.5}$$

improved accuracy

Perrone and Cooper derive the important result that if the errors $\epsilon_i(\mathbf{I})$ of the different networks are assumed to be uncorrelated and to have zero mean, then the estimated error of the ensemble E_{ens} and the averaged estimated error of the single networks $\overline{E}$ obey

$$E_{\text{ens}} = \frac{1}{N}\, \overline{E}. \tag{9.6}$$

limitations

At first sight, this is a remarkable observation in so far as it suggests that the sum-of-squares error of the ensemble prediction can be reduced below any desired threshold $\epsilon > 0$ simply by regarding committees of sufficient size. In practice, however, the assumption that the errors $\epsilon_i(\mathbf{I})$ of the single networks are uncorrelated does not hold. In fact, the errors are anticipated to be highly correlated, since they are strongly influenced by the used network architecture, the employed training algorithm, and the structure of the available training data. In general, when networks with equal architectures are trained on the same data in a more or less deterministic fashion (see sections 2.1.3 and 2.2.2), the remaining errors of the individual networks are likely to occur on approximately the same subset of problematic input values.

guaranteed: preserved accuracy

Therefore, equation 9.6 can merely serve as a lower bound on the actually observed ensemble error E_{ens}. On the other hand, it can also be shown quite easily [20] that the use of an ensemble can never increase the on average expected error, i.e.,

$$E_{\text{ens}} \leq \overline{E}. \tag{9.7}$$

Experimental results suggest that some measurable improvement in performance will generally be observed [20] [160].

Network Ensembles and Diversity

ensemble diversity

It remains that in order for an ensemble to be superior over a single network, the members of the committee need to exhibit a sufficient diversity, i.e., they must not behave equally on all input patterns. Even an arbitrary number of identical networks will not yield any improvement over a single instance of the same network. This intuitively plausible insight has been formalized by Krogh and Vedelsby [124]. For each network i, they consider the deviation $d_i(\mathbf{I})$ of its output from the ensemble mean

$$d_i(\mathbf{I}) = (y_i(\mathbf{I}) - \overline{y}(\mathbf{I}))^2 \tag{9.8}$$

and define the on average expected ensemble diversity (or "ambiguity") to be

$$D_{\text{ens}} = \frac{1}{N} \sum_{i=1}^{N} \langle\, d_i(\mathbf{I}) \,\rangle. \tag{9.9}$$

diversity and accuracy

On the basis of this definition, a simple relation between the estimated ensemble error E_{ens}, the expected error of the single networks $\overline{E}$, and the ensemble diversity D_{ens} is obtained:

$$E_{\text{ens}} = \overline{E} - D_{\text{ens}} \tag{9.10}$$

Again, it turns out that the error of the ensemble is guaranteed to be not larger than the average error of the single networks. The achieved reduction is determined by the diversity D_{ens} of the networks within the committee.

promoting diversity

Aiming for the construction of an efficient ensemble, it is thus necessary to not only optimize the accuracy, but also the diversity of the contained networks. Various approaches to promote a sufficient disagreement between the different committee members are conceivable and have been investigated in literature. Prominent strategies are to vary the used training data between the individual networks [124] [153], to employ different architectures, start from different initializations of the weights [130], or to apply varying training algorithms (for an overview see [189]).

actively enforcing diversity

More recent approaches aim to actively minimize the similarity between the individual ensemble members. For example, Opitz and Shavlik use a genetic algorithm to construct a diverse set of networks [154]. Other procedures start with a minimal ensemble of only one or two networks and successively add and train new members in a way that ensures these newly introduced networks to be different from the others [73] [105]. The algorithm proposed by Islam *et al.* [105] additionally incorporates the automatic adjustment of the architectures and the number of networks in the ensemble. While this last approach promises to maximize the diversity within the committee most efficiently and yields excellent results on various benchmark problems (see section 9.4.5), it also requires elaborate procedures to ensure the diversity of the networks. In general, for all of the above approaches, the training of each individual network needs to incorporate knowledge of the current performance of all others, i.e., the networks can no longer be trained independently.

The Stepwise Strategy and Diversity

threshold neurons and classification

It is understood that equations 9.6, 9.7, and 9.10 have been derived for networks with continuous outputs that are being applied to function approximation tasks and can as such not directly be transferred to networks on the HAGEN chip that are trained to solve classification problems. Still, when using a number $N_{\text{out}}^{\text{c}}$ of output neurons per class, the task to activate them all whenever an applied input pattern belongs to a given class and to deactivate them otherwise can be seen as the approximation of a function that only assumes values of either 0 or $N_{\text{out}}^{\text{c}}$. The difference between the number of actually activated outputs and the respectively desired value can thus serve as an — admittedly coarse — error measure similar to the quantity $\epsilon_i(\mathbf{I})$ considered above.

qualitative similarity

As it has been argued before, concatenating the outputs of several networks that are trained to recognize the same class is effectively equivalent to an averaging over the single network responses. In so far, the propositions of the preceeding sections are assumed to at least qualitatively hold also for the introduced generalized stepwise strategy.

It remains that the final response of the whole network is obtained by a majority voting among the different classes. This ultimately impedes an equally straight forward theoretical investigation of the situation as in the case of the originally considered function approximation tasks. A thorough discussion of these topics lies beyond the scope of this thesis. But it is reasonable to expect that an improved performance of the single parts which each specialize on a different class C_k will also lead to an increased classification accuracy of the whole network. Training multiple subnetworks for the same category is therefore anticipated to yield better generalization results — as long as it can be ensured that the different subnetworks exhibit a sufficient diversity (see equation 9.10).

improved accuracy expected

diversity through independent evolution

In the case of the proposed stepwise strategy, the diversity of the multiple subnetworks that are trained for the same class is expected to automatically arise from the intrinsic random nature of the evolutionary training algorithm. Initially, it is not at all clear whether this suffices to yield a satisfactory reduction of the similarity between the individual network responses. But in contrast to the previously mentioned strategies that actively enforce the diversity within the constructed ensembles, the introduced stepwise approach allows for a truly independent training of the single subnetworks. This even holds for the second training stage that occurs for two-layer networks: When optimizing the interconnections that lead to one specific subnetwork, only its own output needs to be evaluated. The modifications that are applied to the corresponding weights do in no way affect the other subnetworks. As stated before, this permits the training procedure to be remarkably simple and allows for a degree of parallelization that cannot be achieved by the more elaborate procedures discussed above. The only remaining restriction is that all phases of the first stage need to be completed before the second stage can be initiated.

information sharing via interconnections

Furthermore, it is to be considered that the additional interconnections that are established in the second training stage allow the single subnetworks to share useful information. In addition to the benefits that automatically arise due to

the use of an ensemble of separate networks, this might even permit a more efficient exploitation of the thus increased network resources. Still, it remains to be verified whether the introduced generalized stepwise strategy can ultimately yield a measurable gain in classification accuracy or whether it can compete with a simple enlargement of the single subnetworks. These questions are addressed in the following sections.

9.4 Experiments with the Extended Stepwise Strategy

fixed subnetwork size

An extensive campaign of measurements is performed to evaluate in how far the classification performance of networks on the HAGEN chip can be increased by training multiple subnetworks $N^{\mathrm{c}}_{\mathrm{net}}$ per class. For simplicity, the size of each subnetwork remains fixed to $N^{\mathrm{c}}_{\mathrm{hid}} = 6$ and $N^{\mathrm{c}}_{\mathrm{out}} = 4$, and the number of allowed generations per training phase is consistently set to 1000 generations (see table 9.1).

varied number of subnetworks

Similar to the preceeding investigations, a repeated stratified 10-fold cross-validation measurement is conducted for each benchmark and for several different numbers $N^{\mathrm{c}}_{\mathrm{net}}$ of subnetworks per class. Besides the described two-layer architecture, this is also done for networks with only one layer where the second training stage is naturally omitted.

unmodified training

In each phase, the training is performed by the evolutionary algorithm that has also been employed for the experiments of the foregoing section, and the used parameters equal those listed in tables 9.1 and 8.3. Apart from a different initialization of the adjustable weight values and the modified mutation probabilities, the training algorithms of the two successive stages that are used for the two-layer architectures are essentially equal (see section 9.2). In consideration of the results shown in table 9.2, both training stages are completed and the final networks are fully-connected two-layer perceptrons.

9.4.1 General Observations

As an introductory example, the results obtained on the liver disorder benchmark are presented in figure 9.3. The diagrams show the averaged classification rates on the training and test data A_{t} and A_{g} as a function of the number of subnetworks per class $N^{\mathrm{c}}_{\mathrm{net}}$. The upper part refers to two-layer networks while the lower part presents the accuracies of the single-layer perceptrons (see also figure C.1 in the appendix). It shall be repeated that each pair of data points involves the training of 500 networks.

liver disorder

For the achieved performance of both, the single-layer and two-layer networks, a clear dependence on the number of subnetworks per class can be observed: With increasing $N^{\mathrm{c}}_{\mathrm{net}}$, the classification accuracies A_{t} and A_{g} measurably improve. The gain in performance is distinct for $N^{\mathrm{c}}_{\mathrm{net}} = 2$ and 3 but is seen to abate for larger numbers of subnetworks. Around approximately $N^{\mathrm{c}}_{\mathrm{net}} = 5$, any potential further increase in the generalization rate begins to founder in the remaining statistical fluctuations. This behavior is commonly encountered for most of the benchmarks.

wine

Figure 9.4 shows the outcome of the corresponding set of experiments with the

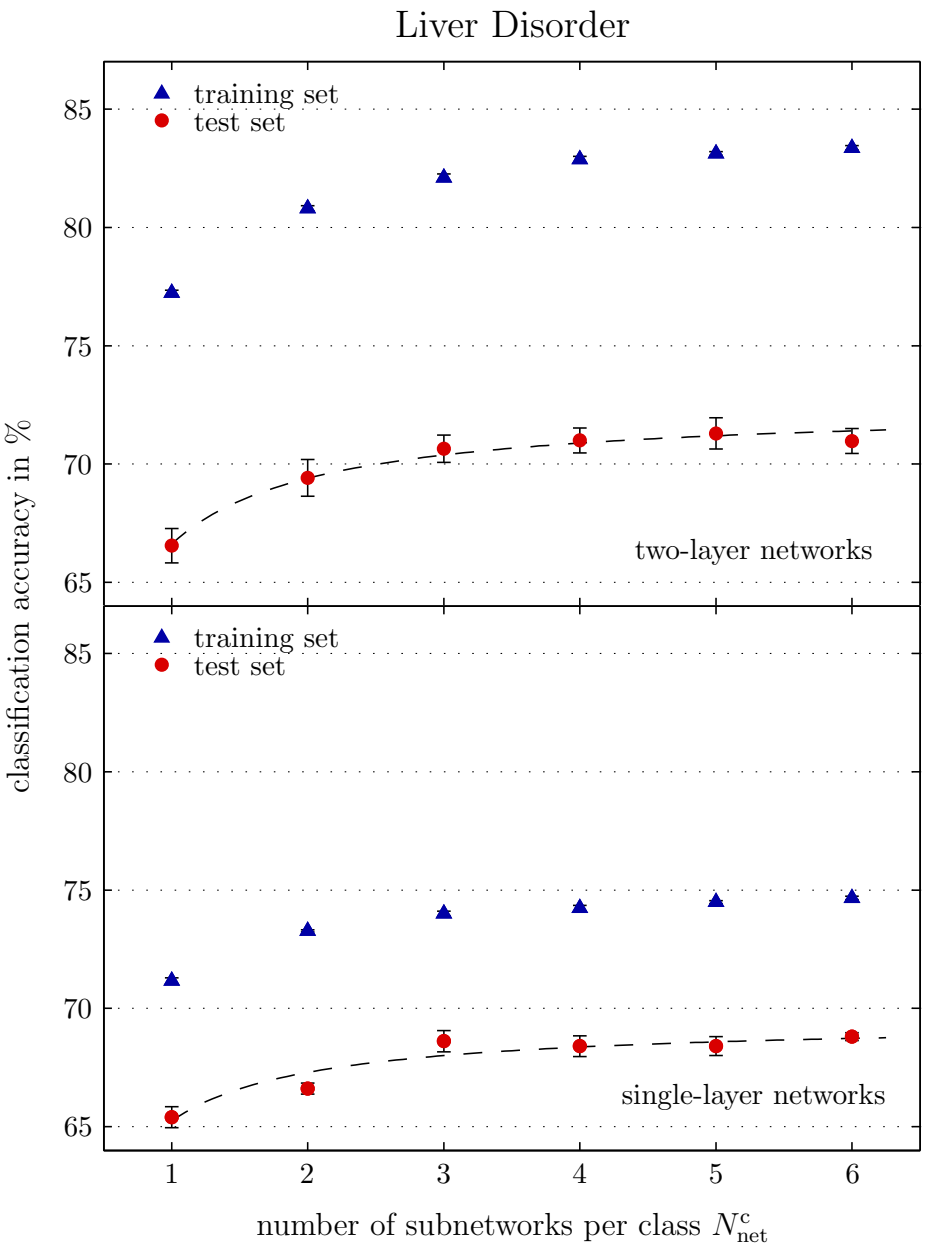


Figure 9.3: *The achieved classification accuracies on the training sets A_t and test sets A_g as a function of the number N^{c}_{net} of subnetworks per class for the liver disorder problem. The upper diagram presents the results of the two-layer networks, the lower diagram refers to single-layer perceptrons. Each pair of data points is the result of a repeated stratified 10-fold cross-validation. The results for $N^{c}_{net} = 1$ correspond to those listed in table 9.2. The shown dashed lines are the result of a heuristic fit (see equation 9.11) and are merely intended to serve as a guide to the eye (see text). For the classification accuracies on the training set, the error bars are very small and are partly covered by the markers.*

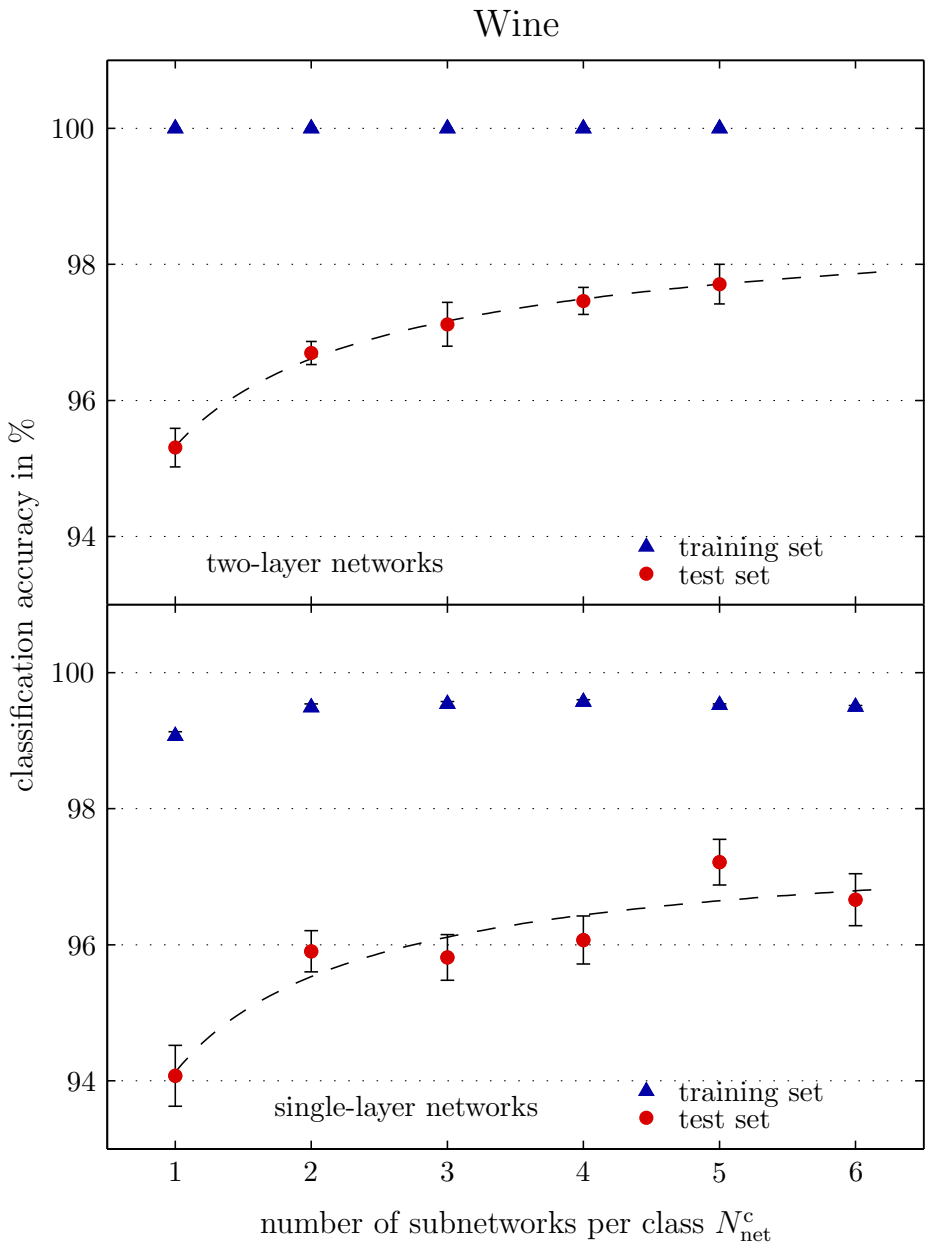


Figure 9.4: *The achieved classification accuracies on the training sets A_t and test sets A_g as a function of the number N_{net}^{c} of subnetworks per class for the wine problem. The upper diagram presents the results of the two-layer networks, the lower diagram refers to single-layer perceptrons. Each pair of data points is the result of a repeated stratified 10-fold cross-validation. The results for $N_{net}^{c} = 1$ correspond to those listed in table 9.2. The shown dashed lines are the result of a heuristic fit (see equation 9.11) and are merely intended to serve as a guide to the eye (see text). Given the three classes of the wine problem and the fixed number N_{hid}^{c} of hidden neurons per subnetwork, a maximum of 5 subnetworks per class can be realized when a two-layer architecture is employed. For the classification accuracies on the training set, the error bars are very small and are partly covered by the markers.*

wine data set (see also figure C.2 in the appendix). Already for one subnetwork per class, the classification accuracy of the two-layer perceptrons on the training data is 100 % and is thus not increased any further. Apart from that, the results are comparable to those obtained with the liver disorder problem. The increase in generalization performance for the two-layer perceptrons is remarkable in so far, as the number of misclassifications is reduced by nearly a factor of two. Although the results of the single-layer networks exhibit a certain noise, the general characteristics of the curve are anticipated to correspond to those observed during the liver disorder measurements.

measurable improvement

These results confirm the theoretically motivated expectation that the use of multiple subnetworks per class can considerably improve the performance of the trained networks. In the case of the single-layer perceptrons, the individual subnetworks remain unconnected and the achieved gain in classification accuracy solely arises from the usage of a diverse ensemble of independent networks (compare equation 9.10).

simple fit

With increasing $N^{\mathrm{c}}_{\mathrm{net}}$, the observed improvement in generalization accuracy even seems to show a qualitative similarity to the reduction of the ensemble error which is predicted for networks with continuous outputs that are applied to function approximation tasks (see section 9.3.2). In coarse analogy to equation 9.6, a simple mapping of the form

$$E(N^{\mathrm{c}}_{\mathrm{net}}) = \frac{E_{\max} - E_{\min}}{N^{\mathrm{c}}_{\mathrm{net}}} + E_{\min} \tag{9.11}$$

is fitted to the squared averaged classification errors $E_{\mathrm{g}} = (100 - A_{\mathrm{g}})^2$ obtained on the test sets. The curve fit is performed by a least squares fit routine included in the MATLAB software package [134], and the adjustable variables are the two parameters $E_{\max}$ and $E_{\min}$. The fit results are used to calculate corresponding values for the generalization accuracy $A_{\mathrm{g}}(N^{\mathrm{c}}_{\mathrm{net}}) = 100 - \sqrt{E_{\mathrm{g}}(N^{\mathrm{c}}_{\mathrm{net}})}$ that are included in figures 9.3 and 9.4 as dashed lines.

guide to the eye

It has been discussed in section 9.3.2 that the propositions of equation 9.6 cannot directly be transferred to binary networks on the HAGEN chip that are used to perform pattern classifications by majority voting. In so far, it shall be emphasized that equation 9.11 is not claimed to be justified by sound theoretical considerations. Rather, it represents a heuristically chosen regression function that is primarily intended to serve as an appropriate guide to the eye. It remains an interesting observation that the simple functional mapping given by equation 9.11 can describe the experimental results obtained with both, two-layer and single-layer networks remarkably well (compare also the following figures). However, a thorough theoretical investigation of this topic exceeds the scope of this thesis.

9.4.2 Hardware Limitations and Single-Layer Networks

limited number of hidden neurons

Given the three classes of the wine problem and considering networks with one hidden layer, a maximum of $N^{\mathrm{c}}_{\mathrm{net}} = 5$ subnetworks per class can be implemented (see figure 9.4). This is due to the fact that the available inter-block connections on the HAGEN ASIC allow a maximum number of hidden neurons of 96 (see

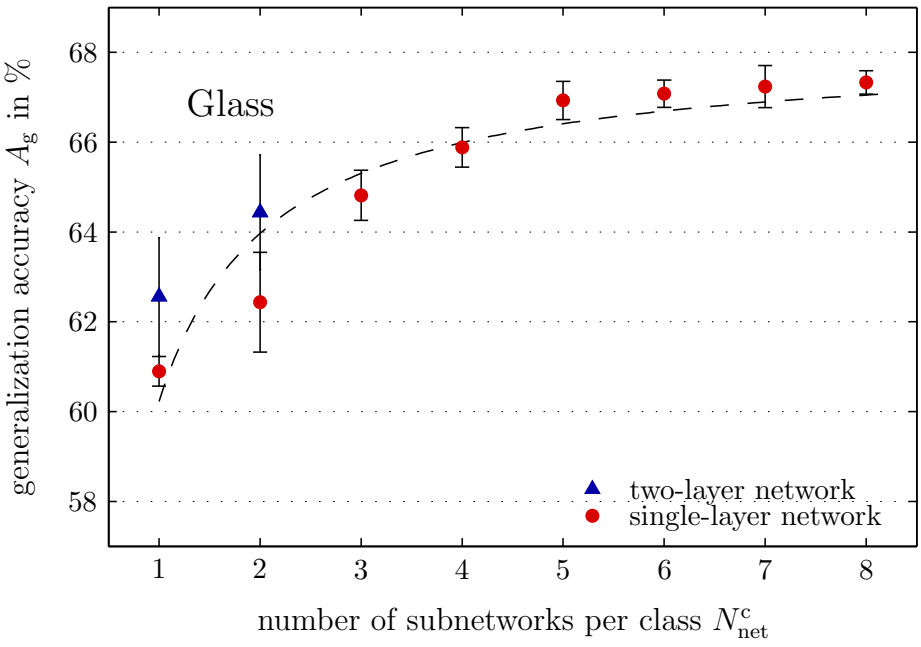


Figure 9.5: *The achieved classification accuracies on the test sets A_g as a function of the number N^c_{net} of subnetworks per class for the glass problem. Due to the six classes of this benchmark and given the fixed subnetwork size of $N^c_{hid} = 6$ hidden neurons, a maximum of two subnetworks per class can be realized for the two-layer networks. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1).*

sections 5.4.1 and 5.4.2). The lower right block, for example, can receive up to 64 inputs from its counterpart on the left side and 32 signals from the left block in the upper half. As it has been discussed in section 5.5.4, one of the involved inputs is omitted to avoid edge effects and two are reserved for the neuron offset calibration. This effectively limits the size of the hidden layer to a maximum number of 93 hidden neurons.

simulated inter-block connections

A more flexible interconnectivity between the network blocks can in principle be simulated by reading back the outputs of all blocks after each network cycle and appropriately processing this data within the FPGA to construct the new inputs for the following cycle (see sections 5.2 and 6.1). A corresponding functionality is implemented [28]. But it is evident that such a procedure would severely slow down the network operation. Apart from that, the maximum number of subnetworks per class would still be limited by the fact that for a full connectivity of the second layer, the latter has to fit on one single network block. Restricting oneself to a homogeneously connected, two-layer architecture, the number of output neurons of the whole network is thus ultimately limited to 64.

limited number of outputs

limited number of subnetworks

Depending on the number of classes in the investigated task and given the agreed number of hidden/output nodes per subnetwork, this poses a limitation to the maximum number of subnetworks per class that can be realized on the HAGEN prototype. In contrast, single-layer architectures do not require any

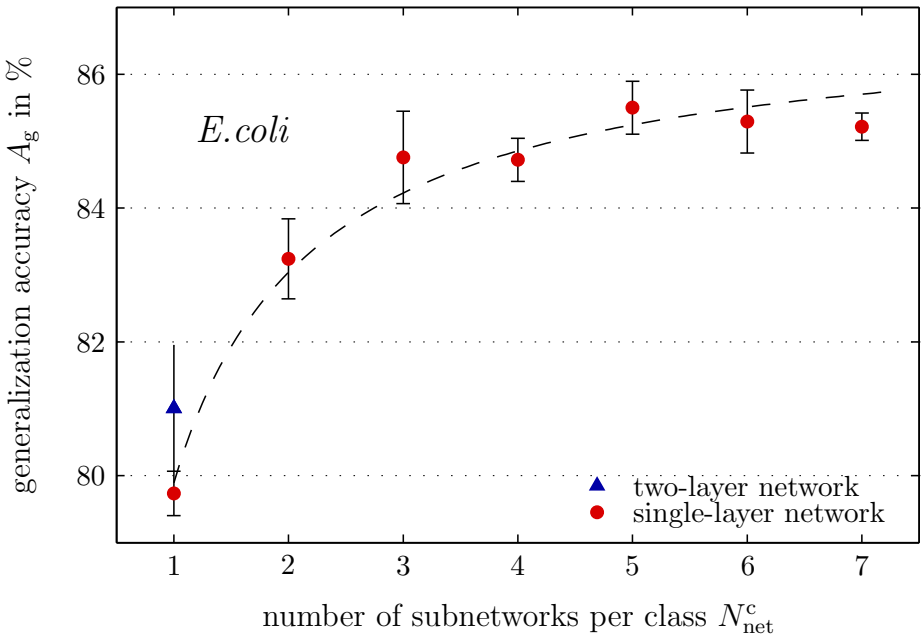


Figure 9.6: *The achieved classification accuracies on the test sets A_g as a function of the number N^c_{net} of subnetworks per class for the* E.coli *problem. Due to the eight classes of this benchmark and given the fixed subnetwork size of $N^c_{hid} = 6$ hidden neurons, only one subnetworks per class can be realized for the two-layer networks. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1).*

feedback connections and since the desired input patterns can easily be applied simultaneously to multiple blocks, single-layer networks can be distributed over the different blocks of the HAGEN chip without restrictions. Here, the size of the network is only limited by the total number of available output neurons. With regard to a feasible increase of the number of subnetworks per class, the use of single-layer networks therefore constitutes an important potential alternative, especially for task with many classes.

single-layer networks: unrestricted

Figures 9.5 and 9.6 present the results obtained with the glass and *E.coli* problems. In these and all following diagrams, the classification rates on the training sets are omitted and only the generalization rates for two-layer and single-layer networks are compared. The complete data is shown in figures C.3, and C.4 in the appendix.

glass and E.coli

For the glass and *E.coli* problems, the networks with hidden layer can at most include two respectively one subnetwork of the agreed size per class. Compared to the generalization rates that are initially achieved with these two-layer architectures, the single-layer networks show a remarkable improvement in performance when the number of subnetworks is increased. In the cases of the previously discussed liver disorder and wine benchmarks, the accuracies of the one-layer networks are observed to converge to approximately the accuracy of a two-layer network with two subnetworks per class. For the glass and *E.coli* problems, the

single-layer networks: efficient

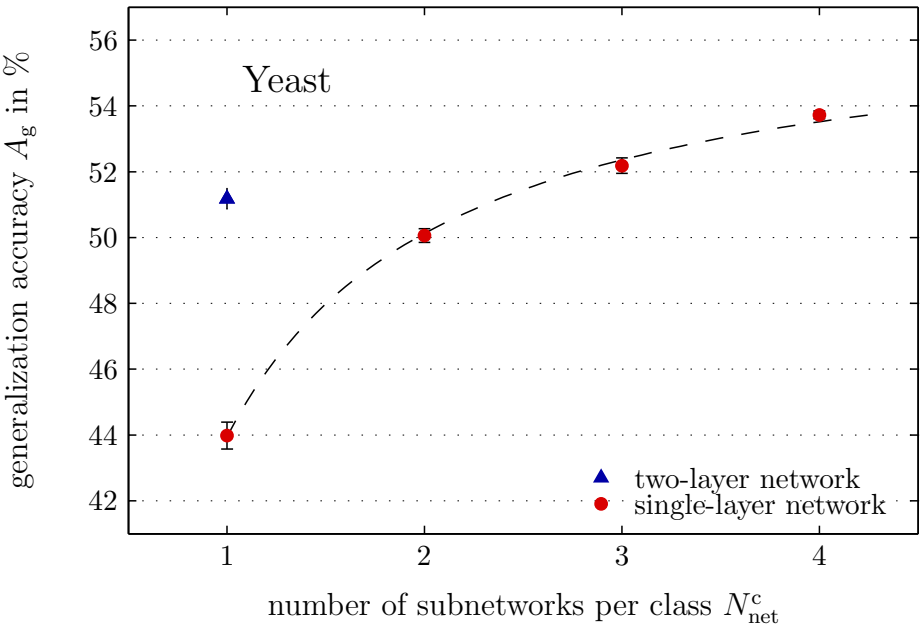


Figure 9.7: *The achieved classification accuracies on the test sets A_g as a function of the number N^c_{net} of subnetworks per class for the yeast problem (note that the error bars are very small). Due to the ten classes of this benchmark and given the fixed subnetwork size of $N^c_{hid} = 6$ hidden neurons, only one subnetworks per class can be realized for the two-layer networks. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1).*

respective increase in the generalization rate is significantly more distinct.

As it has already been stated in section 9.2.1, the performance of the simple single-layer architecture on these benchmarks indicates that the different classes of the respective data sets might to a large extend be pairwise linearly separable. Since the available experimental data for the two-layer networks is only limited, it cannot be inferred whether the training of multiple subnetworks per class might affect networks with and without hidden layer differently. But at least the measurements with the glass problem suggest that two-layer networks can be anticipated to benefit from multiple subnetworks per class in approximately the same way as single-layer networks. This topic will be returned to below.

saturation behavior

For both, the glass and the *E.coli* problem, the avaliable resources on the HAGEN chip would allow to further increase the number of subnetworks. However, the achieved improvement in generalization seems to settle at approximately five subnetworks per class like it is also observed for the other tasks. Although the measurements indicate that a further increase in the number of subnetworks only yields minor improvements, they also suggest that it does not in fact lead to a significant deterioration of the results.

preserved accuracy

This can be seen in direct analogy to equations 9.7 and 9.10 and thus supports the theoretically motivated expectation that the use of a network ensemble will not lead to an actual increase in the observed error. With respect to the training

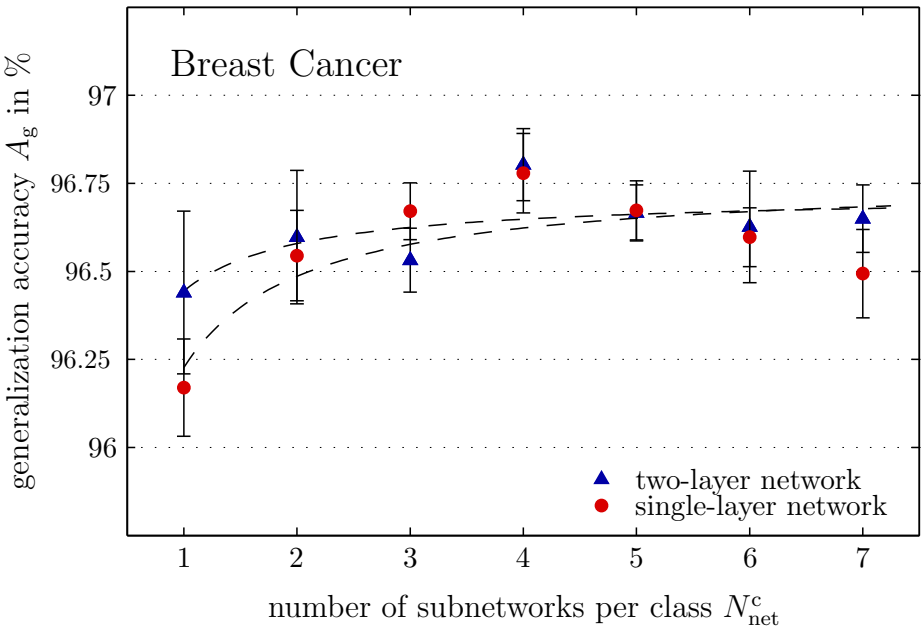


Figure 9.8: *The achieved classification accuracies on the test sets A_g as a function of the number N^c_{net} of subnetworks per class for the breast cancer problem. For this task, the observed increase in performance is not in fact significant compared to the estimated error. The shown dashed lines are intended as mere guides to the eye (see section 9.4.1).*

of networks on the HAGEN chip, this is an important result in so far as it allows to employ the stepwise approach for an arbitrary scaling of the network to fit the available resources without running the risk of overfitting the data.

yeast

Similar results are obtained on the yeast problem (see figure 9.7). Given the size of this data set and the comparably large number of ten classes, only a limited set of measurements is conducted for this problem due to time considerations. Although single-layer networks with up to $N^c_{net} = 6$ subnetworks per class would fit on one HAGEN ASIC, the resulting generalization rates are only evaluated up to $N^c_{net} = 4$ (see also figure C.5). Nevertheless, given the shown data, it is reasonable to assume that the achieved averaged classification accuracy on the test set obeys the same saturation behavior that is also observed on the previously discussed benchmarks. In so far, a further increase of the number of subnetworks per class is expected to yield only minor improvements. For the two-layer architecture, the number of realizable subnetworks per class is once more limited to one.

9.4.3 Approximately Linearly Separable Data Sets

breast cancer

It has already been observed during the measurements presented in table 9.2 that for the breast cancer problem, two-layer and single-layer networks perform nearly equally well. This condition holds when multiple subnetworks per class are trained on this benchmarks. Figure 9.8 shows the corresponding results, and it can be observed that both, single-layer and two-layer networks benefit from an increased

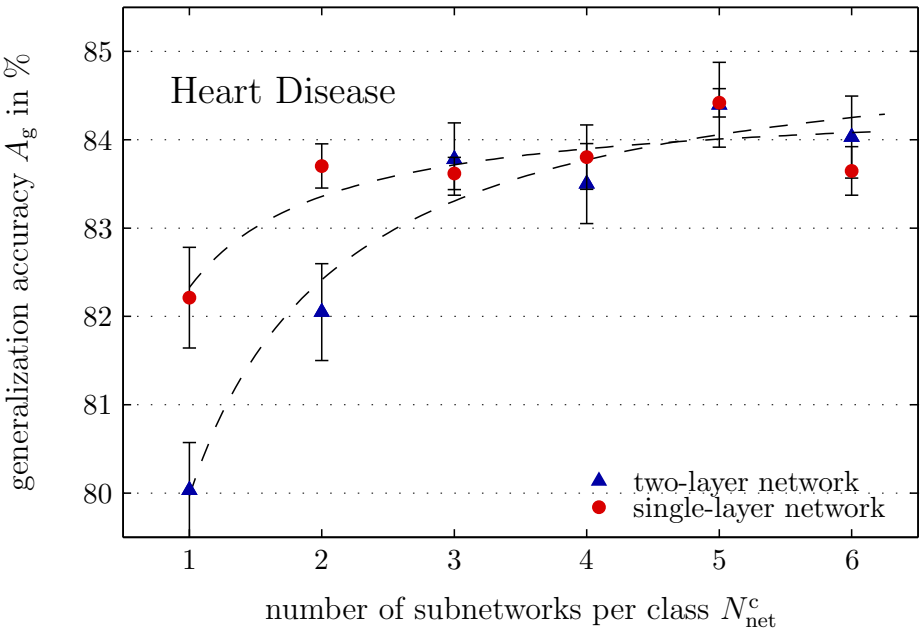


Figure 9.9: *The achieved classification accuracies on the test sets A_g as a function of the number N_{net}^c of subnetworks per class for the heart disease problem. With increasing N_{net}^c, the obtained generalization rates for two-layer and single-layer networks converge. The shown dashed lines are intended as mere guides to the eye (see section 9.4.1).*

number of subnetworks. Beyond $N_{net}^c = 5$, any potential further improvement is occluded by the remaining statistical uncertainty of the measurement.

no significant improvement

Although for the single-layer networks, the generalization accuracy seems to actually decrease again for $N_{net}^c \geq 5$, the estimated error does not allow for a definite conclusion. Additional measurements are necessary to ultimately clarify this question. It is generally observed that for this benchmark, the benefits that arise due to the training of multiple subnetworks per class are not significant compared to the magnitude of the statistical uncertainty. The breast cancer data set is the only investigated task where this is the case.

heart disease

For the heart disease benchmark, single-layer networks are initially observed to be superior over two-layer networks (see table 9.2). An increase in the number of subnetworks compensates for this differences, as can be seen in figure 9.9. A number of subnetworks per category of $N_{net}^c \geq 5$ yields approximately the same performance for both architectures.

converging results

Against the background of the results reported above for the glass and *E.coli* problems, it is an important observation that for tasks where two-layer and single-layer network architectures perform approximately equally well, the remaining differences are reduced even further when multiple subnetworks per class are considered. It can be anticipated that also for the glass and *E.coli* benchmarks, two-layer networks with an increased number of subnetworks might yield about the same performance as it is obtained for the single-layer networks.

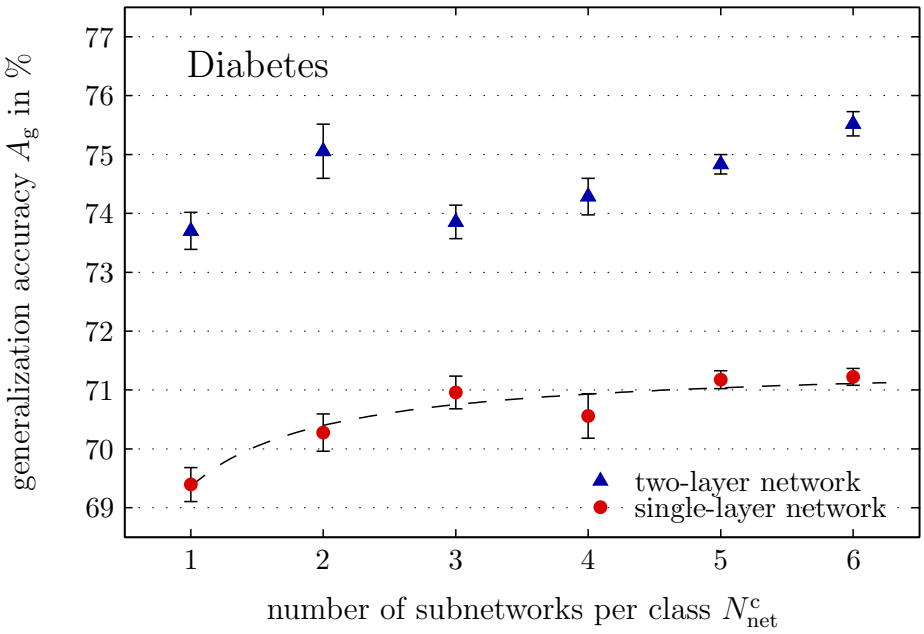


Figure 9.10: *The achieved classification accuracies on the test sets A_g as a function of the number N^c_{net} of subnetworks per class for the diabetes problem. For networks with hidden layer, the observed generalization rates show a nonmonotonic development that is different from those observed on all other tasks. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1).*

9.4.4 Exceptional Cases

diabetes and iris plant

Figures 9.10 and 9.3 present the measurements for the diabetes and iris plant problems. Although they fit into the general picture, the results on these two tasks are peculiar in some aspects. In both cases, the behavior of the single-layer networks corresponds to what is also observed on the other data sets. Slightly different results, however, are obtained with the two-layer networks.

diabetes: nonmonotonic behavior

In the case of the diabetes benchmark, the improvement in performance turns out to be nonmonotonic. A number of subnetworks of $N^c_{net} = 3$ leads to a significantly worse generalization rate than $N^c_{net} = 2$. Given the shown data, it seems reasonable to assume that the point at $N^c_{net} = 2$ is to be regarded as an outlier. The observed behavior might thus be accredited to the remaining statistical uncertainty and the inherently random nature of the evolutionary training algorithm. It then remains that the increase in performance is initially slow and becomes more pronounced for $N^c_{net} \geq 4$. On all other tasks — and with the single-layer networks that are trained on this benchmark — the opposite behavior is observed: For $N^c_{net} < 5$, the generalization rates usually increase measurably, for about $N^c_{net} \geq 5$, the performance slowly saturates.

abnormal behavior

At present, it is not clear whether this seemingly abnormal behavior is connected to specific peculiarities of the diabetes data set or whether it is to be accredited to statistical deviations. Additional measurements are required to illuminate this

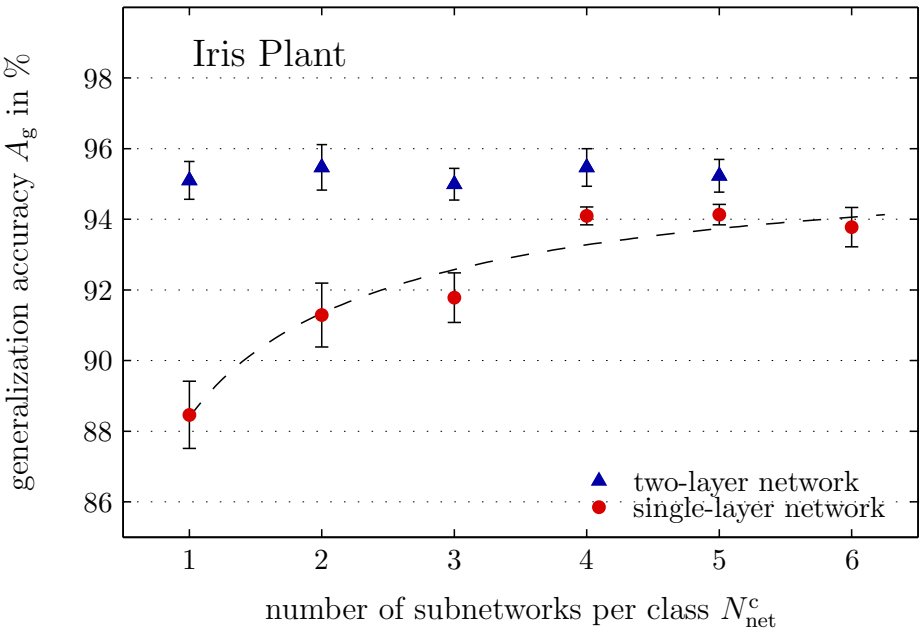


Figure 9.11: *The achieved classification accuracies on the test sets A_g as a function of the number N^c_{net} of subnetworks per class for the iris plant problem. Here, the two-layer networks do not benefit from an increased number of independently trained subnetworks per class at all. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1).*

question. But apart from that, it is important to note that a training of multiple subnetworks per class persists to significantly benefit the generalization capability of the networks also for this benchmark.

iris plant: no improvement

This is not true for two-layer networks that are trained on the iris plant problem (see figure 9.11). Here, including more than one subnetwork per category does not yield any significant change in the achieved generalization rates at all. Again, this behavior is unique among all investigated data sets.

efficient single-layer networks

In this context, it is an important result that with an increasing number of subnetworks, the generalization rates of the single-layer networks approximately converge to the maximum accuracy that can be observed for two-layer networks. This is remarkable in so far as two of the three classes of this benchmark are known to be not linearly separable. It seems that the use of multiple, independently trained networks can at least partially compensate for the limitation to linearly separable problems that is in principle inherent to the single-layer perceptron architecture (see section 2.1.1).

"parallel perceptrons"

It has recently been shown by Auer *et al.* that n independently trained simple perceptrons whose outputs are combined to a single binary response via majority voting can compute every Boolean function $f_B\colon \{0,1\}^n \to \{0,1\}$ [6]. The N^c_{net} subnetworks that are trained for the same class during the proposed stepwise procedure are not combined by majority voting (see sections 8.2.3 and 9.3.2). In fact, the increased number of outputs per class is rather intended to allow for a more

differentiated comparison with the remaining networks that are trained for the other classes. In so far, it is understood that the above universal approximation theorem for "parallel perceptrons" [6] cannot directly be applied. But the experimental data shown in figure 9.11 suggests that similar propositions might also hold for single-layer networks that are trained by the generalized stepwise strategy. A more detailed theoretical investigation of this question exceeds the scope of this thesis but promises to be an interesting topic for future investigations.

Summarizing Remarks

improved results

For all investigated benchmark problems — except for the breast cancer problem — the training of multiple subnetworks per class yields some kind of measurable improvement of the achieved classification accuracies. At the same time, both, the simplicity of the evolutionary training algorithm that is used in each phase and the favorable inherent parallelism of the original stepwise approach are preserved.

single-layer networks: feasible

Furthermore, given the results that are obtained on the breast cancer, heart disease, iris plant, glass, and *E.coli* data sets, it can be concluded that the generalized stepwise training strategy also allows for the application of single-layer perceptrons to demanding, real-world tasks — partly even for those problems that cannot ordinarily be solved satisfactorily by such simple networks. Apart from being an interesting observation by itself, this particularly suits the used HAGEN prototype, since single-layer networks are not affected by the restrictions that arise from the limited set of available, hard-wired feedback connections or the fixed size of the network blocks.

results for different architectures

It is another important outcome of the presented measurements that the training of multiple subnetworks per class tends to compensate for the differences in performance between the investigated two-layer and single-layer architectures. Especially in the case of the heart disease problem, the choice of a network with hidden layer has initially been encountered to be problematic (see section 9.2.1). But when it is possible to train a sufficient number of subnetworks per class, the two-layer architecture is seen to be at least as efficient as the simple single-layer topology on all respective benchmarks.

architecture optimization: largely redundant

These results indicate that the generalized stepwise strategy can largely relieve the user — or the training algorithm — from determining the optimal architecture for a specific task: If the number of classes in the considered problem allows to realize a number of two-layer subnetworks of about $N_{\text{net}}^{\text{c}} = 5$, the two-layer architecture is to be chosen. Otherwise, satisfactory generalization rates can still be achieved with the single-layer architecture. At this point, having experimentally investigated only a limited number of exemplary benchmarks, this directive is admittedly nothing more than an empirical rule of thumb. Nevertheless, it persists that in all examined cases where a sufficient number of subnetworks can be implemented on the HAGEN chip, the two-layer architectures never perform measurably worse than the single-layer networks.

In summary, the proposed generalized stepwise strategy suggests itself as a feasible way of training enlarged networks that can efficiently exploit the resources of the HAGEN ASIC. But it remains to be investigated whether the performance

of the hereby constructed networks can eventually compete with those of other systems.

9.4.5 Comparison to Previous Results

It has already been stated in section 8.4.2 that most of the reported experiments that evaluate the performance of other neural network setups or classification systems on the selected benchmarks do not employ a stratified 10-fold cross-validation scheme. In order to allow for a direct comparison with these results, an additional set of experiments is performed where for each benchmark, the same evaluation scheme is employed that is also used in a corresponding earlier publication. Apart from alternative training setups for single neural networks, the performance of the introduced stepwise strategy is compared also to other classification algorithms — such as support vector machines and k nearest neighbor classifiers — as well as to contemporary approaches for the training of neural network ensembles.

Fixed Partitionings and Single Neural Networks

"Proben1"

Among numerous other benchmarks, Prechelt [163] examines multiple feedforward neural network architectures on the breast cancer, diabetes, heart disease and glass problems. For each task, a set of three different partitionings into training and test data is provided. Since these partitionings are all included in the "Proben1" benchmark suite that is introduced in the same publication, the respective combinations of training and test data shall be denoted as "Proben1 a)", "Proben1 b)", and "Proben1 c)".

EPNet

The breast cancer and diabetes problem are also investigated by Yao and Liu [236] who evaluate the performance of their EPNet system (see section 4.2.2) on these tasks. They use one fixed partitioning of the data for each benchmark, and their investigations also include the heart disease problem. But unfortunately, the precise compositions of training and test set for this task cannot unambiguously be reconstructed.

training set and validation set

In both publications, the training data is divided further into an actual training set and a so-called validation set [163] [236]. The training of the network then only employs the former while the latter is exclusively used to determine the termination of the algorithm: During training, the network is periodically tested on the validation set, usually after a fixed number of iterations. While the classification accuracy on the training set is expected to improve approximately monotonically, the performance on the validation set might stagnate or even deteriorate due to overfitting of the data. If this is the case, the training is terminated. The evolutionary stepwise strategy introduced in this thesis does not employ any validation data during training. The subsets of the data that are marked as the validation sets are therefore appended to the respective training sets.

training setup

Table 9.3 compares the results of the cited publications with the generalization rates that are obtained by neural networks on the HAGEN chip. The networks are trained with the same algorithm and network setup that is also used during the investigations described in the previous sections. The only exception is that

during the evaluation of a network's classification accuracy on the test data, each pattern is applied only $m_p^g = 20$ times instead of the 50 repetitions used for the previous experiments (see section 8.4.1 and table 8.3). This accounts for the fact that according to the used fixed partitionings, the different test sets now contain about 25 % of the total instances in the respective data set.

network architecture

In case of the glass problem, the network architecture comprises only one single layer; for the other tasks, the networks have two layers. The numbers of subnetworks per class that are contained in the trained networks are included in the table.

evaluation procedure

For each of the used fixed partitionings, a total of $N_\mathrm{n} = 10$ entire networks is trained, and the one with the best classification rate on the training data is regarded as the result. This is repeated $N_\mathrm{r} = 5$ times for each partitioning and the obtained averaged generalization accuracies A_g together with the corresponding standard deviations are listed in table 9.3. Only the classification rates on the test data are compared. For the networks trained on the HAGEN chip, the performance on the training data A_t can be found in table C.3 in the appendix.

compared networks

A variety of results are reported in the "Proben1" collection that each employ slightly different network architectures and transfer functions [163]. The networks are trained with a variant of the backpropagation algorithm (see section 2.2.2). For each individual partitioning of the individual data sets, table 9.3 lists the best respective result that can be found in this publication regardless of the employed network setup. Since some of these values represent the generalization rates of individual, outstanding networks, the respective results include no error estimates. The remaining values represent the mean of 60 independent training runs (10 in the case of the heart disease problem) and the provided error measures are given by the corresponding standard error of the mean[1]. Within the experiments reported by Yao and Liu, the network topology is optimized by the training algorithm (see also section 4.2.2). Here, the cited results each represent the mean of 30 training runs with the EPNet system. The generalization rates that are obtained by these two previous investigations represent the best results of single neural networks on the considered benchmarks that have been found in literature and that can also exactly be reproduced due to an unambiguous specification of the used training and test data. The performance of recent neural network ensemble approaches will be discussed below.

results: competitive

A comparison to the generalization accuracies that are obtained with networks on the HAGEN chip reveals that the introduced stepwise training strategy in combination with the used strictly layered, feedforward perceptron architecture constitutes a competitive approach to tackle the investigated problems. In most cases, the trained networks either achieve better generalization rates than those obtained by the previous setups (marked red in table 9.3) or perform approximately equally well (blue entries).

This result is particularly remarkable in so far, as neither the various parameters of the used evolutionary training algorithm nor the settings of the generalized

[1]The provided standard errors of the mean are calculated from the respective standard deviations and the number of experiments that are reported in the two cited publications.

benchmark	fixed partitioning	$N_{\text{net}}^{\text{c}}$	generalization accuracy A_{g} in %	
			stepwise strategy	previously reported [163] [236]
breast cancer	Proben1 a)	7	98.73 ± 0.39	98.85
	Proben1 b)		94.95 ± 0.28	95.48 ± 0.10
	Proben1 c)		95.94 ± 0.04	97.71
	Yao *et al.*		99.12 ± 0.15	98.63 ± 0.17
diabetes	Proben1 a)	6	83.88 ± 0.44	75.90 ± 0.25
	Proben1 a)		84.86 ± 0.57	76.56
	Proben1 b)		82.65 ± 0.40	78.65
	Yao *et al.*		78.05 ± 1.29	77.63 ± 0.003
heart disease	Proben1 a)	6	81.60 ± 0.50	80.27 ± 0.18
	Proben1 b)		94.13 ± 0.68	96.80 ± 0.49
	Proben1 c)		84.71 ± 1.28	85.73 ± 0.22
glass	Proben1 a)	8	72.75 ± 0.74	67.92
	Proben1 b)		62.64 ± 0.71	47.17
	Proben1 c)		63.57 ± 0.92	66.04

Table 9.3: *Comparison of the results that are obtained by a stepwise training of networks on the HAGEN chip with the generalization rates that are reported by other authors. The cited results are achieved on fixed separations of the tasks into training and test data. Prechelt [163] specifies three different fixed partitionings for each task that are denoted as "Proben1 a)–c)". Yao and Liu use one fixed separation [236]. For the experiments with the stepwise training strategy, the exact compositions of the respective training and test data sets are reproduced. It is to be noted that during the investigations reported by Yao and Liu, the network architectures are optimized for the task in question [236]. The results cited from Prechelt represent the best generalization rates that can be found in the corresponding publication regardless of the actual network architecture. Generalization accuracies A_{g} which exceed those reported by the other authors are set in red color. All cases where the results are approximately equal within the estimated error bounds are shown in blue. On the remaining partitions, the stepwise strategy performs measurably worse.*

stepwise approach — like, e.g., the size of the subnetworks — have been subject to any optimization. The same simple training strategy is successfully applied to all four benchmarks and the trained networks can readily compete with networks whose architectures have especially been optimized for the respective task. This even holds for the glass problem where the trained network employs a mere single-layer architecture.

Cross-Validation Experiments and Alternative Classifiers

In the cases of the liver disorder, iris plant, wine, *E.coli*, and yeast data sets, experiments are reported in literature that employ N-fold cross-validation measurements similar to those used for the experiments in this thesis but with varying numbers of subsets. The selected publications represent the best of this kind of results that have been found in literature for non-ensemble classification methods.

wine: comparison problematic

The best reported generalization rate for the wine problem is obtained by an exhaustive leave-one-out test [3] which yields an accuracy of 100 % . If the previously used cross-validation scheme (see section 8.1.2) was extended to a number of partitionings of $N_\mathrm{p} = 178$ (the number of instances in the wine data set), a total of $5 \cdot 10 \cdot 178 = 8900$ networks would have to be trained. In consideration of the required time, it has been abstained from doing so. It has already been stated in section 8.1.2 that the reliability of the leave-one-out procedure is in fact deemed to be inferior to cross-validation schemes with 5–10 partitionings.

evaluation setup

For each of the the remaining benchmark problems, a stratified cross-validation experiment is performed with a number N_p of partitionings that equals the one which is also used in the respective reference publication. Apart from that, the experimental setup and evolutionary algorithm correspond to those utilized for the previously described experiments. For the *E.coli* and yeast benchmarks, the used networks employ a simple one-layer architecture, while the networks for the remaining tasks contain one hidden layer.

compared systems

The achieved generalization rates together with the results of the cited publications are listed in table 9.4. As it has already been stated in section 8.4.2, these investigations do not examine neural networks but use support vector machines [65], approximate distance classifiers [30], statistical methods [3], or k nearest neighbor classifiers [101]. Apart from the generalization accuracies that are cited for the *E.coli* and yeast problems, none of these experiments uses a stratified partitioning of the data, i.e., the N_p individual subsets are created entirely randomly without further constraints. The measurements that are conducted with the stepwise strategy persist to use stratified cross-validation (see section 8.1.2).

results: slightly inferior

It can be inferred from table 9.4 that feedforward networks on the HAGEN chip which are trained and constructed according to the proposed stepwise procedure cannot actually outperform competing approaches on the selected tasks. For the iris plant and *E.coli* benchmarks, the generalization rates of the compared classifiers are only marginally better. In the cases of the liver disorder, wine, and yeast problems, the differences are more distinct.

For the liver disorder and wine benchmarks, it cannot be excluded that the remaining discrepancies might partly be accredited to the slight differences that

benchmark	$N_{\text{net}}^{\text{c}}$	generalization accuracy A_{g} in %			cross-validation
		stepwise strategy	previously reported		
liver disorder	6	70.98 ± 0.52	73.7	[65]	10-fold
iris plant	5	95.41 ± 0.47	96.2	[30]	5-fold
wine	5	97.71 ± 0.28	*100**	[3]	(10-fold)
E.coli	7	84.89 ± 0.49	86	[101]	4-fold
yeast	4	53.73 ± 0.12	60	[101]	10-fold

Table 9.4: *The results that are achieved with the introduced generalized stepwise strategy are compared with the generalization rates that are reported by previous experiments. The cited publications employ N-fold cross-validation schemes with varying numbers of partitionings. The corresponding values are listed in the last column.* **In the case of the wine problem, the cited investigation actually includes a leave-one-out measurement, but in consideration of the required time, a corresponding experiment has not been conducted for the stepwise strategy. The shown generalization rate is the outcome of a stratified 10-fold validation. Therefore, a direct comparison between the two results remains problematic. It is to be noted that apart from the results reported for the* E.coli *and yeast benchmarks, the cited publications perform a random, unstratified cross-validation.*

remain between the used cross-validation procedures. But it is to be expected that these effects do not significantly affect the general observation that on all examined task, the proposed stepwise training procedure performs slightly worse than the respective best classifier system that is found for each benchmark in literature.

satisfactory performance

Nevertheless, although a superior classification algorithm does exist for each task, it remains that the stepwise training of feedforward networks on the HAGEN chip evidently constitutes a more than satisfying approach for all tasks alike. As it has already been emphasized in the preceeding section, none of the parameters of the used stepwise training approach has undergone any optimization. Moreover, the networks for the *E.coli* and yeast benchmarks merely exhibit a simple single-layer architecture. Under these adversarial conditions, it is a satisfactory observation that the employed divide-and-conquer training approach readily achieves generalization rates that are comparable to the results of other systems.

Comparison to Neural Network Ensembles

As it has been discussed in section 9.3.2, the generalized stepwise training strategy is comparable to common neural network ensemble approaches in several respects. Therefore, the performance of the introduced training procedure is also compared to the generalization accuracies that have been reported for committees of networks on the same benchmark problems.

Bagging

One popular way of promoting the diversity within an ensemble of networks is to train the individual committee members on different sets of training patterns. According to the so-called Bagging procedure, the training set for each network is

benchmark	$N_{\text{net}}^{\text{c}}$	generalization accuracy A_{g} in %		
		stepwise strategy	Bagging [153]	CNNE [105]
breast cancer	7	96.65 ± 0.10	96.6	98.9
diabetes	6	75.52 ± 0.21	77.2	82.2
heart disease	6	84.03 ± 0.46	83.0	88.8
glass	8	67.33 ± 0.26	66.9	75.4

Table 9.5: *The results that are achieved with the introduced generalized stepwise strategy are compared to the generalization rates that are reported by other investigations on neural network ensembles. The cited publications both employ an unstratified 10-fold cross-validation scheme. Bagging is a popular approach where the individual members of the ensemble are trained on different training sets [153]. The CNNE algorithm proposed by Islam* et al. *incorporates elaborate schemes of adjusting the number and architecture of the single networks and promoting the diversity within the ensemble by negative correlation learning [105]. The generalization rates that are achieved by this latter approach are by far the best that have been encountered for the investigated tasks in literature.*

generated by randomly drawing, with replacement, N_{t} examples from the original training data, where N_{t} is the number of instances in the whole training set [153]. Some of the instances will then appear multiple times in the training data of a given network, while others will be left out. Since each network in the ensemble is trained on a new, randomly created training set, it is expected that their responses will exhibit a measurable disagreement.

compared ensembles

It has been discussed in section 9.3.2 that more recent approaches actively enforce the diversity of the networks in the ensemble during training. Among other benchmarks, Islam *et al.* [105] test their proposed CNNE (Constructive Neural Network Ensemble) algorithm also on the breast cancer, diabetes, heart disease, and glass problem and compare it to the results that are reported for a Bagging ensemble by Opitz *et al.* [153].

evaluation procedure

In both investigations, a 10-fold cross-validation scheme is employed and the reported results can therefore directly be compared to the generalization rates that are achieved by the introduced stepwise strategy. The respectively obtained classification accuracies on the test sets are summarized in table 9.5.

comparison

Compared to the results of the Bagging ensemble, the stepwise strategy is once more observed to exhibit a competitive performance. Still, the generalization rates that are achieved by the CNNE approach are not only significantly better than those of the Bagging ensemble, to the author's best knowledge they also represent by far the best results that are reported for N-fold cross-validation measurements on the respective benchmarks in literature. It can be inferred from table 9.5, that the generalization accuracies that are achieved with the introduced stepwise strategy cannot compete with these results.

CNNE

As it has already been stated in section 9.3.2, the CNNE model involves a complex learning procedure that actively promotes the diversity of the individual networks in the ensemble. During training, the weight modifications that are

applied to each single network are affected not only by its own output but also by the current performance of the other networks, such that any correlation between the responses of the different networks can effectively be reduced [105]. In other words, the networks need to be trained simultaneously and cannot be considered independently. Furthermore, elaborate construction schemes are applied in order to optimize the individual architectures as well as the number of networks in the committee during training.

complexity vs. speed

Given the complexity of this approach, it is not surprising that the training of networks via the generalized stepwise strategy — which is optimized for simplicity, fast training speeds, and inherent parallelism — is eventually outperformed by the CNNE algorithm. On the other hand, an elaborate approach like the CNNE procedure is unlikely to achieve an efficient exploitation of the speed benefits that arise from the use of a neural network hardware. In particular, the interdependencies that remain between the networks during training do not allow for the same degree of parallelism that can readily be achieved by the proposed stepwise strategy. In so far, it is a more than satisfying observation that the stepwise training is comparable in performance to common neural network ensemble approaches and at the same time allows to efficiently utilize the evolutionary coprocessor for the fast training of neural networks on the HAGEN chip.

9.4.6 Multiple Subnetworks vs. Increased Subnetwork Size

Having demonstrated the potential of the generalized stepwise approach, the chapter is to be concluded by an investigation of whether an alternative increase of the subnetwork size might actually yield an equal or even better performance. A set of experiments is conducted where only one subnetwork is trained for each class of the investigated problem and where the classification accuracies of the final networks on the used test sets are evaluated as a function of the number of hidden neurons $N_{\text{hid}}^{\text{sn}}$ and outputs $N_{\text{out}}^{\text{sn}}$ per subnetwork.

equal network size

In order to allow for a direct comparison with the training of multiple subnetworks per class, the enlarged subnetworks are scaled to contain numbers of hidden nodes and outputs that are simple integer multiples m_{s} of the agreed initial parameters $N_{\text{hid}}^{\text{sn}} = 6$ and $N_{\text{out}}^{\text{sn}} = 4$, respectively. Therefore, the resulting networks comprise an equal total number of neurons as those that are obtained by an independent training of m_{s} subnetworks for each category. The allowed number of generations of the evolutionary training algorithm in each phase is scaled accordingly, i.e., an increase of the network size by a factor of m_{s} is accounted for by multiplying the number of training iterations per phase by the same factor. Hence, the total amount of processed generations is the same as for the corresponding experiment with the generalized stepwise strategy described in the preceeding sections. Apart from the increased subnetwork size and the extended training time, the experiments follow the same setup as described in section 9.2.

equal training time

General Observations

liver disorder

In figure 9.12, the results that are obtained on the liver disorder problem are compared with the performance of the generalized stepwise strategy investigated in section 9.4.1. For both results to be displayed in the same diagram, the achieved classification accuracies on the test sets A_g are shown as a function of the number of output neurons of the networks. The circular dots refer to networks with increased subnetwork size, the triangles represent those with multiple subnetworks per class. It is to be repeated that each pair of corresponding data points effectively represents networks with the same size that have been trained for the same total number of iterations. Merely the training procedures are different. The upper diagram corresponds to two-layer networks, the lower part refers to single-layer networks.

enlarged subnetworks: inferior

The presented results immediately reveal that a straight-forward increase of the subnetwork size is measurably inferior to the alternative training of multiple subnetworks with fixed numbers of neurons. For the two-layer networks, an enlargement of the class specific subnetworks is indeed observed to yield a slight increase in the achieved generalization rates. But these improvements cannot compete with the benefits that arise due to a training of multiple subnetworks per class.

In the case of the single-layer architecture, an increased subnetwork size leads to only slightly improved generalization rates, and an enlargement to the fifefold or sixfold of the original size even yields a distinct deterioration of the observed accuracy. The efficient training of enlarged single-layer networks for this benchmark obviously exceeds the capability of the used simple evolutionary algorithm. The independent training of multiple subnetworks per class does not equally suffer from such limitations of the employed training procedure.

wine

A similar behavior is commonly observed on all examined data sets. Figure 9.13 shows the outcome of corresponding experiments with the wine benchmark. Here, an increase of the subnetwork size turns out to leave the obtained generalization rates at best unaffected. For several specific numbers of hidden neurons and/or outputs, the performance even seems to be significantly reduced. However, in the light of the estimated errors, it is to be said that these effects do not necessarily need to follow any systematics. Nevertheless, against the background of this behavior and comparing the magnitudes of the estimated errors with those obtained for the networks with multiple subnetworks per class, it can be concluded that the training of enlarged subnetworks is also disadvantageous in terms of reliability — at least on this data set.

Examining Different Benchmarks

E.coli

As an example for a problem with more than three classes, figure 9.14 presents the results for the *E.coli* data set. Due the constraints of the HAGEN prototype, only single-layer networks are considered for this task. The results are comparable to those obtained with the liver disorder benchmark. Increasing the subnetwork size does indeed yield a measurable improvement in performance, but the training

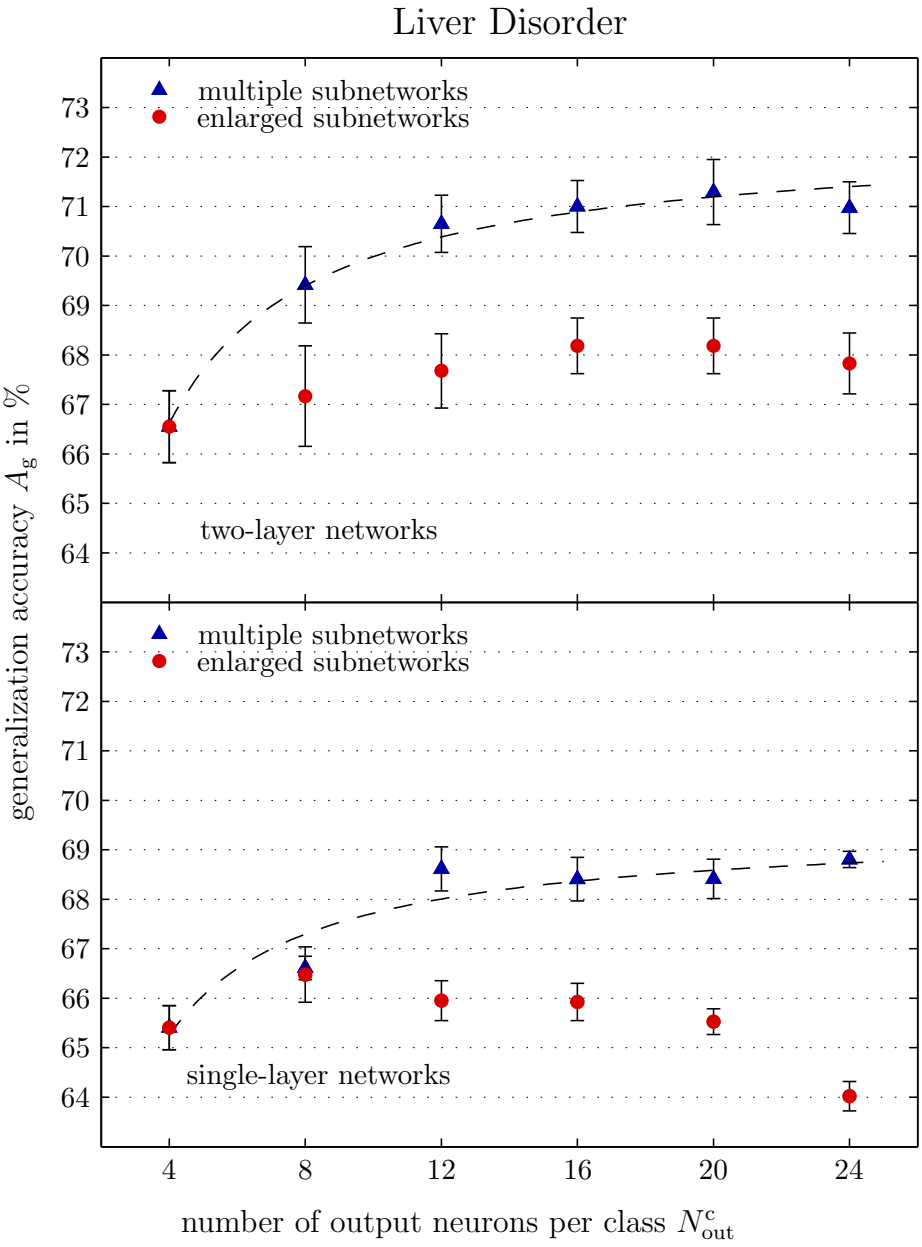


Figure 9.12: *The generalization rates A_g on the liver disorder data set that are achieved by networks with increased subnetwork size are compared to the obtained accuracies of networks that contain multiple subnetworks per class (see figure 9.3). The numbers of output neurons and hidden nodes of the enlarged subnetworks are integer multiples m_s of the initially agreed values $N^{sn}_{hid} = 6$ and $N^{sn}_{out} = 4$. Therefore, the compared networks are effectively equal in size, and the obtained generalization accuracies can be shown as a function of the total number of output neurons per class. The upper part refers to two-layer architectures, the lower half represents singe-layer networks. The shown dashed lines are intended as mere guides to the eye (see section 9.4.1).*

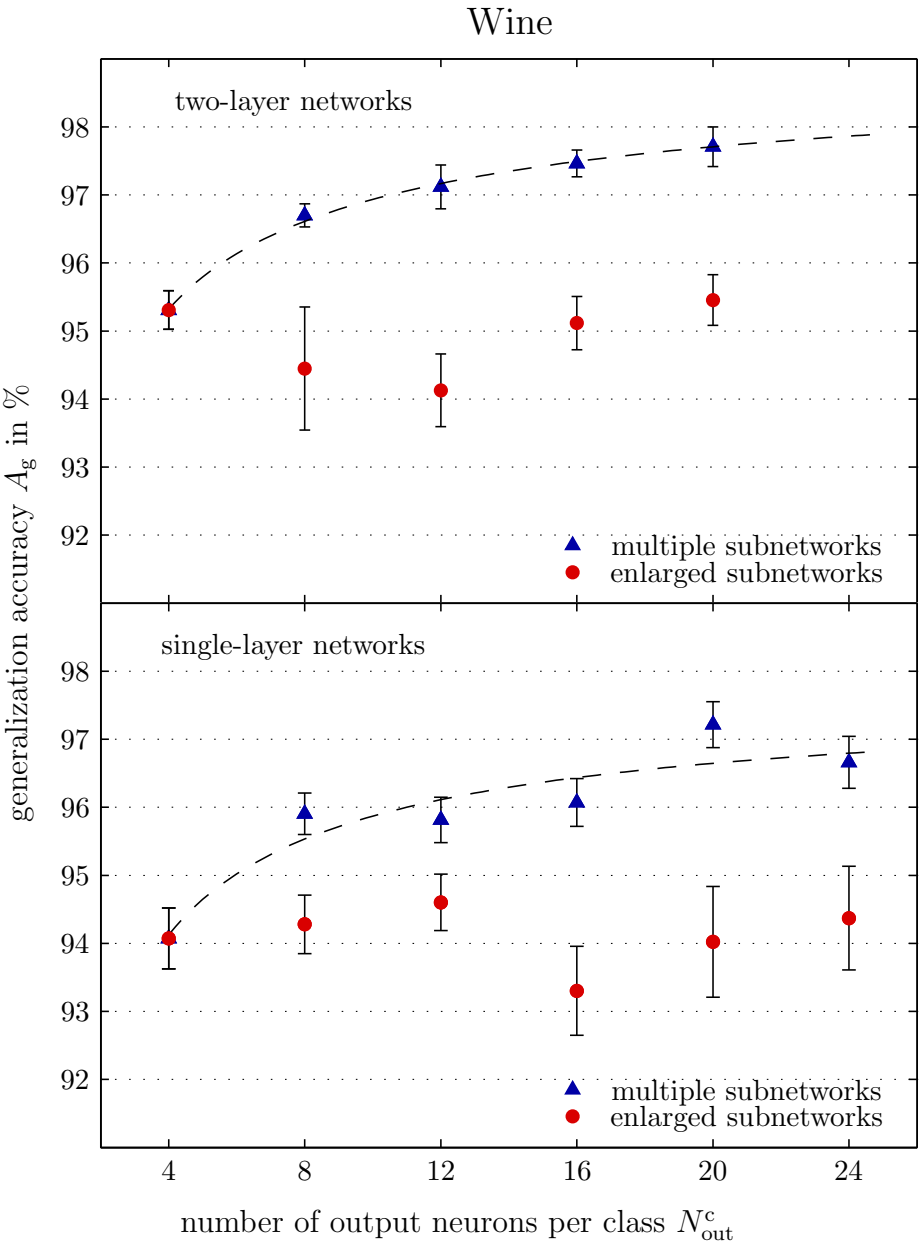


Figure 9.13: *The generalization rates A_g on the wine data set that are achieved by networks with increased subnetwork size are compared to the obtained accuracies of networks that contain multiple subnetworks per class (see figure 9.4). The upper part refers to two-layer architectures, the lower half represents singe-layer networks. The shown dashed lines are intended as mere guides to the eye (see section 9.4.1).*

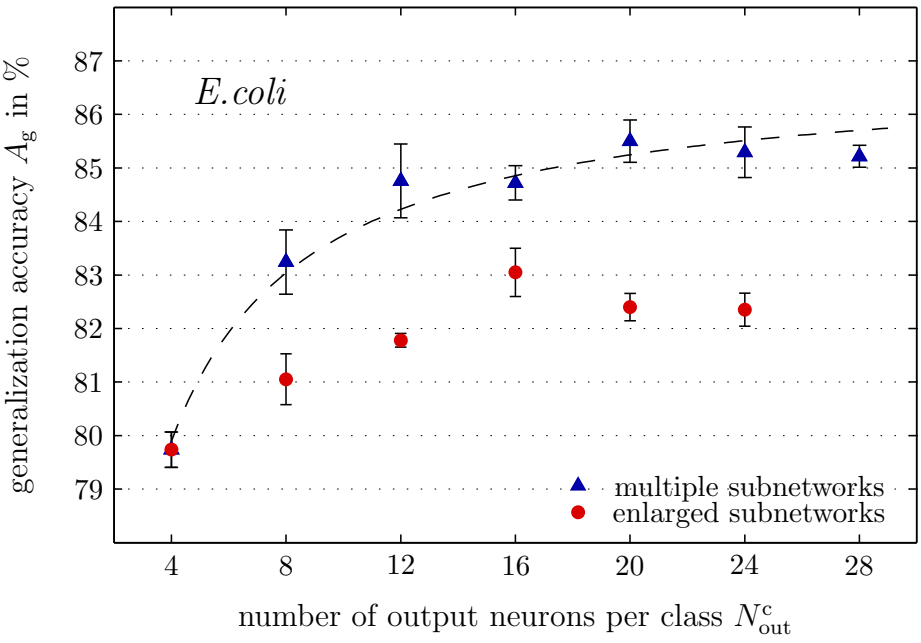


Figure 9.14: *The generalization rates A_g on the* E.coli *data set that are achieved by networks with increased subnetwork size are compared to the obtained accuracies of networks that contain multiple subnetworks per class (see figure 9.6). The shown dashed line is intended as a mere guide to the eye (see section 9.4.1).*

of multiple subnetworks per class is significantly more efficient.

breast cancer and heart disease

Except for the yeast and diabetes problems, corresponding experiments are performed for all remaining benchmarks. The results can be found in appendix. For the breast cancer and heart disease problem, the discrepancy between the two training approaches is observed to be less distinct (figures C.11 and C.12). In the case of two-layer networks that are trained for the former data set, the results seem to indicate that an increase of the subnetwork size might even be superior to the training of multiple subnetworks. However, in the light of the obtained statistical accuracy, a definite conclusion cannot be drawn with certainty and the two approaches are to be regarded as at best equally effective.

iris plant

On the iris plant benchmark, it has been observed that the training of multiple two-layer subnetworks per class does not yield any improvement in the generalization performance (see figure 9.11). It turns out that he same applies to an alternative enlargement of the subnetworks (see figure C.13). One possible interpretation of this result is that an averaged generalization rate of $A_g \approx 95.4\,\%$ represents the maximum that can be achieved on this data set with the used kind of two-layer neural network and that any enlargement of the networks is thus redundant. The results obtained for single-layer networks are consistent with the observations on the other tasks, i.e., the training of more subnetworks per class is clearly to be favored above an increase of the subnetwork size.

9.5 The Stepwise Strategy: Conclusion

successful training

The presented strategy of independently optimizing and interconnecting multiple subnetworks for each class of the investigated categorization task is shown to be a feasible approach to successfully train large neural networks on the HAGEN ASIC for demanding real-world problems. Each of the single training phases can be accomplished by a simple evolutionary algorithm that is itself well suited for hardware acceleration. The presented experiments demonstrate — for the first time — that networks on the HAGEN chip provide more than competitive means for solving challenging classification benchmarks compared to software-implemented neural networks (see table 9.3).

exploited resources

It is shown that by increasing the number of subnetworks per category, the entire network can readily be scaled to the desired extent, limited only by the available resources on the HAGEN ASIC. The hereby increased network size does not lead to a measurable overfitting of the data, i.e., the proposed stepwise training approach exhibits a favorable up-scaling behavior. Although an alternative increase of the subnetwork size can yield an improvement of the network performance as well, it is observed to be clearly inferior to the training of multiple subnetworks per class on the majority of examined benchmarks.

alternative black-box approaches

Theoretical considerations suggest that a sufficient dissimilarity between the individual subnetworks of the same class constitutes a vital precondition for the success of the generalized stepwise strategy. Apart from that, it is reasonable to expect that other model-free chip-in-the-loop training algorithms can be employed for the single training phases as well. In so far, the simulated annealing and weight perturbation algorithms which have been introduced in section 4.3 provide potential alternatives to the evolutionary training approach. However, while simulated annealing — similar to evolutionary optimization — is highly indeterministic and can therefore be assumed to automatically promote a reasonable diversity, the weight-perturbation approach might not be able to yield a sufficient disagreement between the subnetworks that are trained for each class. A thorough examination of alternative optimization algorithms exceeds the scope of this thesis and is deferred to future investigations.

Having demonstrated the potential of the stepwise evolutionary training strategy in principle, several aspects remain to deserve further evaluation, e.g., how to efficiently exploit its inherent parallelism or in how far the performance of the final network can be optimized by appropriately adjusting the subnetwork size. These and some other topics will be investigated in the following chapter.

Chapter 10

Hardware Implications

The trouble with facts is that there are so many of them.

Samuel McChord Crothers, The Gentle Reader

There are several important aspects of the used hardware neural network framework in general (see chapter 6) and the introduced stepwise evolutionary training strategy in particular (see chapter 9) that deserve further consideration but have not yet been examined in detail. This final chapter is to provide a more thorough discussion of these topics.

Most notably, the succeeding sections will discuss the training speed that can be achieved within the currently used neural network framework. It will be demonstrated that — as it has repeatedly been claimed in the foregoing chapter — the proposed stepwise strategy allows for an efficient parallelization of the training procedure. Furthermore, it has already been discussed in section 5.5 that the utilization of calibrated HAGEN chips in principle allows to transfer the network configurations that have been trained for one ASIC also to other chips. In section 10.2, it will be evaluated in how far this affects the performance of the transferred networks.

The remaining sections of this chapter will present several preliminary investigations that are not as exhaustive as the other measurements and are primarily intended to provide an outlook to future experiments: Section 10.3 examines how the eventual success of the proposed stepwise strategy is affected by modifications to the size and architecture of the single subnetworks (see sections 9.1.1 and 9.3.1). Section 10.4, finally, is concerned with software-implemented networks. It is investigated whether the concepts of the stepwise approach might also be applicable to an ideal network that accurately obeys equation 5.4 and does not suffer from the inevitable device variations and temporal fluctuations that are present in the used HAGEN ASIC.

10.1 Training Speed and Parallelization

parallelized network evaluation

During evolutionary chip-in-the-loop training, the testing of individual networks on the used HAGEN ASIC and the evaluation of their fitness values on the basis of

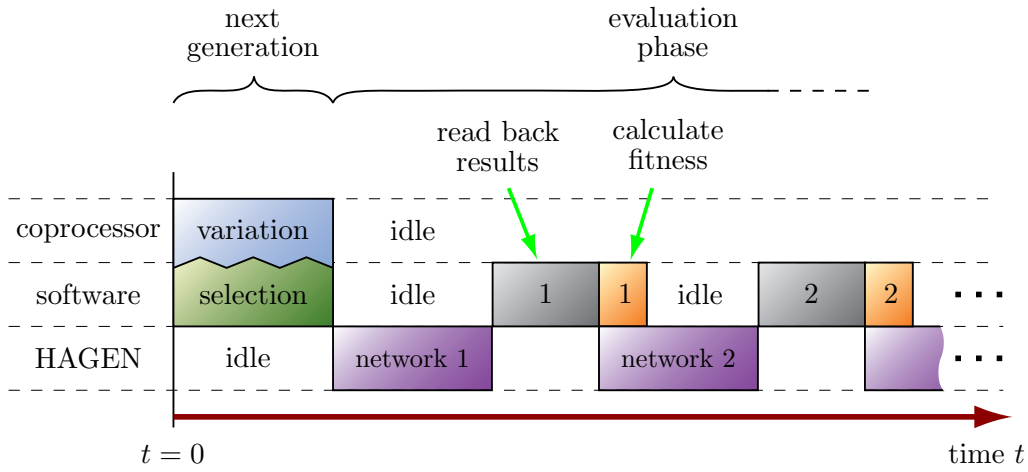


Figure 10.1: *Schematical illustration of the evolutionary chip-in-the-loop training procedure. While the selection scheme is implemented purely in software (green), the necessary genetic operations are assumed by the evolutionary coprocessor (blue, see section 6.2). In the current setup, the testing of the new individuals on the hardware does not commence before the creation of the next generation is finished completely. On the other hand, the fitness calculation (orange) is performed in parallel to the network execution (purple): While the next individual is already tested on the HAGEN chip, the fitness of the previously implemented network is simultaneously calculated in software. At present, the training speed is limited by the required transfer of the individual network responses over the PCI bus (grey, see section 6.1.3), while the fitness can actually be calculated in about one fourth of the CPU time that is available during a network execution (for the exact numbers see section 10.1.1).*

the obtained network responses are partially performed in parallel: While a given candidate solution is executed on the hardware via the multi-threaded version of the `HNetMan::run(·)` method (see also section 7.3.2), the fitness of the previously tested network is already evaluated in software. Figure 10.1 schematically illustrates this procedure.

software-implemented selection process

The genetic variation operations are performed by the evolutionary coprocessor (see sections 6.2 and 8.3.1). Using common selections schemes (see section 3.4.1 and 8.3.2), this step can only be performed, once the fitness values of all individuals of the preceeding generation have been determined (the parent selection process is currently implemented purely in software). In contrast, the evaluation of the newly generated candidate solutions can in principle commence as soon as the evolutionary coprocessor has created the first offspring.

unparallelized coprocessor operation

However, in the setup that is used for all experiments presented in the previous chapters, the operation of the evolutionary coprocessor and the execution of networks on the HAGEN chip are not yet performed in parallel. In other words, the testing of individuals on the hardware and the parallel calculation of their fitness values is not initiated before the coprocessor has completely finished executing (see figure 10.1). But as it will be shown below, this does not lead to a significant loss of training speed in practice.

10.1.1 Time Measurements

Within the present hardware framework, performing the fitness calculation in software requires to transfer the output data of each tested network from the local memory of the FPGA to the RAM of the host PC over the PCI bus (see section 6.1.3 and compare figure 10.1). Being currently executed on a Pentium IV with 2.4 GHz, the actual evaluation of the fitness can then be performed comparably fast. Considering the exemplary case of the liver disorder problem where a number of 1550 patterns needs to be processed by both, the tested network and the fitness function, the latter requires 0.28 ± 0.01 ms, while the execution of the network on the hardware — including the transfer of the output data over the PCI bus — takes 2.04 ± 0.01 ms. If the time-consuming exchange of output data between the memory of the FPGA and the host computer is omitted, this time is reduced to 1.26 ± 0.01 ms[1].

liver disorder

The reported numbers are obtained by measuring the amount of time that is spent in the `HNetMan::run(·)` function and the fitness calculation method[2] (see section 7.4.1), respectively. The given values each represent the mean of 100 measurements and the respective standard deviation. Similar to all previously presented experiments, the interface of the HAGEN chip is operated at a frequency of 84 MHz (see section 6.1.3) and the network frequency f_{net} (see section 5.2) is thus determined to be 14 MHz [181] [185]. The first phase of the first stage of the proposed stepwise strategy is considered (see sections 9.1 and 9.2). The trained network comprises $N^{\text{c}}_{\text{hid}} = 6$ hidden units and $N^{\text{c}}_{\text{out}} = 4$ outputs.

measurement procedure

It is to be noted that in the case of the `HNetMan::run(·)` method, the reported time does not only include the network operation on the hardware: Even if the output data is not read back over the PCI bus, the measured value persists to be increased by the remaining communication between the PC and the FPGA that is necessary to synchronize their operation. Given the used base frequency of 84 MHz and the processed number of patterns of 1550, the time that is exclusively spent for the actual execution of the network can be estimated to be approximately 1.1 ms [181].

communication overhead

It remains that in the present state of the system, the calculation of the fitness only occupies about one fourth of the time that effectively elapses until the results of the next network can be read back from the memory of the FPGA. Since the times for the fitness evaluation and the network execution each grow linearly with the number of processed patterns, similar conditions are expected to apply also to the other investigated tasks (see below). This generally leads to a considerable amount of idle time for the CPU, and a strategy of how this time can be used more efficiently will be introduced in section 10.1.2.

remaining idle time

[1] In the current setup, such a procedure is evidently not practical, since the fitness calculation can in this case not yield reasonable results.

[2] For these measurements, the network execution and the fitness evaluation are performed sequentially and not in parallel. The standard built-in C-functionality for time measurements is used as it is provided by the `ctime` library.

Averaged Individual Processing Time

experimental setup

In order convey a general feeling for the speed of the current training setup, measurements are performed where the average overall processing time for one individual network is estimated. This is done for all of the nine investigated benchmarks. Again, all networks contain $N^{\mathrm{c}}_{\mathrm{hid}} = 6$ hidden units and $N^{\mathrm{c}}_{\mathrm{out}} = 4$ outputs, and the effective number of adjustable genes within the genome merely varies according to the number of inputs required for the respective benchmarks. The processing of the genetic material is taken over by the evolutionary coprocessor. Apart from that, the evaluation time of each network depends on the respective number of input patterns that needs to be processed.

averaged processing time per individual

For each task, it is evaluated how much time is spent in one generation step of the evolutionary algorithm, i.e., the time that elapses between the moment when the `HPopulation::newGeneration(·)` function is invoked and the point when the succeeding `HPopulation::evaluate()` method returns (compare figure 7.21 and see sections 7.4.3, 7.4.4, and 7.4.5). Similar to the experiments presented in the previous chapters, the population size is set to $\mu = 20$ and the average time for one individual is thus obtained by dividing the resulting numbers by 20. The measurements are repeated for a respective number of 1000 generations and the corresponding average is taken as the final result.

a complete iteration

According to the scheme depicted in figure 10.1, the fitness calculation is performed in parallel to the network execution. Still, apart from the actual implementation of the networks on the HAGEN ASIC, the operation of the evolutionary coprocessor, and the necessary communication via the PCI bus (see above), the measured times also include the sorting of the population according to the individuals' fitness values, the selection process, the generation of the desired recombination instructions for the coprocessor (see section 7.3.3), and the general organizational overhead for the parallelized execution of networks on the hardware (see section 7.3.2). The latter four parts are all implemented in software.

In so far, it is to be emphasized that the presented numbers do not claim to be an accurate estimate for the actual speed of the hardware itself. Rather, they are to serve as a practical measure for the speed of the training as it is experienced by the user. A second set of measurement evaluates the time that is exclusively spent in the `HPopulation::newGeneration(·)` method. These times remain to include the selection process, the generation of appropriate instructions for the evolutionary coprocessor in software, the transmission of these commands over the PCI bus, and the operation of the coprocessor itself. Again, the resulting numbers are normalized by the number of individuals in the population.

creating the new generation

results

The obtained values are presented in figure 10.2. The results are given in milliseconds and are shown as a function of the number of processed patterns. The overall times per individual for one complete generation step are represented by circular markers, the data points that correspond to only the creation of the next population are shown as squares.

linear dependence

The averaged overall time per individual shows the expected dependence on the number of processed patterns: The times for both, the execution of the actual network and the calculation of its fitness should grow linearly with the number

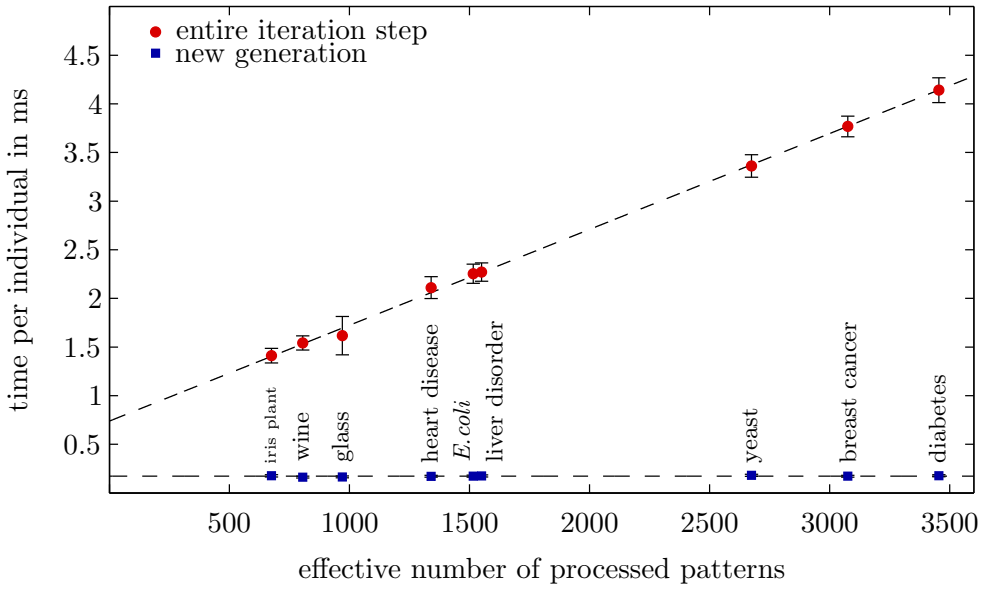


Figure 10.2: *The averaged overall processing time per individual during evolutionary training as a function of the effective number of patterns that are processed by the single networks. The circular markers refer to the total time that is required for one generation step. As expected, this time shows a linear dependence on the number of processed patterns. The squares correspond to the time that is used to merely create the next generation of individuals. This value depends only on the size of the genome. Similar to the first phase of the fist stage of the proposed stepwise training strategy (see chapter 9), the trained networks all contain* $N_{\text{hid}}^{\text{sn}} = 6$ *hidden neurons and* $N_{\text{out}}^{\text{sn}} = 4$ *outputs, and the respective genomes differ only marginally in size (see text).*

of input patterns. Indeed, within the remaining statistical error, the measured points very accurately form a straight line. The observed linear behavior can be extrapolated to the case where no input pattern is applied at all. The execution of a network then only incorporates the implementation of the corresponding network configuration on the HAGEN ASIC. The estimated intersection point of the extrapolated line[3] and the ordinate axis yields a value of $0.74 \pm 0.03\,\text{ms}$. *constant offset*

The effective time per individual that is lost to the creation of a new generation from the previous population is independent of the number of patterns that need to be processed by the single individuals. Any remaining differences between the examined benchmark problems arise due to the varying numbers of inputs to the respective network and the hereby determined numbers of adjustable genes in the genome. However, the observed variations in processing time are only marginal. On average, the time per individual that is spent in the `HPopulation::newGeneration(·)` method is obtained to be $0.17 \pm 0.01\,\text{ms}$.

[3]The fit is a simple linear regression performed in MATLAB [134]

Outlook: Speeding up the Training

current limitations

Especially for large problems it can be inferred from figure 10.2 that the training speed is presently limited by the the time that is required for the network execution. But it is to be emphasized that while the evaluation time for each network will persist to grow linearly with the number of applied input patterns, the absolute value of the execution time as well as the slope of the observed linear increase are specific to the currently used hardware environment which cannot in fact fully exploit the potential of the HAGEN ASIC.

advanced setup

It has already been stated in section 6.3 that within the new NATHAN/Backplane setup, the interface to the HAGEN chip can be operated at higher frequencies and no data transfer over the PCI bus is required during training. As soon as this more advanced setup is fully operable, it will serve as the primary platform for future applications and training experiments. Regarding the linear dependence of the averaged overall processing time per individual on the number of input patterns (see figure 10.2), both, its slope and its offset will then be considerably decreased. At the same time, given the frequency of the featured PowerPC microprocessor of only 350 MHz (see section 6.3.3), all parts of the iteration step that are implemented in software — most notably the fitness calculation and the selection process — will be slowed down significantly. Under these circumstances, the roles of the fitness calculation and the network execution are likely to be exchanged in so far as the former will require more time than the latter.

future ASICs

Finally, it is to be noted that the way in which incoming and outgoing data is managed within the HAGEN prototype leaves some room for improvement [178]. Future neural network ASICs will provide additional functionality that will allow for a more efficient, pipelined handling of input and output data directly on the chip. This is bound to speed up the execution of the networks even further.

fast algorithms

At present, exact quantitative predictions for the eventual training speed of the NATHAN/Backplane setup are difficult [181] and do not actually fall into the scope of this work. But as it has already been discussed in sections 6.2.6 and 6.3.3, the above considerations remain to fuel the desire for simple evolutionary algorithms that can keep up pace with the used neural network ASICs and can preferably benefit from hardware acceleration and/or parallelization themselves.

10.1.2 Parallelization of the Generalized Stepwise Strategy

unexploited potential

During evolutionary training, the calculation of the former individual's fitness value currently takes only about one fourth of the time that is required for the next candidate network to finish executing on the hardware (see also section 10.1.1 and compare figure 10.1). The remaining intermediate CPU time is bound to be wasted. It is true that due to the high degree of parallelism in the network operation of the HAGEN prototype, the execution time for, e.g., a two-layer network does not increase with a growing size of its single layers. This allows for a feasible scaling to large networks. But for comparably small networks like those that are trained during the single phases of the proposed stepwise strategy (see chapter 9) and which occupy merely a small fraction of the offered network resources each,

this parallelism is only insufficiently exploited.

On the other hand, it has repeatedly been claimed that the stepwise approach itself readily promotes a viable parallelization of the training process. The following sections will describe how this parallelism can efficiently be exploited already within the currently used hardware environment that merely features one single network ASIC per setup.

Simultaneous Evaluation of Candidate Solutions

Figure 10.3 presents a schematical illustration of the upper left corner of the upper left network block of a HAGEN ASIC (see section 5.4). The image is taken from a screenshot of the network visualization tool that is included in the HANNEE software (see chapter 7). The black and white squares at the top and the left side represent the single input nodes and output neurons respectively. The shown matrix of colored cells illustrates the synaptic array of the network block (compare figures 5.2 and 5.5). The red, blue and white cells correspond to positive, negative, and zero weights, respectively. The orange or grey color of the buttons at the edges of the array indicate whether any of the feedback or inter-block connections of the corresponding input or output node is activated (see section 5.4.2).

exemplary network

The shown configuration represents a network that has been trained for the iris plant problem using the stepwise training strategy proposed in chapter 9. One subnetwork is included for each of the three classes. In pursuit of a better visualization, this exemplary network is implemented on only one single block instead of being distributed over two blocks (see section 8.2.4). A subset of the available feedback connections is utilized to realize the two desired network layers. The network is shown in a state where the first training stage is completed but the second stage is not yet initiated, i.e., the additional interconnections between the subnetworks are not yet established (an illustration of the complete network can be found in figure C.14 in the appendix). The input nodes and output neurons that are actually used for the implemented network are marked by black squares.

subnetworks: independent operation

By construction, the single subnetworks occupy distinct areas of the synaptic array and are entirely independent. It is an important consequence that during training, three different subnetworks can in principle be tested simultaneously during one single operation of the HAGEN ASIC: one for each of the three different classes. Similar considerations also apply to the interconnections that are introduced in the second stage. It has already been discussed in sections 9.1.1 and 9.3.2 that the respective synaptic connections which lead to the output neurons of one particular subnetwork can be optimized independently of those that lead to all other outputs.

In general, given a total desired number of subnetworks $N_{\text{net}} = N_{\text{c}} \cdot N_{\text{net}}^{\text{c}}$ in the final network, N_{net} corresponding candidate solutions can be evaluated on the network chip simultaneously. Due to the parallel operation of the HAGEN chip, this does not lead to an increase in the required execution time.

parallelization

Within each of the two training stages, the single phases of the proposed stepwise approach can therefore partly be parallelized.This is schematically illustrated in figure 10.4. For the parallel training of $N_{\text{net}}^{\text{p}}$ subnetworks, $N_{\text{net}}^{\text{p}}$ corresponding

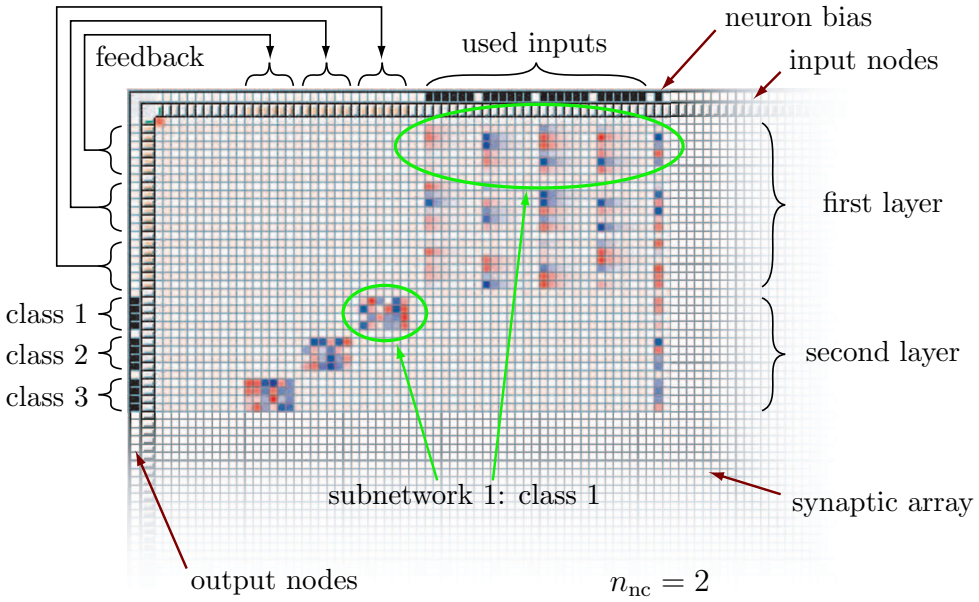


Figure 10.3: *Schematical illustration of the upper left corner of the upper left block of a HAGEN ASIC, taken from a screenshot of the HANNEE software (see chapter 7). The input nodes are on top, the neurons are to the left, and the array of colored squares represents the synaptic array (compare also figures 5.2, 5.5, and 8.2 b)). Positive weight values are a shade of red, negative weights are blue, and white squares correspond to deactivated synapses with zero weight. The presented network has a two-layer architecture and is trained for the iris problem. It contains one subnetwork for each class and is shown in a state where the first stage of the stepwise training is completed but the second stage is not yet initiated: the single subnetworks are still unconnected (compare figure C.14).*

populations of candidate solutions are processed partly simultaneously. In each iteration, $N_{\text{net}}^{\text{p}}$ new generations are created from the $N_{\text{net}}^{\text{p}}$ respective previous populations. The populations are entirely independent, and the generation step can thus in principle be parallelized as well. But since the selection scheme is implemented in software and each hardware setup only features one single evolutionary coprocessor, the different populations are at present processed sequentially. However, given the results that have been presented in figure 10.2, this does not breed a significant loss in training speed — at least within the currently used hardware environment.

sequential creation

During the evaluation phase, $N_{\text{net}}^{\text{p}}$ candidate subnetworks — one from each of the $N_{\text{net}}^{\text{p}}$ populations — are tested simultaneously during one execution of the network ASIC. More specifically, based on the respective genetic data that is stored in the local RAM of the FPGA (see section 6.2) one single network configuration is constructed that determines the required feedback and inter-block connections and combines the weight values for all $N_{\text{net}}^{\text{p}}$ candidate solutions. This configuration is sent to the hardware for implementation via one single call to `HNetMan::run(·)` (see section 7.3.2).

parallelized evaluation

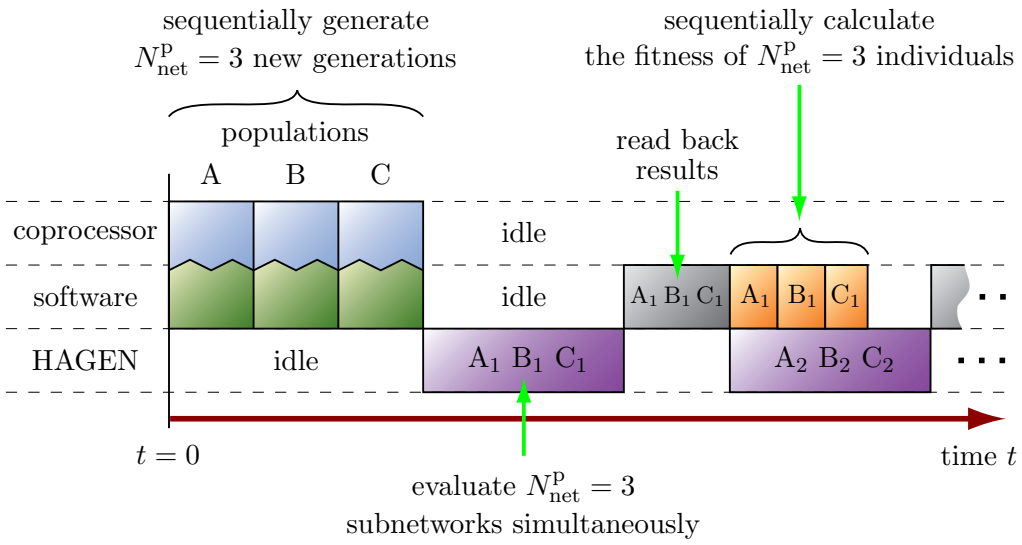


Figure 10.4: *Schematical illustration of the proposed parallelization scheme for the stepwise training approach (see chapter 9 and compare figure 10.1). In the shown example, $N^{\mathrm{p}}_{\mathrm{net}} = 3$ subnetworks are trained in parallel, i.e., three independent populations — one for each of the three subnetworks — are processed partly simultaneously. In each iteration step, the three corresponding next generations are created sequentially. In contrast, three subnetworks — one from each population — can be tested in parallel during one single execution of the hardware (compare figure 10.3 and see text). While the next three candidate solutions are evaluated, the fitness values of the former three networks are already calculated in software. This needs to be done sequentially but can efficiently exploit the CPU time that is available during the network operation. Since the creation of the three new generations can be achieved comparably fast (see section 10.1.1) and the time for one network execution is independent of the amount of used network resources, this parallelization procedure yields a considerable increase in training speed (see figure 10.5).*

sequential fitness calculation

Once the resulting network output is available, the fitness values of the $N^{\mathrm{p}}_{\mathrm{net}}$ candidate solutions are calculated sequentially in software, while the next $N^{\mathrm{p}}_{\mathrm{net}}$ subnetworks are already tested on the hardware in parallel. This way, the considerable amount of CPU time that is currently available during the hardware operation (see section 10.1.1) can be used more efficiently (see figure 10.4): For up to approximately $N^{\mathrm{p}}_{\mathrm{net}} = 4$, the time that is required for the fitness calculations can completely be hidden in the network execution time.

Time Measurements

evaluation procedure

For the exemplary case of the liver disorder benchmark, figure 10.5 shows the averaged overall processing time per individual as a function of the number of subnetworks that are trained in parallel. Similar to the measurements presented in section 10.1.1, the subnetworks each contain $N^{\mathrm{c}}_{\mathrm{hid}} = 6$ hidden units and $N^{\mathrm{c}}_{\mathrm{out}} = 4$ outputs and the processed populations include $\mu = 20$ individuals. Again, it is measured, how much time passes from the invocation of the `HPopulation::newGeneration(·)` method to the moment when the `HPopulation::evaluate()`

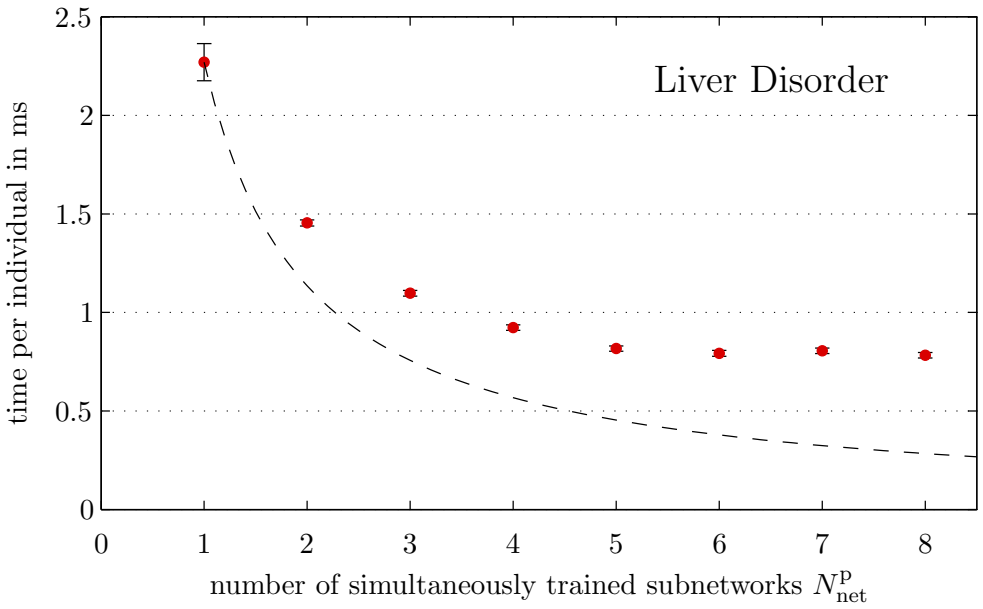


Figure 10.5: *For the liver disorder benchmark, the averaged overall processing time per individual during evolutionary training is shown as a function of the number of subnetworks $N^{\mathrm{p}}_{\mathrm{net}}$ that are trained simultaneously according to the proposed parallelization scheme (compare also figure 10.4). Again, the single subnetworks contain $N^{\mathrm{sn}}_{\mathrm{hid}} = 6$ hidden neurons and $N^{\mathrm{sn}}_{\mathrm{out}} = 4$ outputs each. The reported time for $N^{\mathrm{p}}_{\mathrm{net}} = 1$ corresponds to the value already shown in figure 10.2. The dashed line refers to an optimal $1/n$-behavior. Since several parts of the evolutionary training procedure remain to be performed sequentially, the achieved gain in training speed is not as high as is expected in the ideal case.*

function returns. If $N^{\mathrm{p}}_{\mathrm{net}}$ subnetworks are evolved simultaneously, the considered population in fact contains $N^{\mathrm{p}}_{\mathrm{net}}$ independent subpopulations and each iteration step effectively processes $20 \cdot N^{\mathrm{p}}_{\mathrm{net}}$ individuals[4]. Accordingly, the measured times are normalized by the respective value $20 \cdot N^{\mathrm{p}}_{\mathrm{net}}$. The shown data points each represent the average over 1000 generations, and the reported time for $N^{\mathrm{p}}_{\mathrm{net}} = 1$ corresponds to the value that has also been included in figure 10.2.

measurable acceleration

The dashed line illustrates an ideal $1/n$-reduction in processing time. Since several parts of the iteration step — i.e., the creation of the $N^{\mathrm{p}}_{\mathrm{net}}$ new generations and the calculation of their fitness values — remain to be performed sequentially and given the fact that the available CPU time during one network execution effectively suffices for only about four fitness calculations, the eventually achieved gain in training speed is bound to be smaller than the ideally expected factor $N^{\mathrm{p}}_{\mathrm{net}}$. Still, it can be observed that following the above parallelization procedure, increasing the number of simultaneously evolved subnetworks $N^{\mathrm{p}}_{\mathrm{net}}$ yields a signifi-

[4]The multiple subpopulations that each correspond to a different subnetwork are all encapsulated by one dedicated `HPopulation` subclass. As it has been discussed in section 7.4.5, this allows to leave the algorithm implementation unmodified on a higher level (compare also figure 7.21).

cant reduction in the effective processing time that is on average required for each individual. For $N^{\mathrm{p}}_{\mathrm{net}} = 6$, the training time is reduced by approximately a factor of 3. Training more than 6 subnetworks in parallel only yields minor improvements in training speed.

Parallelization in Practice

remaining limitations

Leaving hardware limitations aside, the maximum number of subnetworks that can actually be trained simultaneously is ultimately limited by the given number of classes in the examined categorization problem times the desired number of subnetworks per class. Apart from that, it is understood that the proposed scheme does not exploit the inherent parallelism of the stepwise training strategy to the fullest possible extent: Several parts of the iteration step remain to be performed sequentially that could in principle be parallelized as well. But on the other hand, the presented measurements do not only provide a first demonstration for the claim that the stepwise training approach is indeed suited for a parallel implementation, the proposed scheme can also readily be employed within the currently used hardware setup. In fact, the exhaustive investigations presented in the foregoing chapter have successfully utilized the proposed parallelized implementation of the training process.

parallelization in future setups

For future investigations, it is planned to also perform the execution of the evolutionary coprocessor in parallel to the evaluation of networks on the HAGEN ASIC. With regard to the upcoming NATHAN/Backplane framework (see section 6.3), the faster operational speed of the network hardware and the reduced performance of the featured PowerPC microprocessor will render the proposed parallelization scheme infeasible, since the fitness calculation will under these conditions most likely take longer than the actual network execution (see also section 6.3.3). On the other hand, the advanced hardware setup will allow to efficiently benefit from the parallel nature of the stepwise training strategy by distributing both, the final networks[5] and the simultaneous training of multiple subnetworks over several NATHAN boards (see also section 6.3.3).

10.2 Network Transferability

substrate deficiencies

It is one of the important advantages of chip-in-the-loop training that it can automatically cope with the potential deficiencies and peculiarities of the used hardware neural network substrate (see section 2.4.5). Beyond that, it has been discussed in section 5.5 that the device variations which affect the network operation on the HAGEN prototype can to a large extend be compensated for by an appropriate calibration. During the experiments presented in the preceeding sections, the used HAGEN chips have always been operated in calibrated mode such that the single neuron offsets and the row-specific averages of the individual synapse offsets are accounted for (see sections 5.5.1 and 5.5.4). Under these conditions, it remains the primary task of the evolutionary training algorithm to deal

[5] It has already been discussed in section 9.4.2 that at least for single-layer architectures, the trained networks can be distributed over an arbitrary number of blocks without restrictions.

with the single static synapse offsets and to appropriately adjust the individual weights as to best exploit the dynamic ranges of the neurons and synapses (see also section 5.4.3).

network transfer

Given the allowed maximum postsynaptic current $\mathrm{I}_{\mathrm{ps}}^{\max} = 22\,\mu\mathrm{A}$ (section 5.4.3) that is used for all presented experiments, the magnitudes of the individual synapse offsets are in the order of only 1 % of the total weight range (see also section 5.5.3). In so far — as it has already been discussed in section 5.5.4 — any network configurations that are obtained during the training of networks on one HAGEN ASIC should to some extent also be usable on other HAGEN chips. It is not expected, however, that the transfer of a network configuration to another chip will yield exactly the same performance as on the original ASIC it has been trained on. Therefore, it is worthwhile to investigate if and in how far the performance of a network configuration that has been trained with the generalized stepwise strategy deteriorates once it is being loaded into a chip that it has not originally been optimized for.

10.2.1 Transfer Experiments

For the liver disorder, wine and *E.coli* benchmarks, the different network configurations that have been obtained during the experiments presented in the preceeding chapter are each transferred to two different HAGEN ASICs. This is done for all investigated numbers of networks per class $N_{\mathrm{net}}^{\mathrm{c}}$ and for both, single-layer and two-layer networks.

absolute differences

In this context, it is to be repeated that the classification accuracies on the training and test sets A_{t} and A_{g} reported in the preceeding chapters each represent the respective average over a total of 50 different networks — five for each of the ten stratified random partitionings of the data into training and test set (see section 8.1.2). Based on the original accuracies a_{t}^r and a_{g}^r of a specific network configuration that has been trained on the rth partitioning and given the respective values $\widetilde{a}_{\mathrm{t}}^r$ and $\widetilde{a}_{\mathrm{g}}^r$ that are obtained on the same training and test data once this very network is implemented on another chip, consider the absolute differences

$$d_{\mathrm{t}}^r = |a_{\mathrm{t}}^r - \widetilde{a}_{\mathrm{t}}^r| \tag{10.1}$$

$$d_{\mathrm{g}}^r = |a_{\mathrm{g}}^r - \widetilde{a}_{\mathrm{g}}^r| \,. \tag{10.2}$$

Since d_{t}^r and d_{g}^r quantify the respective difference in the fraction of correctly classified training and test patterns, they are given in percentage points. After measuring these quantities for all of the ten networks that have originally been considered for one single 10-fold cross-validation, the resulting values are averaged to yield the mean differences $\overline{d}_{\mathrm{t}}$ and $\overline{d}_{\mathrm{g}}$. This is repeated for each of the five performed repetitions of the cross-validation measurement (see section 8.1.2), and the achieved values are averaged once more to yield the final results D_{t} and D_{g} together with the corresponding standard errors of the mean ΔD_{t} and ΔD_{g}.

averaged results

liver disorder

The outcome of this measurement for the liver disorder benchmark is presented in figure 10.6. The networks are originally trained on the HAGEN ASIC denoted as “chip A” and are once transferred to “chip B” (left plot) and once to “chip C” (right plot). The three chips are operated in calibrated mode, work at the same

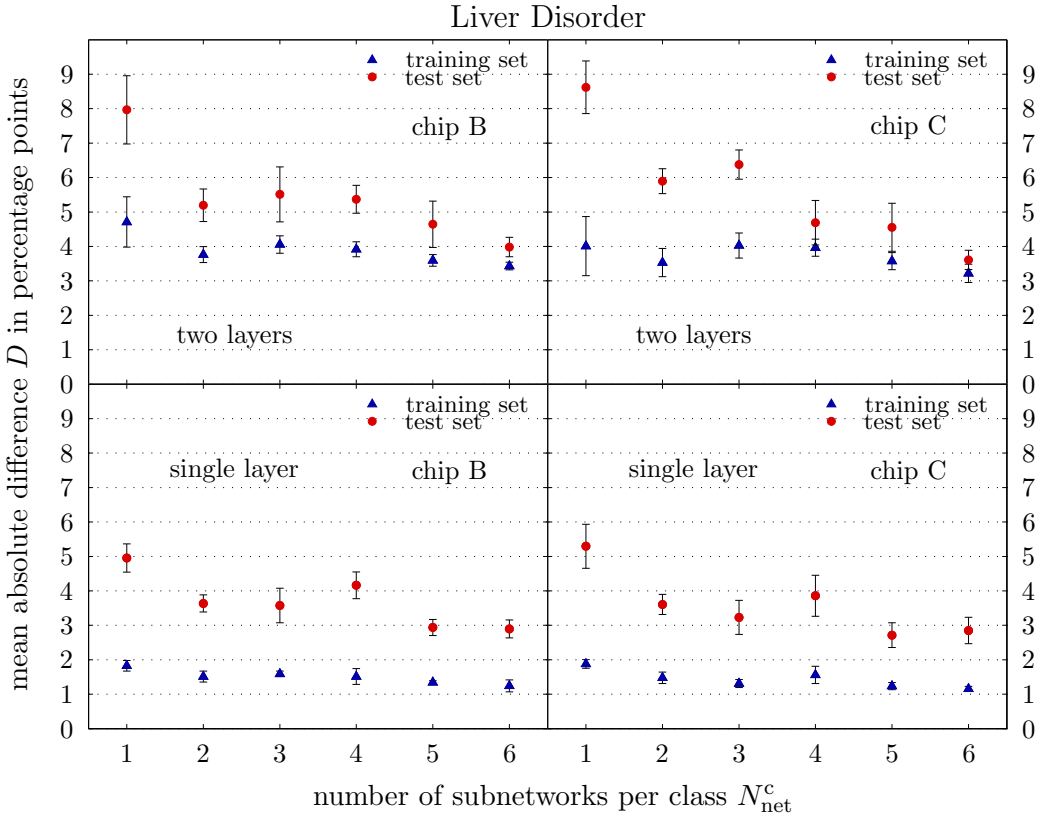


Figure 10.6: *The networks that have been trained for the liver disorder benchmark during the experiments presented in section 9.4 are loaded into different chips. Originally, the networks have been trained on the HAGEN ASIC denoted as "chip A" (not shown). Now, they are transferred to"chip B" (left plot) and "chip C" (right plot). The mean absolute differences in classification accuracy on the training and test sets between the original and the transferred networks (see also equation 10.2) are shown as a function of the number of subnetworks per class N^{c}_{net}. It can be observed that on average, increasing the number of subnetworks also improves the stability of a network's response when it is being transferred to another chip.*

frequency of 84 MHz (section 6.1.3), and use the same analog settings, most notably, a maximum synaptic current of $I^{max}_{ps} = 22\,\mu A$ (section 5.4.3). The obtained averaged absolute differences D_t and D_g are shown as a function of the number of subnetworks that have been trained for each class (see section 9.3.1). The upper plot refers to two-layer networks, the lower plot represents the single-layer networks.

As expected, the responses of the transferred networks deviate from the results that are obtained on the chip they have originally been optimized for. The observed differences in classification accuracy are generally larger on the test sets than on the training sets. This is plausible since on the latter, the networks' sensitivity to temporal fluctuations is automatically minimized during the employed chip-in-the-loop training procedure. With the analog noise being in the same order

training set

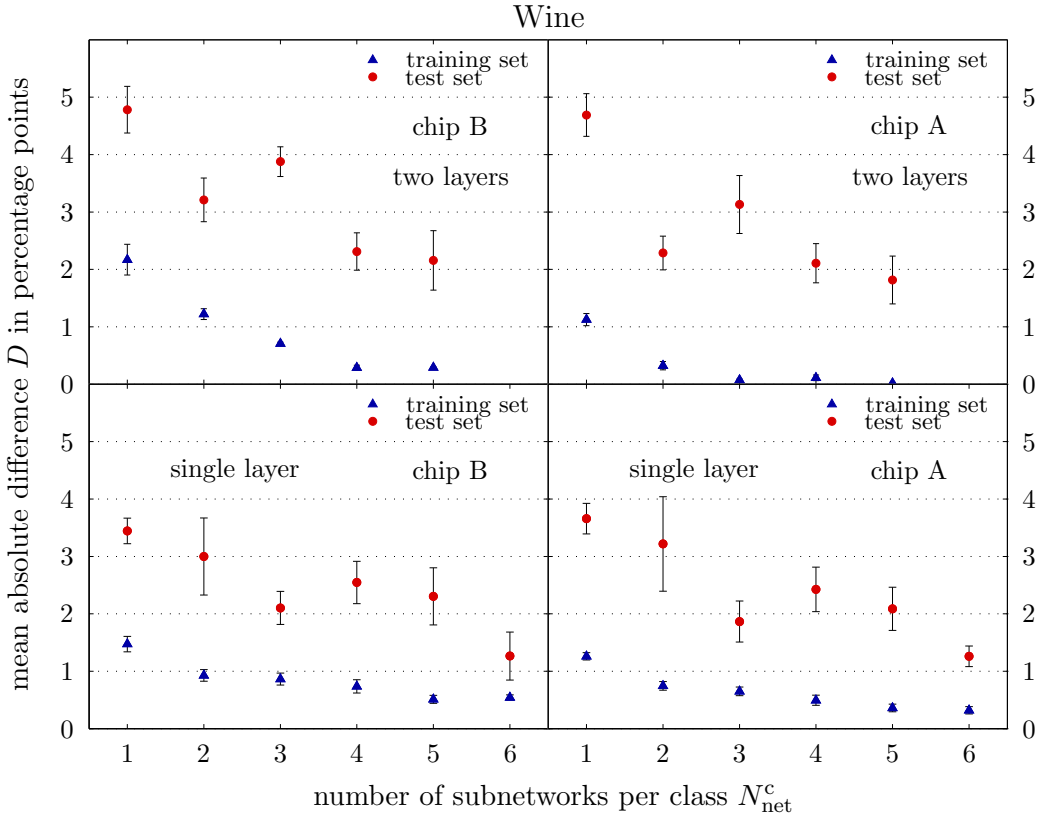


Figure 10.7: *The networks that have been trained for the wine benchmark during the experiments presented in section 9.4 are loaded into different chips. Originally, the networks have been trained on "chip C" (not shown). Now, they are transferred to "chip B" (left plot) and "chip A" (right plot). The resulting mean absolute differences in classification accuracy on the training and test sets (see also equation 10.2) are shown as a function of the number of subnetworks per class N^c_{net}. The results are similar to those obtained with the liver disorder benchmark (see figure 10.6). For two-layer networks that are transferred to "chip A", the averaged absolute differences in classification accuracy on the training data are even reduced to zero.*

of magnitude than the static variations (see section 5.5.3) the improved insensitivity to temporal fluctuations also gives rise to an increased robustness against differences in the static synapse offsets like they occur between different chips. In contrast, for input patterns from the test set, the insensitivity to analog noise is not optimized during training (see also section 8.4.1), and the networks' responses therefore show an increased susceptibility to changes in the static synapse offsets as well.

single-layer networks

Single-layer networks suffer from a measurably lower deterioration in performance than two-layer networks when being transferred to another chip. This might partly be accredited to the circumstance that the former include considerably fewer synapses than the latter. Furthermore, in a network with two layers, the single synaptic connections exhibit much stronger interdependencies as in a single-

layer network. Due to the highly nonlinear behavior of the employed threshold neurons, only small variations of the weights in the input layer can lead to significant changes in the responses of the hidden nodes and are thus likely to cause even larger deviations in the final output of the second layer. Finally, it can be inferred from figure 9.3 that single-layer networks generally perform worse on this task than two-layer networks in the first place. With a linear partitioning of the input space obviously not being the appropriate separation to reliably differentiate between the two classes, it is reasonable to assume that multiple linear partitionings exist which yield approximately the same —suboptimal— classification performance. In so far, it seems plausible that the eventual classification accuracies of networks with only one layer are less sensitive to perturbations of the single weight values than those of two-layer networks.

multiple subnetworks: improved stability

Apart from that, it is an important observation that with an increasing number of subnetworks per class, the averaged absolute difference in accuracy on the test set D_{g} is significantly reduced. This also applies to the difference in performance D_{t} that is obtained on the training data but here, the effect is not as distinct.

wine and E.coli

Similar results are obtained with the wine and *E.coli* data sets, as it is shown in figures 10.7 and 10.8. The networks for these two benchmarks have been trained on "chip C" and are transferred to both, "chip A" and "chip B". Compared to the liver disorder data set, the initial differences between the original and the transferred networks are smaller, and the observed decrease in D_{t} is more pronounced. In the case of the wine data set, the average difference in performance on the training data observed for two-layer networks that are transferred to "chip A" is even reduced to zero.

possible explanation

Initially, it could have been expected that with an increasing size, the networks' susceptibility to variations in the static offsets becomes more severe, since a larger number of mutually dependent synaptic weights is affected. But it can be inferred from the presented measurements that the redundancy that results from the training of multiple subnetworks per class actually gives rise to an improved stability of the final networks once they are confronted with the peculiarities of another chip than the one they have been trained on. It might be a possible explanation for this phenomenon that the deviations which occur when a network is transferred to a different chip promote an improved diversity among the single subnetworks that are dedicated to the same class. It has been argued in section 9.3.2 that an increased dissimilarity between the members of a network ensemble yields a better performance of the whole committee. It is reasonable to assume that this beneficial effect can at least partly compensate for the perturbations that inevitably occur when the network is transfered to another HAGEN ASIC.

saturation behavior

The results that are achieved on the *E.coli* data set reveal that the stability of the final networks is not improved any further when more than approximately 5 subnetworks per class are trained (see figure 10.8). This can be seen in analogy to the saturation behavior of the generalization accuracy that has already been observed during the actual training of an increasing number of subnetworks per class (compare figure 9.6).

remaining differences

For $N_{\mathrm{net}}^{\mathrm{c}} > 5$, the difference in accuracy on the training data even seems to increase again, but given the estimated statistical uncertainty, a definite conclusion

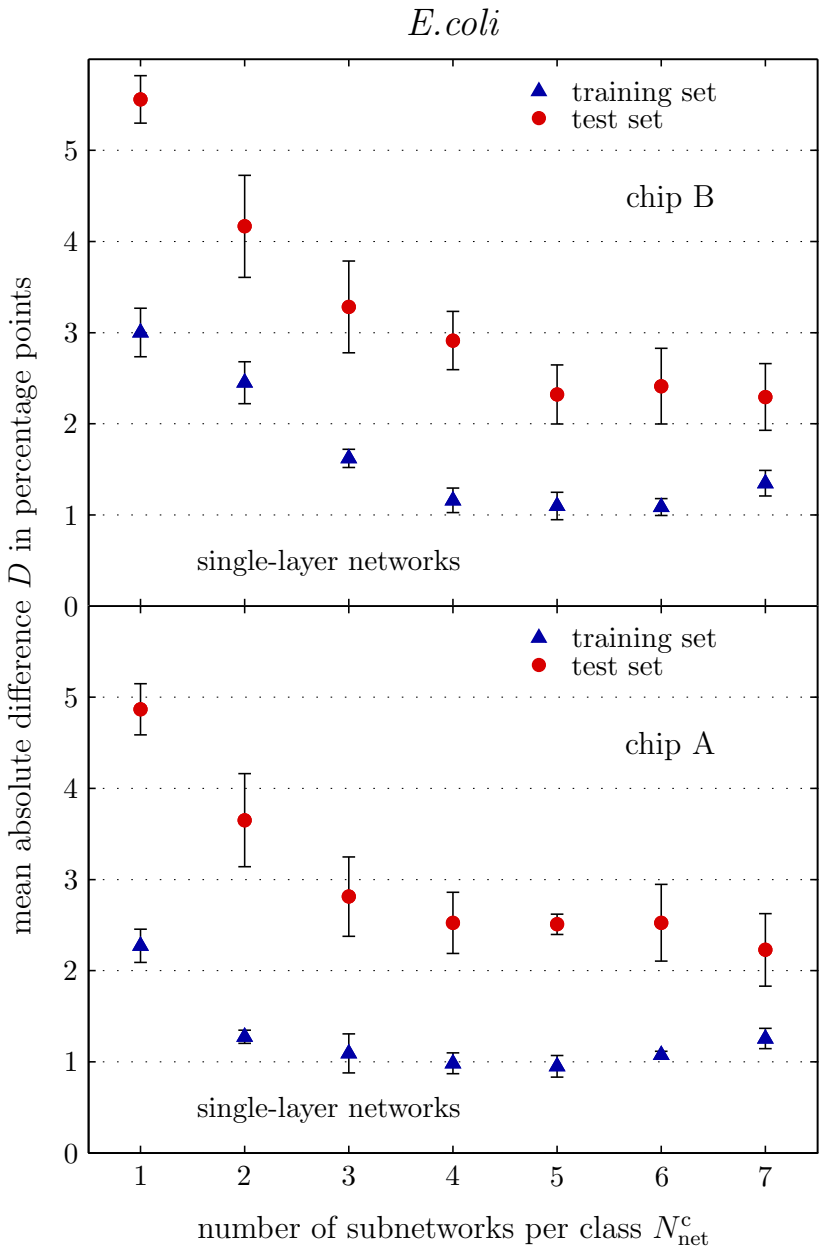


Figure 10.8: *The networks that have been trained for the* E.coli *benchmark during the experiments presented in section 9.4 are loaded into different chips. Originally, the networks have been trained on "chip C" (not shown). Now, they are transferred to"chip B" (left plot) and "chip A" (right plot). The resulting mean absolute differences in classification accuracy on the training and test sets (see also equation 10.2) are shown as a function of the number of subnetworks per class N^c_{net}. The results are similar to those obtained with the liver disorder and wine benchmarks (see figures 10.6 and 10.7). Increasing the number of subnetworks per class beyond $N^c_{net} = 5$ does not seem to further improve the stability of the networks' responses.*

cannot be drawn. Still, the measurements suggest that the differences in performance between the original and the transferred networks are not reduced to zero when more subnetworks per class are trained. It is to be expected that a given minimum deviation will remain.

Having hitherto considered only the absolute values of the observed deviations (see equation 10.2), it is to be noted that especially on the test data, the final classification accuracy of a transferred network can in some cases actually be better than the performance of the respective original. In practice, however, this effect cannot directly be exploited since the behavior of a network under chip transfer is effectively unpredictable. In so far, the mean absolute differences D_t and D_g persist to be the adequate measures for evaluating the stability of a network.

10.2.2 Discussion and Further Improvements

different ASICs

Regarding the measurements for each of the three benchmarks, the results that are achieved on the two different respective target chips are very similar. It can thus reasonably be assumed that the obtained results are of general validity and are not caused by specific peculiarities of any of the three involved HAGEN ASICs.

improved stability

It is a satisfying observation that apart from producing networks with competitive generalization accuracies on realistic problems (see table 9.3), the proposed stepwise training strategy also promotes an improved stability of the final networks once they are being transferred to different chips. But it persists that the individual characteristics of the used ASICs remain to lead to measurable deviations in the network response. The perturbative effects that occur when a network configuration is loaded into another chip are not actually compensated for. Rather, their negative impact on the network's eventual classification accuracy is reduced.

further improvement

Some promising approaches to further improve the network transferability have been discussed in section 5.5.4. Most notably, it is expected that the original performance of a transferred network can be reproduced by a short and cautious retraining. However, corresponding investigations are beyond the scope of this work and have to be deferred to further investigations.

10.3 Outlook: Coping with Chip Limitations

current limitations

The limited size of the single network blocks on the HAGEN prototype (see section 5.4.1) as well as the fixed set of available, hard-wired feedback and inter-block connections (section 5.4.2) remain to impose certain restrictions on the size and architecture of the realizable networks. It has already been discussed in section 9.4.2 that in the case of the generalized stepwise strategy, this eventually defines an upper limit to the number of subnetworks that can be trained for each class. Single-layer networks are not as severely affected by these restrictions as homogeneously connected two-layer architectures since they do not require any feedback connections and can thus more easily be distributed over multiple network blocks.

smaller subnetworks

Having observed that an increased number of subnetworks per class is generally expected to lead to an improvement in the final network's performance (see section 9.4) and given the result that an alternative increase in the subnetwork size is not equally beneficial (section 9.4.6), it suggests itself to investigate the feasibility of smaller subnetworks that in turn allow for a larger number of subnetworks per class to simultaneously fit on one HAGEN chip. Although it is actually regarded as a favorable feature of the proposed generalized stepwise training strategy that networks with competitive performance can be produced even without any adjustment of the subnetwork size, the pragmatically chosen numbers of $N^{\mathrm{c}}_{\mathrm{hid}} = 6$ hidden neurons and/or $N^{\mathrm{c}}_{\mathrm{out}} = 4$ outputs per subnetwork are not necessarily optimal.

limited number of inputs

There is another limitation of the HAGEN prototype that arises from the fixed number of 128 inputs to each block. For classification problems with large numbers of input attributes, the resulting lengths of the binary input strings s^{α} might exceed the number of available input nodes. In such cases, it will be required to distribute the input layer over several network blocks. For a given neuron, it will thus no longer be possible to directly be connected to all inputs.

distributed input layers

As it has been stated in section 5.4.1, potential future network ASICs can partly overcome these restrictions by featuring more and/or larger network blocks. The employed circuit designs should allow for about 1000 input nodes per neuron, and the maximum number of outputs is solely limited by the desired size of the entire ASIC. Still, aiming for an application of the implemented networks to data intensive problems with high-dimensional input spaces — like, e.g., image processing tasks — the available number of inputs might nevertheless turn out to be insufficient. In so far, it is worthwhile to examine whether partially connected architectures, where the single neurons are not equally connected to all inputs, can retain a satisfying performance.

The following sections will present a set of initial measurements that aim to illuminate these questions. However, the reported experiments do not claim to be exhaustive. Rather, they intend to provide a first estimation of how an optimization of the subnetwork size or a distribution of the input nodes over multiple network blocks affect the performance of the final networks. As such, these measurements primarily serve as a basis for future investigations that are to address these topics more systematically.

10.3.1 Varied Subnetwork Size

varied subnetwork size

For the liver disorder benchmark, two additional sets of experiments are performed where the numbers $N^{\mathrm{sn}}_{\mathrm{hid}}$ and $N^{\mathrm{sn}}_{\mathrm{out}}$ of hidden neurons and outputs per subnetwork are modified compared to the setup that is used for the experiments presented in the foregoing chapter. For the first campaign of measurements, each subnetwork contains only $N^{\mathrm{sn}}_{\mathrm{hid}} = 3$ hidden nodes but persists to include $N^{\mathrm{sn}}_{\mathrm{out}} = 4$ outputs. In the second case, both, the numbers of hidden units and outputs are reduced and are set to $N^{\mathrm{sn}}_{\mathrm{hid}} = 3$ and $N^{\mathrm{sn}}_{\mathrm{out}} = 2$, respectively. The former set of measurements shall in the following be denoted as experiment M (medium) and the latter will be referred to as experiment S (small).

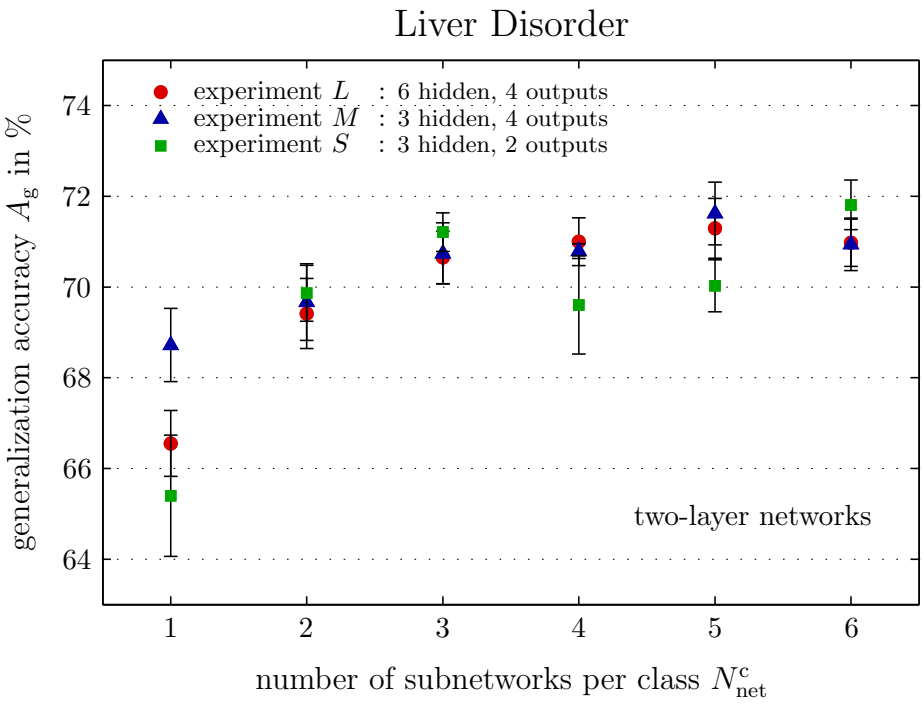


Figure 10.9: *For the liver disorder benchmark, the averaged classification accuracy on the test sets A_g (see section 8.1.2) is measured as a function of the number of subnetworks per class and for different subnetwork sizes. The results of experiment L ($N^{sn}_{hid} = 6$, $N^{sn}_{out} = 4$, circular markers) have already been presented in figure 9.3. Compared to these initial measurements, the subnetworks of experiment M possess fewer neurons in the hidden layer ($N^{sn}_{hid} = 3$, $N^{sn}_{out} = 4$, triangles), and those of experiment S actually contain only half the number of nodes ($N^{sn}_{hid} = 3$, $N^{sn}_{out} = 2$, squares). Training a sufficient number N^c_{net} of subnetworks per class, the eventual generalization rates of the final networks become approximately equal.*

measurement procedure

Apart from the modified subnetwork sizes, the employed experimental setup and training algorithm equal those already used for the investigations presented in section 9.4. For each phase of the stepwise training, a maximum of 1000 iterations are performed. The mean classification accuracy A_g of the final networks on the test set is measured as a function of the number of subnetworks per class N^c_{net}, and the results are shown in figure 10.9. For comparison, the generalization rates that have been obtained with the original subnetwork size are included as well (see figure 9.3). This set of measurements will henceforth be called experiment L (large).

different total sizes

Note that for a given number of subnetworks per class N^c_{net}, the three results that are obtained during the respective experiments L, M, and S are achieved by networks of different size. In fact, for the same value of N^c_{net}, the networks that are produced during experiment S are only half the size of the corresponding networks that are obtained in the original experiment L.

observations

For $N^{\mathrm{c}}_{\mathrm{net}} = 1$, the smaller networks of measurement S tend to perform worse than the corresponding networks of experiment L, while the medium sized networks of experiment M show a significantly better generalization rate. It is plausible that the decreased number of hidden nodes for networks in experiment M reduces the risk of overfitting, while the retained number of $N^{\mathrm{sn}}_{\mathrm{out}} = 4$ outputs per subnetwork persists to allow for a reasonably differentiated response. Furthermore, with the overall number of training iterations depending only on the number of subnetworks, the reduced dimensionality of the search space in each training phase might allow for a more effective optimization of the synaptic weights. On the other hand, although the smaller networks of experiment S feature hidden layers of the same size as those of experiment M, the decreased numbers of outputs are observed to measurably reduce the achievable generalization performance — despite the further reduced search space and the equal effective training time.

In so far, it is a remarkable observation that an increase in the number of subnetworks per class eventually yields approximately the same generalization performance for all investigated subnetwork sizes. At $N^{\mathrm{c}}_{\mathrm{net}} = 6$, the on average achieved classification accuracies on the test sets are approximately equal for all experiments.

independence of subnetwork size?

It has already been discussed in sections 9.4.3 and 9.4.4 that the training of multiple subnetworks per class can in some cases compensate for the initial differences in performance that are observed between single-layer and two-layer architectures. It seems that the results achieved with the generalized stepwise training strategy might to some extent also be insensitive to changes in the employed subnetwork size.

future investigations

Admittedly, the available experimental data does at this point not permit any definite conclusion. Similar experiments need to be performed also with the other investigated benchmarks in order to verify these initial observations. Apart from that, it still remains to be tested whether slightly enlarged subnetworks might even yield better results. Finally, since smaller subnetworks naturally allow for more of them to fit on one HAGEN ASIC simultaneously, it will be interesting to evaluate in how far this eventually permits a more efficient exploitation of the offered network resources.

10.3.2 Partitioned Input Layers

restricted connectivity

Within a further set of experiments, slightly modified networks are trained for the heart disease benchmark that feature only a reduced connectivity: The single hidden neurons of each subnetwork are not connected to all input nodes. Similar to the experiments presented in sections 9.2 and 9.4, the subnetworks contain $N^{\mathrm{sn}}_{\mathrm{hid}} = 6$ hidden units and $N^{\mathrm{sn}}_{\mathrm{out}} = 4$ outputs. But while the output layer persists to be homogeneously connected, half of the hidden nodes of each subnetwork are connected only to those binary input nodes that code the first 7 input attributes, while the other half is exclusively connected to those inputs that correspond to the remaining 6 input variables.

The hereby restricted connectivity is intended to simulate a situation where the number of available input attributes requires the input layer of the network to be

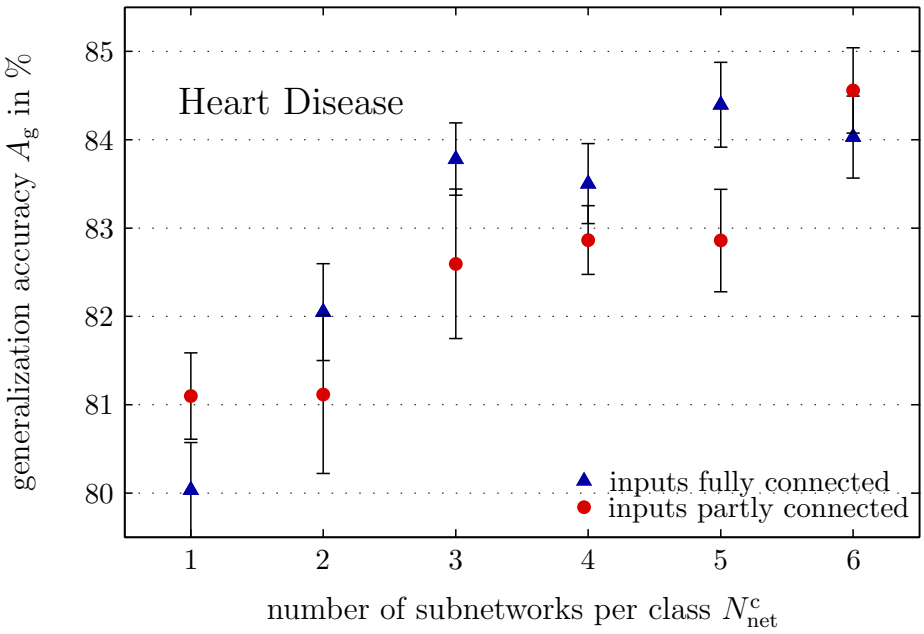


Figure 10.10: *For the heart disease benchmark, the generalized stepwise strategy is employed for the training of modified two-layer networks that feature only a partially connected input layer: Half of the hidden neurons of each subnetwork are exclusively connected to the input nodes of the first 7 attributes, while the other half is merely connected to those input nodes that correspond to the remaining 6 attributes. The averaged generalization rate A_g of the final networks (see section 8.1.2) is shown as a function of the number of subnetworks per class (circular markers). Compared to the original, fully connected networks (triangles, compare figure 9.9), the performance seems to be slightly decreased, but the differences are only marginal.*

distributed over multiple network blocks of the HAGEN ASIC. The heart disease problem is well suited for this kind of investigation since it features a comparably large number of 13 input attributes.

evaluation procedure

Once again, the networks are trained with the stepwise evolutionary strategy introduced in the preceeding chapter and the generalization accuracies of the final networks are measured as a function of the number of subnetworks per class. Apart from the missing connections in the input layer, the training setup is the same as for the experiments presented in section 9.4. The results are shown in figure 10.10 (circular markers). To allow for a better comparison, the corresponding classification rates on the test set that have been observed during the original experiments with a fully connected input layer are included as well (triangles, see also figure 9.9).

observations

For $N_{net}^c = 1$, the modified networks actually achieve a better generalization accuracy than those which are fully connected. It is conceivable that the restricted connectivity of the input layer reduces the risk of overfitting and thereby leads to the observed superior classification accuracy on the test set. Nevertheless, for $N_{net}^c > 1$, the networks with the incomplete input layers either show a worse per-

formance than their homogeneously connected pendants or the respective values are approximately equal within the estimated error.

marginal deterioration

All in all, the results seem to suggest that a reduced connectivity of the first layer indeed tends to slightly decrease the achievable performance of the final networks on this task. Still, given that the remaining statistical uncertainty ultimately impedes a clear decision, it is important to note that any potential deterioration in generalization accuracy is evidently only marginal. Furthermore, it can be inferred from figure 10.10 that even if the input layers needed to be distributed over several blocks, the hereby restricted networks would persist to measurably benefit from an independent training of multiple subnetworks per class.

Having investigated only a single exemplary benchmark problem, it cannot be predicted with certainty in how far these observations hold also for other classification tasks. But in the light of the presented measurements, it is reasonable to assume that the restricted number of input nodes of the featured network blocks does not necessarily prevent the HAGEN chip from being successfully applicable to problems with high-dimensional input spaces. Ultimately, this question will have to be clarified by future experiments.

10.4 Outlook: Software Simulations

The numerous networks that have been trained during all hitherto presented experiments are each optimized for the specific HAGEN ASIC they have been trained on. The particular deficiencies and peculiarities of the used chip are partly compensated for by the calibration procedure described in section 5.5.4, and the remaining deviations from the ideal network model (see section 5.2) are dealt with by the used chip-in-the-loop training algorithm. It has already been investigated in section 10.2, in how far this procedure allows for a viable transfer of the resulting network configurations to other calibrated HAGEN chips.

hardware networks in software

Apart from the static device variations and the present analog noise, the operation of the implemented network obeys equation 5.4. This in principle allows to use the obtained configurations also for software simulations of the utilized network model. Like in the case of a transfer from chip to chip, a simulation in software is not assumed to entirely preserve the network's performance, but the observations of section 10.2 are expected to hold also in this scenario.

training software networks

Furthermore, the applicability of the proposed stepwise training strategy is not *a priori* restricted to hardware implementations. It is true that the usage of multiple subnetworks per class naturally leads to larger networks. Software simulations that usually persist to be implemented on ordinary sequential computers rather call for preferably small systems, but this does not inhibit the utilization of the generalized stepwise approach for training in principle. On the other hand, it is conceivable that the demonstrated success of this strategy actually relies on the characteristics of the used hardware substrate. The inherent static variations and temporal fluctuations might promote the diversity of the multiple subnetworks that are trained for each class (see also sections 9.3.2 and 10.2.1). Finally, it will be interesting to evaluate the performance of network configurations that have

software networks in hardware

been optimized to work well in a software simulation and are transferred to a real HAGEN ASIC.

A detailed investigation of these aspects is not the purpose of this thesis. Nevertheless, in order to gain a first impression of to what extent the reported results are influenced by the characteristics of the used neural network hardware substrate, an initial set of experiments are conducted that employ a software simulation of the trained networks. Similar to the experiments presented in the foregoing section, these investigations primarily aim to provide an outlook to future work.

10.4.1 Hardware Networks in Software

stepwise training and software networks

In analogy to the measurements reported in section 10.2.1, the network configurations that have been obtained during the experiments discussed in section 9.4 for the liver disorder, wine, and *E.coli* problems are used to implement the corresponding networks in software. The employed software simulation is included in the HANNEE framework (see chapter 7) and implements the ideal network model defined by equation 5.4.

measurement procedure

Apart from the fact that the network configurations are not transferred to another HAGEN ASIC but are simulated on an ordinary CPU, the measurements are conducted in the same way as those of section 10.2.1. For the liver disorder benchmark, the resulting averaged absolute differences D_t and D_g together with the corresponding standard errors of the mean ΔD_t and ΔD_g are shown as a function of the number of subnetworks per class in figure 10.11 (left side). For comparison, the right half shows the outcome of the transfer to "chip B" that has already been presented in figure 10.6.

results

The shown data immediately reveals that the transfer to another chip on average has approximately the same effect as a transfer of the network configuration to a simulated ideal substrate. The corresponding measurements with the wine and *E.coli* data set yield similar result and are shown in figures C.15 and C.16 in the appendix. In all cases, the obtained differences are in remarkable agreement with those presented in section 10.2.1.

conclusion

This is an important observation since it provides an experimental proof for the claim that — apart from the known deviations — the used HAGEN ASICs actually implement the desired network model. If the trained networks exploited any further and hitherto unconsidered peculiarities of the used analog circuits, the transfer to an ideal substrate should yield measurably worse results than the transfer to another chip. In fact, it is observed that the deviations of each single ASIC from the ideal model (see section 5.2) are in the same order of magnitude as the differences between two calibrated chips. It can reasonably be concluded that the network operation of the HAGEN ASIC differs from the ideal model primarily by the remaining statistical device variations and not by any inherent deficiencies of the employed circuits.

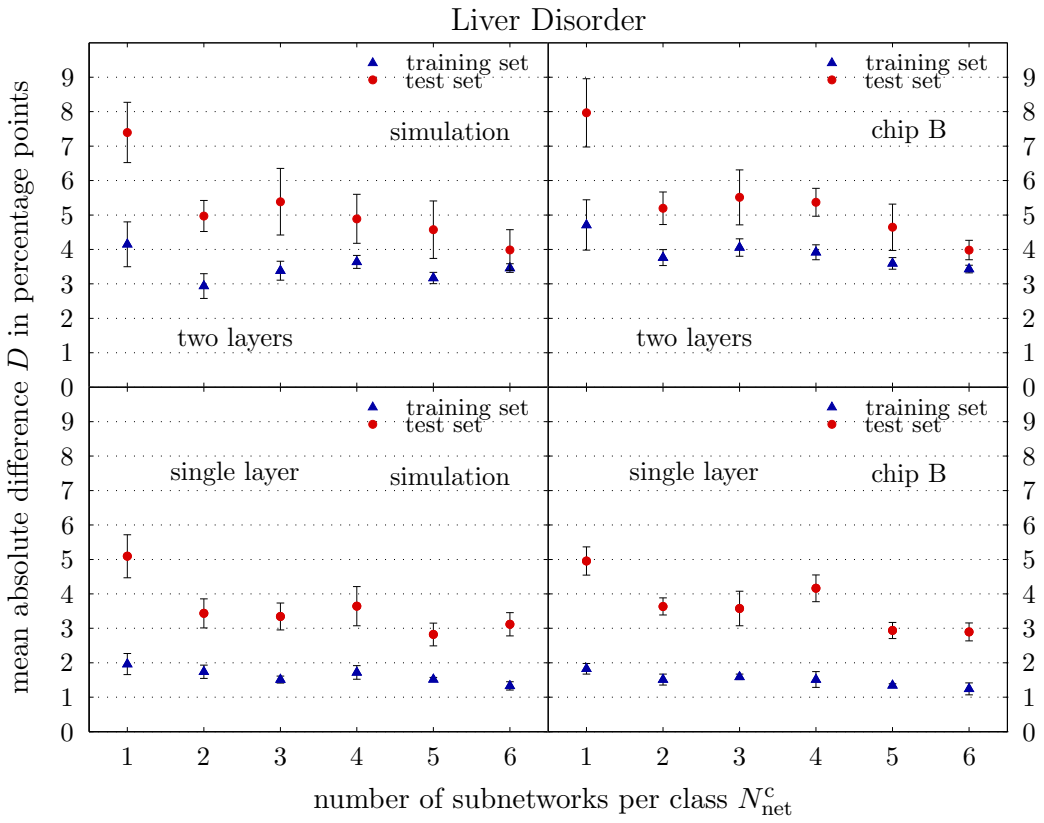


Figure 10.11: *The networks that have been trained for the liver disorder benchmark during the experiments presented in section 9.4 are implemented in software using a dedicated chip simulation that is based on the ideal network model defined by equation 5.4. The mean absolute differences in classification accuracy on the training and test sets between the original and the simulated networks (see also equation 10.2) are shown as a function of the number of subnetworks per class N^c_{net} (left half). For comparison, the corresponding averaged differences that are obtained during a network transfer to "chip B" are shown in the right half (compare figure 10.6). The results are very similar. It can be concluded that apart from the known deviations, the HAGEN prototype actually implements the desired network model.*

10.4.2 Stepwise Training of Software Networks

For the wine benchmark, networks are trained with the generalized stepwise training strategy that are not implemented on the HAGEN ASIC but are executed in software. The currently used software simulation of the employed network model is mainly designed to be smoothly integrated into the HANNEE framework and to be usable in conjunction with the same training algorithm implementations as the actual neural hardware. At present, this setup is not optimized for speed and the required training times are considerably larger than if networks are trained on the HAGEN chip. For this reason, the measurements reported here are not conducted with the same thoroughness as those presented in section 9.4.

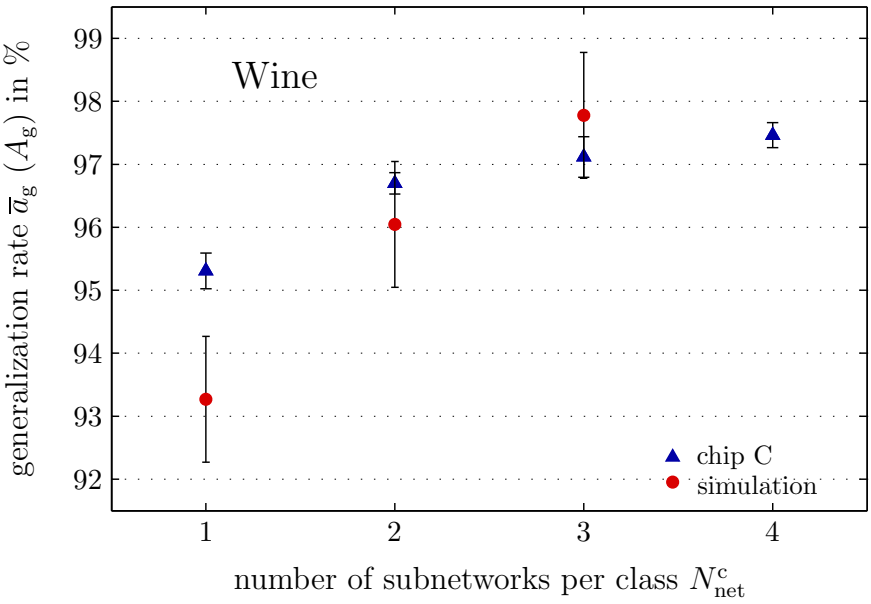


Figure 10.12: *The generalized stepwise strategy is employed for the training of networks for the wine problem that are purely implemented in software. For each number of subnetworks per class N_{net}^c, a single 10-fold cross-validation (see section 8.1.2) is performed and the resulting averaged accuracy on the test sets $\overline{a}_g$ is shown (circular markers). For comparison, the mean generalization rates A_g that have been obtained during the corresponding measurements with hardware implemented networks (see section 9) are presented as well (triangles, compare figure 9.4).*

measurement setup

Most of the parameters of the training setup equal those already used for the original investigations of chapter 9 (see tables 8.3 and 9.1). The networks exhibit a two-layer architecture and contain $N_{hid}^{sn} = 6$ hidden nodes and $N_{out}^{sn} = 4$ outputs per subnetwork. However, the following modifications to the experimental procedure are made: Since the software simulation is deterministic and does not suffer from any analog noise, each training pattern is only processed once by each network in each iteration. The same applies to the evaluation of the final network's classification performance on the training and test sets. To further reduce the expenditure of time, only a single 10-fold cross-validation is performed for each measurement, and the maximum number of training iterations in each phase of the stepwise training procedure is reduced to 500.

estimated uncertainty

Figure 10.12 shows the obtained averaged classification accuracy on the test set $\overline{a}_g$ as a function of the number of subnetworks that are trained per class. The standard error of the mean that is on average obtained on the 10 networks that are trained for each of the 10 partitionings (see section 8.1.2) is in the order of $\pm 1\,\%$ (corresponding error bars are included in figure 10.12). Although it is understood that this quantity is not the adequate error measure to be consulted for a comparison between the shown data points (see section 8.1.2), it is expected to provide a realistic upper limit to the remaining statistical uncertainty that will be observed when multiple cross-validations are performed. For comparison, the

corresponding accuracies A_g that have been obtained during the experiments of section 9.4 (see figure 9.4) are included in figure 10.12 as well.

comparison

For $N^\mathrm{c}_\mathrm{net} = 1$, the software simulation performs measurably worse than the networks that are implemented on the hardware. Given the used setup, this might partly be accredited to the reduced training times. But it is also conceivable that since the temporal fluctuations within the HAGEN chip and the resulting noisy network response might yield a smoother fitness measure (see also section 8.3.3), the evolutionary training algorithm could optimize networks on the hardware more efficiently. In so far, the present analog noise might even be beneficial for the eventual training success. Still, the available experimental data does not yet suffice to support this conclusion, and further measurements will be necessary to illuminate this interesting question.

improving generalization

Apart from that, it can be observed that training multiple subnetworks per class also improves the performance of software-implemented networks. For a number of 3 subnetworks per class, the simulated networks even seem to achieve a better performance than those on the HAGEN ASIC. Due to the limited amount of data and the remaining uncertainty, this result should not be over-interpreted. Still, the shown data suggests that with growing $N^\mathrm{c}_\mathrm{net}$, the simulated networks might become at least as good as the networks that are trained on the hardware.

It is a satisfying observation that the concepts of the proposed stepwise training strategy seem to be readily transferable to ideal, software-implemented networks of threshold neurons as well. Although it can reasonably be assumed that these observations will hold also for the other benchmark problems, this will ultimately have to be confirmed by additional experiments.

10.4.3 Software Networks in Hardware

evaluation procedure

Within a final set of measurements, the software networks that are trained for the wine problem during the investigations discussed in the preceeding section are loaded into the HAGEN ASIC formerly denoted as "chip B" to be implemented in hardware. The observed differences in performance between the original software networks and their hardware implemented pendants are evaluated according to the procedure described in section 10.2.1. But since only one 10-fold cross-validation is performed for each value of $N^\mathrm{c}_\mathrm{net}$, no final averaging over multiple repetitions is done.

observations

The resulting averaged absolute differences $\overline{d}_\mathrm{t}$ and $\overline{d}_\mathrm{g}$ are shown in figure 10.13. With only the networks of one cross-validation measurement being considered, no error measures can be provided that are reasonably comparable to those included in figure 10.7 (see also section 8.1.2). It is expected that a repetition of the cross-validation will reveal statistical fluctuations in the same order of magnitude as those obtained during the previously presented experiments. In comparison to the results that have been presented in sections 10.2.1 and 10.4.1 (see also figure C.15), it can be observed that training a network for the ideal substrate and transferring it to a real HAGEN chip on average yields approximately the same difference in performance as it is obtained in the inverse case or when networks are transferred between different ASICs. Again, this can be seen as to support the claim that

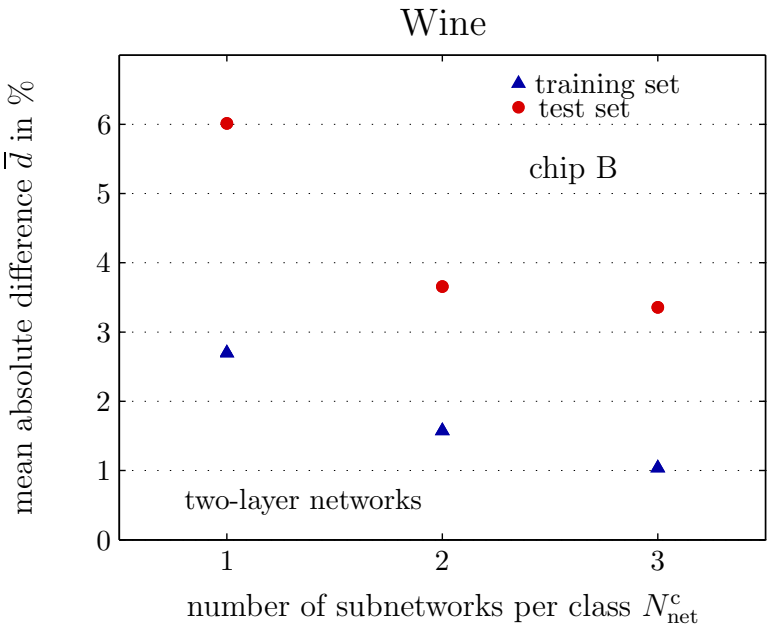


Figure 10.13: *The networks that have been trained in software for the wine benchmark during the experiments presented in the previous section are loaded into the HAGEN ASIC denoted as "chip B". The obtained averaged absolute differences in classification accuracy on the training and test sets $\overline{d}_{\mathrm{t}}$ and $\overline{d}_{\mathrm{g}}$ (see also equation 10.2) are shown as a function of the number of subnetworks per class $N_{\mathrm{net}}^{\mathrm{c}}$. The resulting differences are in the same order of magnitude and show the same dependence on $N_{\mathrm{net}}^{\mathrm{c}}$ as it is observed when networks are trained in hardware and are transferred to an ideal software simulation (compare also figure C.15).*

the employed circuits actually implement the intended network model defined by equation 5.4 — at least apart from the inevitable static device variations and temporal fluctuations.

off-chip training

Moreover, the shown data suggests that even if an off-chip optimization of network configurations for the HAGEN ASIC in software cannot benefit from the speed of the used hardware environment or the inherent parallel nature of the stepwise training approach as easily as a chip-in-the-loop procedure, it does provide a feasible training approach in principle. This is also suggested by earlier results that have been obtained with different network setups and training approaches [151].

improving off-chip training

Several precautions to improve the transferability of networks between different chips have been discussed in section 5.5.4, and these concepts can be expected to also improve the transferability of network configurations that are trained in software. In combination with a potential fine-tuning of the transferred networks by a suitable chip-in-the-loop algorithm, this eventually opens interesting new possibilities for alternative training strategies. However, the investigation of efficient off-chip training approaches for networks on the HAGEN ASIC lies beyond the scope of this work and is to be deferred to future investigations.

Summary and Outlook

You cannot depend on your eyes when your imagination is out of focus.

Mark Twain

the HAGEN chip

The HAGEN neural network ASIC provides flexible means for the implementation of fast and massively parallel neural networks with in the order of 10^4 synapses in a low power, mixed-signal hardware. By confining the analog operation to blocks with an entirely digital interface, the employed network model allows for a feasible up-scaling to even larger networks. On the other hand, the binary nature of the neurons in combination with the inevitable analog noise that remains to affect the operation of the network impede a direct application of traditional gradient-based training algorithms. Chip-in-the-loop training approaches can not only automatically cope with these peculiarities of the HAGEN chip, but can also make best use of its high reconfigurational speed and fast operation.

classification benchmarks

Within this thesis, it has been investigated in how far evolutionary chip-in-the-loop algorithms can successfully be applied to the training of neural networks on the HAGEN ASIC for demanding classification tasks. To this end, the performance of the examined evolutionary strategies has been tested on a set of nine well-known classification benchmarks: the breast cancer, diabetes, heart disease, liver disorder, iris plant, wine, glass, *E.coli*, and yeast problems.

evolutionary coprocessor

In order to efficiently train neural networks on the HAGEN chip, the evolutionary algorithm itself needs to be realized in a way such that it can keep up with the speed of the networks. The used hardware environment features a dedicated evolutionary coprocessor that is implemented within a configurable logic and accelerates evolutionary algorithms by performing the time-consuming genetic operations in hardware. In its current version, the coprocessor persists to impose certain restrictions on the complexity of the used genetic representation and the applied variation operators. At present, only direct encoding schemes are supported and the architecture of the trained networks needs to be fixed during training. In general, to be suited for use within the introduced hardware environment, the training algorithm is preferably kept simple and should rather aim to efficiently benefit from parallelization.

the simple evolutionary approach

Experiments have been presented that examine a simple evolutionary algorithm that utilizes the evolutionary coprocessor for the genetic operations. While this algorithm allows for a fast implementation and therefore ideally suits the used

hardware neural network framework, it turns out to be not capable of training networks on the HAGEN ASIC to achieve a satisfactory performance on the selected classification benchmarks. This particularly applies to problems that include more than two classes.

the stepwise training strategy

Therefore, a stepwise training strategy has been developed that divides the original task of training one large network for the entire categorization problem into the separate optimization of multiple, smaller subnetworks that each address only a part of the whole task: Each of the subnetworks is trained to differentiate the members of only one specific class against all remaining instances. Using a two-layer architecture, it is a notable feature of this strategy that additional interconnections between the single subnetworks can be introduced and optimized within a second training stage. This can be done separately for all subnetworks. As an important consequence, the single training steps of the whole procedure can be performed in parallel. Furthermore, it has been verified experimentally that each of the independent training phases can successfully be accomplished by simple and fast evolutionary algorithms that benefit from the functionality of the evolutionary coprocessor.

multiple subnetworks per class

It is straight forward to generalize the proposed stepwise strategy to incorporate the training of multiple subnetworks for the same class instead of one. An exhaustive campaign of measurements has been conducted that evaluates in how far an increased number of subnetworks per category improves the classification performance of the final networks. This has been done for all investigated benchmarks and for both, single-layer and two-layer networks. The results reveal that the use of multiple subnetworks per class never leads to a decrease of the achieved classification rate.

improved accuracy

For all considered tasks — except for the breast cancer and iris plant problems — an increased number of subnetworks has been observed to lead to a significant improvement of the network performance. In the case of the breast cancer data set, the achieved gain in generalization accuracy does not exceed the estimated statistical uncertainty and for the iris data set, a measurable improvement has only been obtained with single-layer networks.

single-layer networks

Apart from that, single-layer and two-layer networks have been shown to equally benefit from an enlarged number of subnetworks. For the heart disease and iris plant benchmarks, it has even been observed that any initial differences in performance between the two architectures tend to vanish once a sufficient number of subnetworks is trained. It is an important aspect of the proposed generalized stepwise strategy that it allows for an efficient application of simple single-layer networks to challenging classification tasks. Single-layer architectures can more easily be scaled to fully exploit the resources of the currently used HAGEN prototype than two-layer networks.

comparison to previous results

For the breast cancer, diabetes, heart disease, and glass benchmarks, the finally obtained generalization rates have been compared with the accuracies of software-implemented networks that have previously been reported by other authors. It is a satisfying observation that in comparison to these results, the networks on the HAGEN ASIC show a more than competitive performance (see table 9.3). The achieved generalization rates have been shown to be even comparable to those of

common neural network ensemble approaches (see table 9.5). Although for each of the nine investigated task, a better classifier — not necessarily a neural network — can eventually be found in literature (see tables 9.4 and 9.5), networks on the HAGEN ASIC that are being trained by the introduced stepwise strategy have been shown to achieve a satisfying performance on all benchmarks.

successful training

The presented experiments are the first to demonstrate that networks on the HAGEN chip can actually be trained to outperform other neural network implementations on realistic classification tasks. This is remarkable in so far as each of the independent training phases is performed by a simple evolutionary algorithm and none of the training parameters has been subject to an exhaustive optimization. Most notably, the size of the independently trained subnetworks has been fixed to 6 hidden neurons and/or 4 output nodes.

an alternative approach

The proposed strategy of training multiple subnetworks per class has been compared to an alternative approach where only a single subnetwork is included for each category that contains a larger number of neurons instead. It has been observed that although an improvement in generalization accuracy can be obtained on some benchmarks, this latter procedure tends to be inferior to the training of multiple subnetworks.

parallelization

Moreover, it has been demonstrated that the inherent parallelism of the stepwise approach can be exploited already within the currently used hardware environment that includes only a single ASIC and one evolutionary coprocessor within each training setup. In this configuration, the proposed parallelization scheme can speed up the training by up to a factor of 3. The upcoming NATHAN/Backplane system will allow to make even better use of both, the speed of the neural network ASIC and the parallel nature of the presented training approach. Since it can successfully employ simple and fast evolutionary algorithms and is readily suited for a parallel implementation, the stepwise training strategy can be seen to ideally complement the introduced hardware neural network framework.

network transfer

For the liver disorder, wine, and *E.coli* benchmarks, the trained network configurations have been transferred to other HAGEN chips than the one they had originally been trained on, and it has been evaluated how this affects the network operation. As expected, the response of a transferred network deviates from the output of the original. But it has been observed that the training of multiple subnetworks per class improves the stability of the networks: Although the differences to the respective original do not vanish completely, they are measurably reduced. Transferring the network configurations to a software simulation of the ideal network model has been shown to yield similar results as if these networks are tested on different chips.

varied subnetwork size

As an outlook to future investigations, it has been examined whether a moderately reduced size of the single subnetworks affects the generalization rates of networks that are trained for the liver disorder problem. If only one subnetwork is included for each class, the numbers of hidden neurons and outputs per subnetwork have indeed been observed to affect the performance of the final network. These differences have been seen to disappear as soon as multiple subnetworks per class are trained. Apart from that, it has been shown that the connectivity of networks which are trained for the heart disease problem can be restricted — such

that not all hidden neurons are connected to all inputs — without leading to a significant reduction in classification accuracy. This is a satisfying observation in so far as more complex problems with large numbers of input attributes will require the input layer of the network to be distributed over multiple network blocks of the HAGEN ASIC or even multiple chips. The obtained results suggest that this is not necessarily bound to cause a deterioration in performance.

software simulations

Preliminary measurements have been performed that test the applicability of the stepwise training strategy to software simulations of the used network model. The training of multiple subnetworks per class has been observed to be beneficial also for software-implemented networks that do not suffer from static offsets or analog noise. The resulting network configurations have been loaded into a real HAGEN chip. The observed differences in performance are similar to those that have been obtained with networks that are originally trained on the hardware and transferred to the software simulation. This opens interesting new possibilities for suitable off-chip training approaches.

outlook

It persists that evolutionary chip-in-the-loop algorithms represent an ideal way to benefit from the speed of the neural hardware during training and at the same time automatically compensate for the peculiarities of the mixed-signal network implementation. The presented results establish the stepwise evolutionary strategy as an efficient approach to successfully train complex networks on the HAGEN ASIC for real-world classification tasks. Due to its inherent scalability and parallelism, the proposed training procedure promises to be readily applicable to the enlarged networks that can be realized within the upcoming advanced hardware environment and on future ASICs. In combination with hierarchical network approaches that are also investigated in the Electronic Vision(s) group [52], it is planned to employ the stepwise strategy for the training of large and massively parallel networks for complex real-time image recognition tasks in the future.

Appendix

Appendix A

Exemplary HANNEE Code

```
#include "hvalue.h"

#include "halgorithm.h"

class HMyAlgo :  public HAlgorithm {
  HIntValue* maxiter_;
  bool       paused_;
  int        curriter_;

public:
  HMyAlgo();
  virtual ~HMyAlgo();

  void pause();
  void initialize(HAlgorithm*);
  HNetData* result();
  bool success() const;

protected:
  void setup();
  void exec();
  void clear();
};
```

Figure A.1: *Declaration of the exemplary* `HMyAlgo` *class like it would typically be found in a corresponding* `"hmyalgo.h"` *file (see section 7.2.2). The* `HMyAlgo` *class implements the* `HAlgorithm` *class interface (see figure 7.8). The complete code for a simple but sensible implementation can be found in figure A.2 on the next page.*

```
#include "hmyalgo.h"

HMyAlgo::HMyAlgo()
  : HAlgorithm("HMyAlgo", "This is my algorithm.") {
  maxiter_ = new HIntValue("Maximum Iteration", 10);
  addElement(maxiter_);
  paused_ = false;
  curriter_ = 0;
}

HMyAlgo::~HMyAlgo() {}

void HMyAlgo::pause() {  paused_ = false;   }

void HMyAlgo::initialize(HAlgorithm* algo) {
  log << "I am being initialized.";
  log << "Nothing happens." << hlog;
}

HNetData HMyAlgo::result() {  return 0;   }

bool HMyAlgo::success() const {  return true;  }

void HMyAlgo::setup() {
  log << "Starting HMyAlgo!  " << hlog;
}

void HMyAlgo::exec() {
  paused_ = false;
  while (curriter_ < maxiter_->get() && !paused_) {
    log << "Current Iteration:  " << curriter_ << hlog;
    curriter_++;
  }
}

void HMyAlgo::clear() {   curriter_ = 0;   }

HObjectMapEntryImpl<HMyAlgo> hma("HMyAlgo", "HAlgorithm");
```

Figure A.2: *Complete implementation of the exemplary* `HMyAlgo` *class as it is declared in figure A.1 (see also section 7.2.2). Note how the internal integer variable of* `maxiter_` *is accessed in the termination condition of the* `while` *loop in the* `exec()` *method. The* `log` *stream is a HANNEE peculiarity. Effectively, it conveys the passed text to the output stream and also writes it into a special log file. The last line registers the new class with the HANNEE application. A sreenshot of the corresponding user interface together with the output of one execution of this algorithm is presented in figure 7.10.*

```
#include "hobject.h"

class HMyClass :  public HObject {
public:
  HMyClass();
  virtual ~HMyClass();
};
```

Figure A.3: *Declaration of the exemplary* `HMyClass` *class like it would typically be found in a corresponding* `"hmyclass.h"` *file.*

```
#include "hvalue.h"
#include "hmyclass.h"

HMyClass::HMyClass()
  : HObject("MyObject", "This is my object.")
{
  addElement(new HIntValue("Integer Parameter", 2));
  addElement(new HDoubleValue("Float Parameter", 1.5));
  addElement(new HBoolValue("Bool Parameter", false));

  addElement(new HGroup("Empty Group 1"));
  addElement(new HGroup("Empty Group 2"));
}

HMyClass::~HMyClass() {}

HObjectMapEntryImpl<HMyClass> mycl("HMyClass", "HObject");
```

Figure A.4: *Complete implementation of the* `HMyClass` *as it is declared in figure A.3. Objects of this class are given three parameters, an integer, a float (with double precision) and a Boolean. The default value of each parameter is passed as the second argument to its respective constructor. In this simple example, the two subgroups remain empty. Note that the destructor is not required to* `delete` *the various* `HValue` *objects that have been created in the constructor, since the* `HObject` *base class automatically takes care of their destruction. The last line registers the new class with the HANNEE application.*

```
cop_->allocCmd(7);
ibufaddr_ = cop_->allocIbuf(6);

length_   = 128;
cutpoint_ = 72;

cop_->writePCIbuf(ibuffaddr_, parentone_);
cop_->writePCIbuf(ibuffaddr_+1, length_);
cop_->writePCIbuf(ibuffaddr_+2, parenttwo_);
cop_->writePCIbuf(ibuffaddr_+3, length_);
cop_->writePCIbuf(ibuffaddr_+4, offspring_);
cop_->writePCIbuf(ibuffaddr_+5, length_);

cop_->setProbabUni(983);
cop_->setProbabGau(0);

cop_->maskNull(cutpoint_);
cop_->maskNull(length_-cutpoint_);
cop_->sourceOne(ibuffaddr_);
cop_->sourceTwo(ibuffaddr_+2);
cop_->target(ibuffaddr_+4);

cop_->start();
while (!cop_->finito());

cop_->freeCmd();
cop_->freeIbuf(ibuffaddr_);
```

Figure A.5: *Exemplary code to illustrate the use of the* `EvoCop` *class (see also section 7.3.3 and figure 7.13). It is assumed that the* `cop_` *handle points to a valid* `EvoCop` *object, and that the integer variables* `parentone_` *,* `parenttwo_` *, and* `offspring_` *code the addresses of two parent chromosomes resp. a valid target address for one offspring chromosome in the local memory of the FPGA (see section 6.2). The length of each chromosome is assumed to be 128 and the two parents are to be recombined by a common one-point crossover using the (arbitrary) cutpoint* $r_c = 72$*. The probability for uniform mutation is desired to be* $983/65535 \approx 0.015$*, and the Gaussian mutation is turned off. First, the necessary space in the instruction buffer is allocated. The three pairs of chromosome address and length require a total of six entries in the data buffer, and for the recombination and mutation process, space for a total of seven instructions is to be reserved. Once this is accounted for, the actual address information is written into the data buffer and the coproccessor instructions are issued in the desired order. Finally, the recombination procedure is initiated with a call to* `start()`*. The last two commands do a cleanup in the instruction buffer. In practice, the mutation parameters are specified only once at the beginning of an evolution run. Furthermore, multiple chromosomes are processed with one execution of the coprocessor, and the required address information that is written to the data buffer can be reused for multiple recombination procedures.*

Appendix B

The Investigated Benchmark Problems

All data sets that are used for the presented experiments have been obtained from the UCI KDD online archive [90]. Some are also included in the "Proben1" benchmark collection compiled by Lutz Prechelt [163]. The numbers of classes, instances, and attributes for the different classification tasks are summarized in table 8.1. The following paragraphs are intended to provide additional information about the origins, compositions, and relative difficulties of the different problems.

The breast cancer problem This data set was originally obtained from the University of Wisconsin Hospital, Madison, from Dr. William H. Wolberg [133]. The task is to classify a tumor as either benign or malignant based on cell descriptions gathered by microscopic examination (e.g., clump thickness, cell size uniformity, the frequency of bare nuclei, etc.). The original data set contains 699 instances but 16 include missing attribute values and are therefore omitted. 65 % of the used examples are benign.

The diabetes problem The data in this set was originally donated by Vincent Sigillito from the Johns Hopkins University. Based on various personal data and medical examinations (age, blood pressure, body mass index, etc.), it is to be decided whether a female patient of Pima Indian[1] heritage and age above 21 would test positive for diabetes according to the criteria of the World Health Organization or not. 65.1 % of the patients are tested negative for diabetes.

The heart disease problem The experiments presented in this thesis only use the specific part of this data set that originates from the Cleveland Clinic foundation (this corresponds to the `heartc` data set of the "Proben1" collection by Prechelt [163]). It was supplied by Robert Dotrano of the V.A. Medical Center, Long Beach, CA. On the basis of personal data (age, sex, smoking habits, etc.) and various medical examinations such as blood pressure and electro cardiogram

[1] The Pima Indians live near Phoenix, Arizona, USA.

results, it is to be decided whether at least one major vessel is reduced in diameter by more than 50 %. The whole data set comprises 303 instances, 6 of which exhibit missing attribute values and have therefore been omitted. 53.81 % of the patients have "no vessel reduced".

The liver disorder problem This data set was created by BUPA Medical Research Ltd. and donated by Richard S. Forsyth, 8 Grosvenor Avenue, Mapperley Park, Nottingham. The task is to predict the presence of liver disorder for a male patient on the basis of several blood tests and the number of half-pint equivalents of alcoholic beverages drunk per day. It appears that a value larger than five for the latter attribute is some sort of a selector on this data set. The first class contains 42 % of all instances.

It is to be noted that the above four data sets are commonly considered to represent some of the most challenging problems in the neural network and machine learning field since they all feature a comparably small sample size and high noise level [236].

The iris plant problem This is perhaps the best known database to be found in the pattern recognition literature. It was created by R.A. Fisher and donated by Michael Marshall. The three classes each refer to a type of iris plant (*Iris Setosa*, *Iris Versicolour*, and *Iris Virginica*) that are to be distinguished on the basis of the respective dimensions of the sepal and petal. Each class contains 50 instances (33.3 %) and one class is known to be linearly separable from the other two. The latter are *not* linearly separable. The iris plant benchmark is widely regarded as a comparably simple task.

The wine problem The data in this set was originally donated by M. Forina, Institute of Pharmaceutical and Food Analysis and Technologies, Via Brigata Salerno, Genoa, Italy and was first used as a benchmark for pattern recognition by Aeberhard *et al.* [3]. The data represents the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines. The three classes contain 33.1 %, 39.9 %, and 27 % of the data, respectively. In a classification context, this is a well posed problem which is commonly considered as a good data set for first testing a new classifier, but it is not truly challenging (see also table 9.4).

The glass problem This data set was created by B. German from the Central Research Establishment, Home Office Forensic Science Service, Aldermaston, Reading, Berkshire and was donated by Vina Spehler, PhD., DABFT Diagnostic Products Corporation. The task is to distinguish glass types. The results of a chemical analysis of glass splinters (percent content of eight different elements) plus the refractive index are used to classify the sample to be either float processed or non float processed building windows, vehicle windows, containers, tableware,

or head lamps. This is motivated by forensic needs in criminal investigation. The largest class (non float processed building windows) contributes 35.5 % of all instances. The remaining classes contain 32.7 %, 13.6 %, 7.9 %, 6.1 %, and 4.2 % of the data set, respectively.

The *E.coli* and yeast problems These two data sets were created by Kenta Nakai, Institute of Molecular and Cellular Biology, Osaka University, Japan and were donated by Paul Horton. The task is to predict the cellular localization sites of *E.coli* resp. yeast proteins. The decision is to be made on the basis of a set of feature values that are calculated from the protein's amino acid sequence. These two data sets contain comparably large numbers of classes (compare table 8.1). For the *E.coli* benchmark, the single classes contain 42.6 %, 22.9 %, 15.5 %, 10.4 %, 6.0 %, 1.5 %, 0.6 %, and 0.6 % of the instances, respectively. For the yeast benchmark, the most common class contributes 32.2 %, and the remaining categories comprise 28.9 %, 16.4 %, 11.0 %, 3.4 %, 3.4 %, 2.4 %, 2.0 %, 1.3 %, and 0.3 % of the data, respectively.

Appendix C

Experimental Data

This page remains empty. The presentation of the experimental data starts on the next page.

benchmark	no Gaussian mutation		no crossover	
	training set A_t in %	test set A_g in %	training set A_t in %	test set A_g in %
breast cancer	98.16 ± 0.02	95.90 ± 0.19	98.29 ± 0.02	96.18 ± 0.10
diabetes	77.65 ± 0.09	73.61 ± 0.60	78.34 ± 0.16	73.33 ± 0.76
heart disease	88.66 ± 0.06	80.17 ± 0.85	89.33 ± 0.07	80.76 ± 0.17
liver disorder	75.49 ± 0.10	67.71 ± 0.83	76.08 ± 0.10	67.85 ± 0.83
iris plant	98.34 ± 0.10	93.51 ± 0.97	98.70 ± 0.11	94.03 ± 0.46
wine	88.72 ± 0.56	83.61 ± 0.91	89.29 ± 0.69	80.25 ± 1.60
glass	50.73 ± 0.51	43.25 ± 1.27	55.10 ± 0.36	49.91 ± 0.47

benchmark	one-point crossover		chromosome exchange	
	training set A_t in %	test set A_g in %	training set A_t in %	test set A_g in %
breast cancer	98.28 ± 0.03	96.33 ± 0.18	98.25 ± 0.02	96.07 ± 0.09
diabetes	78.03 ± 0.15	73.62 ± 0.36	78.17 ± 0.08	73.32 ± 0.67
heart disease	89.05 ± 0.14	82.47 ± 0.63	89.04 ± 0.06	82.26 ± 0.53
liver disorder	76.00 ± 0.08	68.21 ± 0.73	76.12 ± 0.16	67.15 ± 0.66
iris plant	98.73 ± 0.04	94.87 ± 0.74	98.71 ± 0.12	94.86 ± 0.62
wine	90.47 ± 0.54	85.03 ± 1.10	89.51 ± 0.49	81.83 ± 1.59
glass	51.68 ± 0.97	46.36 ± 1.22	50.22 ± 1.12	46.30 ± 1.56

benchmark	software algorithm		no coupled weights	
	training set A_t in %	test set A_g in %	training set A_t in %	test set A_g in %
breast cancer	98.17 ± 0.03	96.40 ± 0.10	97.28 ± 0.04	94.71 ± 0.15
diabetes	77.57 ± 0.17	72.68 ± 0.62	69.83 ± 0.15	61.94 ± 0.65
heart disease	88.98 ± 0.11	81.49 ± 0.72	86.68 ± 0.18	76.82 ± 0.32
liver disorder	76.03 ± 0.16	67.31 ± 0.74	69.28 ± 0.19	51.83 ± 1.13
iris plant	98.41 ± 0.08	94.88 ± 0.35	77.37 ± 0.31	58.06 ± 2.50
wine	90.31 ± 0.32	82.23 ± 0.90	64.69 ± 0.44	46.33 ± 1.71
glass	50.79 ± 0.92	44.44 ± 1.37	35.28 ± 0.46	25.00 ± 1.21

Table C.1: *Training results obtained with the simple evolutionary approach presented in chapter 8 using slightly modified training resp. network setups. Several different measurements are performed (see also section 8.4.3 and tables 8.5 and 8.6): For the first experiment, the Gaussian mutation is disabled and the probability for uniform mutation is raised to 4% (upper left table). The next three investigations employ different recombination operators: no crossover, one-point crossover, and chromosome exchange. Another experiment is conducted with unmodified evolution settings but using a pure software implementation of the genetic algorithms instead of the coprocessor. For the last investigation, finally, the enforced coupling of the synaptic weights to form predefined multi-bit integer inputs is relieved and all weights can be adjusted independently of each other.*

benchmark	No. of classes N_c	complete training	
		training set A_t in %	test set A_g in %
breast cancer	2	98.57 ± 0.01	96.44 ± 0.23
diabetes	2	79.79 ± 0.03	73.70 ± 0.31
heart disease	2	90.96 ± 0.08	80.03 ± 0.54
liver disorder	2	77.24 ± 0.11	66.51 ± 0.72
iris plant	3	99.50 ± 0.03	95.10 ± 0.54
wine	3	100.0 ± 0.00	95.31 ± 0.28
glass	6	83.11 ± 0.19	62.56 ± 1.31
E.coli	8	91.82 ± 0.14	81.01 ± 0.93
yeast	10	55.10 ± 0.24	51.18 ± 0.31

benchmark	No. of classes N_c	only stage 1	
		training set A_t in %	test set A_g in %
breast cancer	2	97.98 ± 0.02	95.34 ± 0.19
diabetes	2	72.82 ± 0.18	66.92 ± 0.67
heart disease	2	87.52 ± 0.11	79.82 ± 0.67
liver disorder	2	68.44 ± 0.18	60.24 ± 0.63
iris plant	3	98.29 ± 0.06	92.83 ± 0.44
wine	3	99.34 ± 0.06	91.07 ± 0.36
glass	6	73.00 ± 0.05	54.30 ± 0.60
E.coli	8	84.30 ± 0.18	74.37 ± 0.23
yeast	10	44.53 ± 0.30	42.55 ± 0.28

benchmark	No. of classes N_c	one-layer perceptron	
		training set A_t in %	test set A_g in %
breast cancer	2	97.89 ± 0.14	96.17 ± 0.14
diabetes	2	73.07 ± 0.09	69.39 ± 0.29
heart disease	2	87.44 ± 0.14	82.21 ± 0.57
liver disorder	2	71.17 ± 0.13	65.40 ± 0.45
iris plant	3	93.42 ± 0.50	88.46 ± 0.95
wine	3	99.07 ± 0.06	94.07 ± 0.45
glass	6	70.98 ± 0.07	60.89 ± 0.33
E.coli	8	86.62 ± 0.08	79.74 ± 0.33
yeast	10	45.91 ± 0.13	43.98 ± 0.41

Table C.2: *Training results obtained with the stepwise strategy described in section 9.1.1 (see also table 9.2). The upper table shows the accuracies that are achieved with fully connected two-layer perceptrons. For the experiments shown in the middle table, the second training stage is omitted and the subnetworks thus remain unconnected. The values in the lower table are obtained with single-layer perceptrons.*

benchmark	fixed partitioning	$N_{\text{net}}^{\text{c}}$	training set A_{t} in %	test set A_{g} in %
breast cancer	Proben1 a)	7	98.06 ± 0.07	98.73 ± 0.39
	Proben1 b)		98.67 ± 0.05	94.95 ± 0.28
	Proben1 c)		98.55 ± 0.10	95.94 ± 0.04
	Yao *et al.*		98.44 ± 0.04	99.12 ± 0.15
diabetes	Proben1 a)	6	80.54 ± 2.03	83.88 ± 0.44
	Proben1 a)		81.94 ± 2.30	84.86 ± 0.57
	Proben1 b)		80.01 ± 1.45	82.65 ± 0.40
	Yao *et al.*		83.60 ± 0.24	78.05 ± 1.29
heart disease	Proben1 a)	6	94.47 ± 0.16	81.60 ± 0.50
	Proben1 b)		92.36 ± 0.32	94.13 ± 0.68
	Proben1 c)		94.18 ± 0.16	84.71 ± 1.28
glass	Proben1 a)	8	74.43 ± 0.14	72.75 ± 0.74
	Proben1 b)		78.48 ± 0.37	62.64 ± 0.71
	Proben1 c)		75.82 ± 0.09	63.57 ± 0.92

Table C.3: *The results that are obtained on the fixed partitionings into raining and test data that are used by previous investigations: Prechelt [163] specifies three different fixed partitionings for each task that are denoted as "Proben1 a)–c)". Yao and Liu use one fixed separation [236] for the breast cancer and the diabetes data set, respectively. For the experiments with the stepwise training strategy, the exact compositions of the respective training and test data sets are reproduced. For a comparsion with the results of the cited publications see table 9.3.*

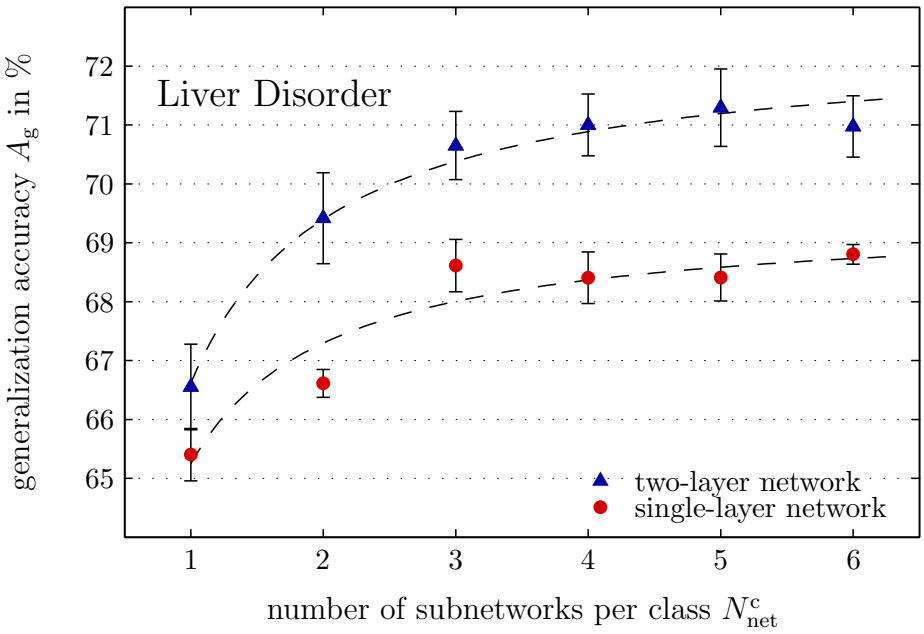


Figure C.1: *The achieved classification accuracies on the test sets A_g as a function of the number N_{net}^c of subnetworks per class for the liver disorder problem. The shown dashed lines are intended as mere guides to the eye (see section 9.4.1). The corresponding classification accuracies on the training sets A_t can be found in figure 9.4.*

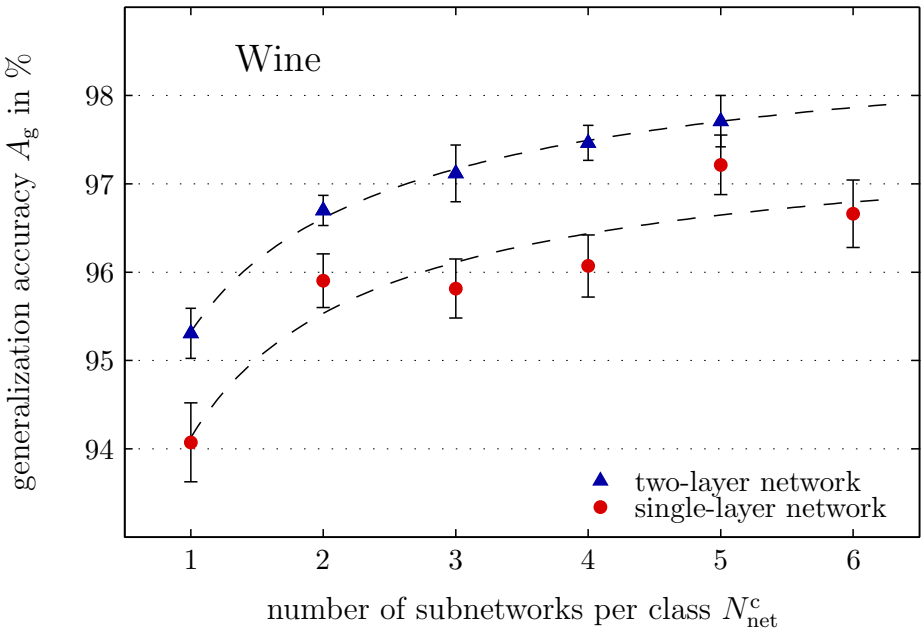


Figure C.2: *The achieved classification accuracies on the test sets A_g as a function of the number N_{net}^c of subnetworks per class for the wine problem. The shown dashed lines are intended as mere guides to the eye (see section 9.4.1). The corresponding classification accuracies on the training sets A_t can be found in figure 9.4.*

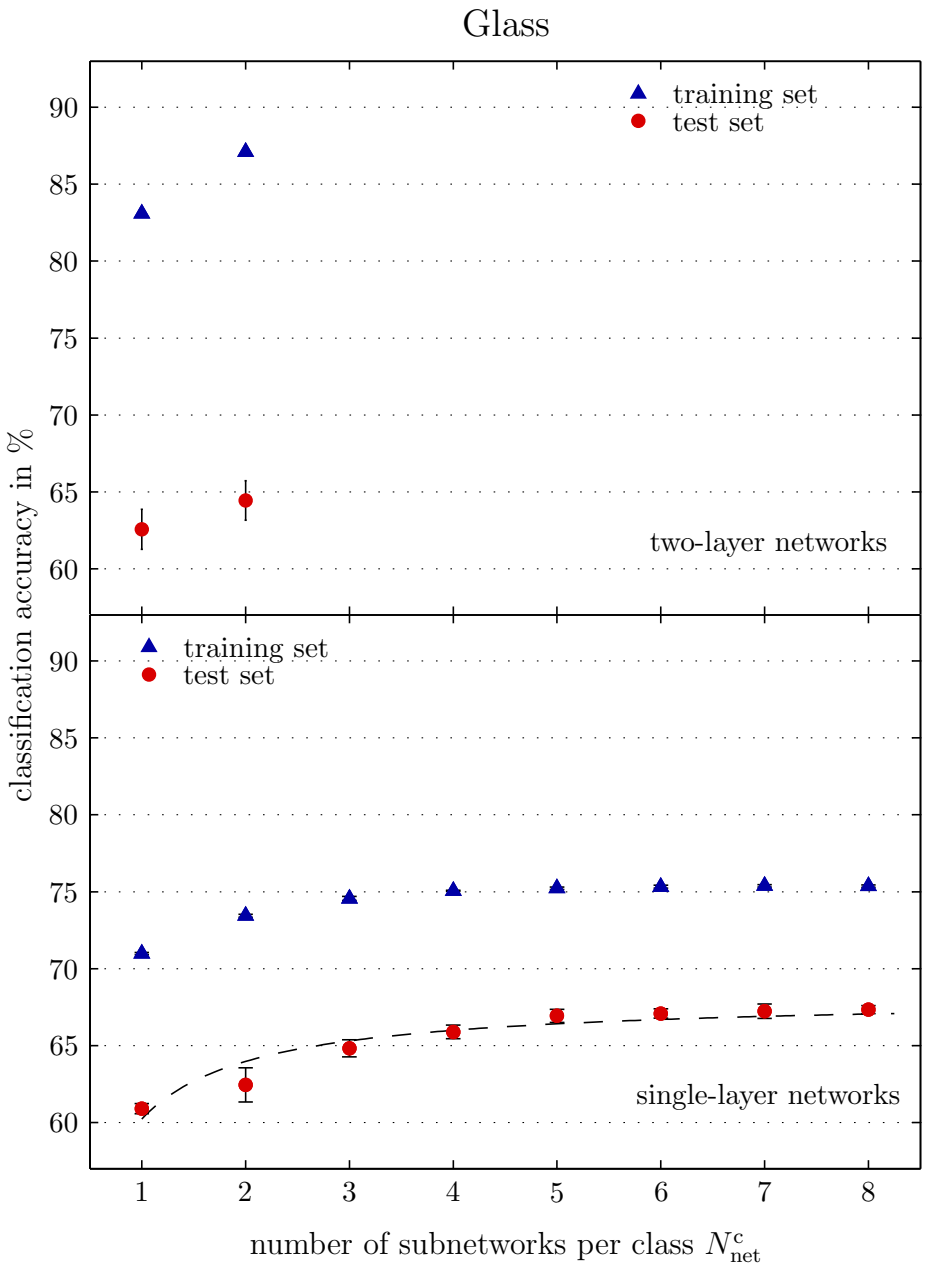


Figure C.3: *The achieved classification accuracies on the training sets A_t and test sets A_g as a function of the number N^c_{net} of subnetworks per class for the glass problem. The upper diagram presents the results of the two-layer networks, the lower diagram refers to single-layer perceptrons. Due the six classes of this benchmark and given the fixed subnetwork size of $N^c_{hid} = 6$ hidden neurons, a maximum of two subnetworks per class can be realized for the two-layer networks. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1). For the classification accuracies on the training set, the error bars are very small and are partly covered by the markers.*

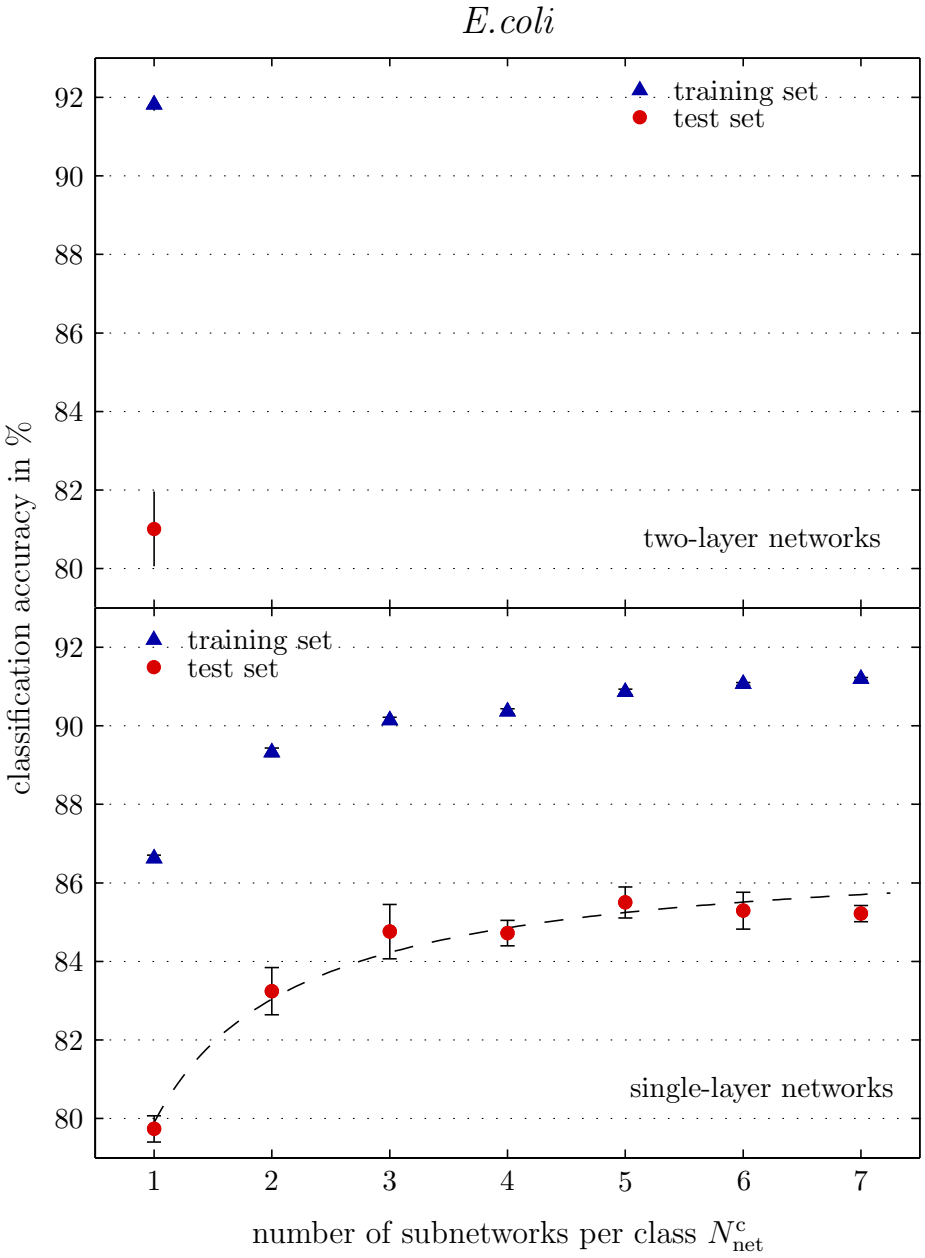


Figure C.4: *The achieved classification accuracies on the training sets A_t and test sets A_g as a function of the number N^c_{net} of subnetworks per class for the* E.coli *problem. The upper diagram presents the results of the two-layer networks, the lower diagram refers to single-layer perceptrons. Due the eight classes of this benchmark and given the fixed subnetwork size of $N^c_{hid} = 6$ hidden neurons, only one subnetwork per class can be realized for the two-layer networks. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1). For the classification accuracies on the training set, the error bars are very small and are partly covered by the markers.*

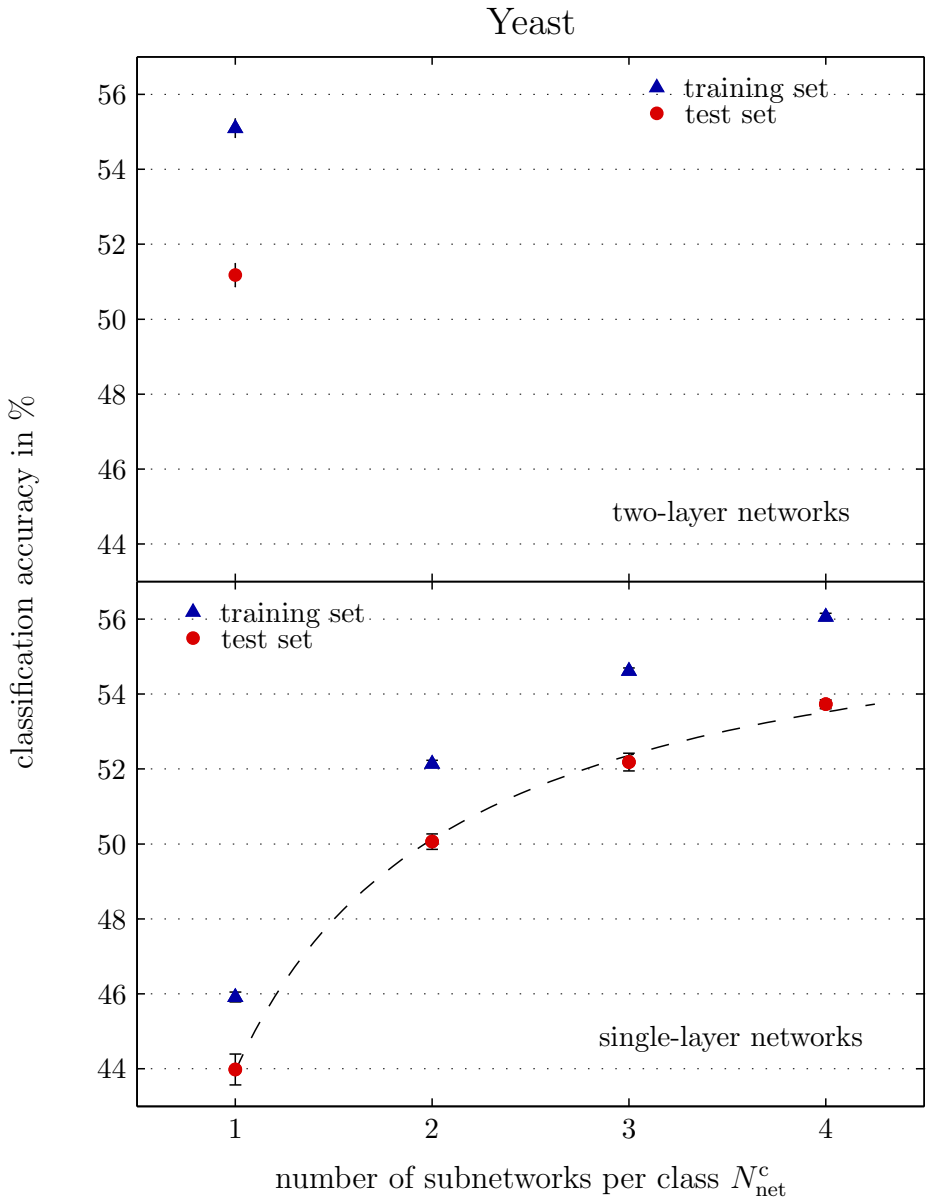


Figure C.5: *The achieved classification accuracies on the training sets A_t and test sets A_g as a function of the number N^c_{net} of subnetworks per class for the yeast problem. The upper diagram presents the results of the two-layer networks, the lower diagram refers to single-layer perceptrons. Due the eight classes of this benchmark and given the fixed subnetwork size of $N^c_{hid} = 6$ hidden neurons, only one subnetwork per class can be realized for the two-layer networks. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1). For the classification accuracies on the training set, the error bars are very small and are partly covered by the markers.*

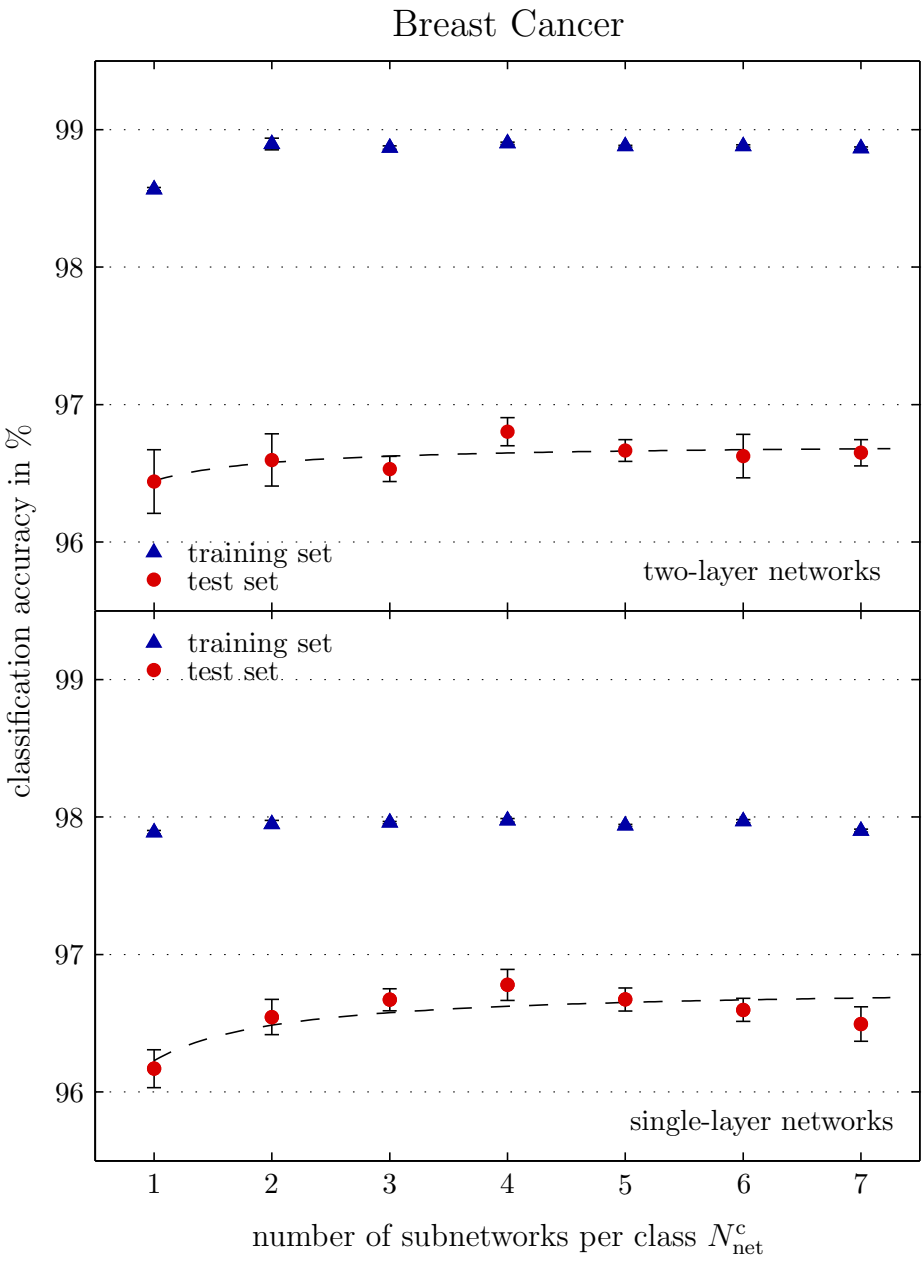


Figure C.6: *The achieved classification accuracies on the training sets A_t and test sets A_g as a function of the number N^c_{net} of subnetworks per class for the breast cancer problem. The upper diagram presents the results of the two-layer networks, the lower diagram refers to single-layer perceptrons. The shown dashed lines are intended as mere guides to the eye (see section 9.4.1). For the classification accuracies on the training set, the error bars are very small and are partly covered by the markers.*

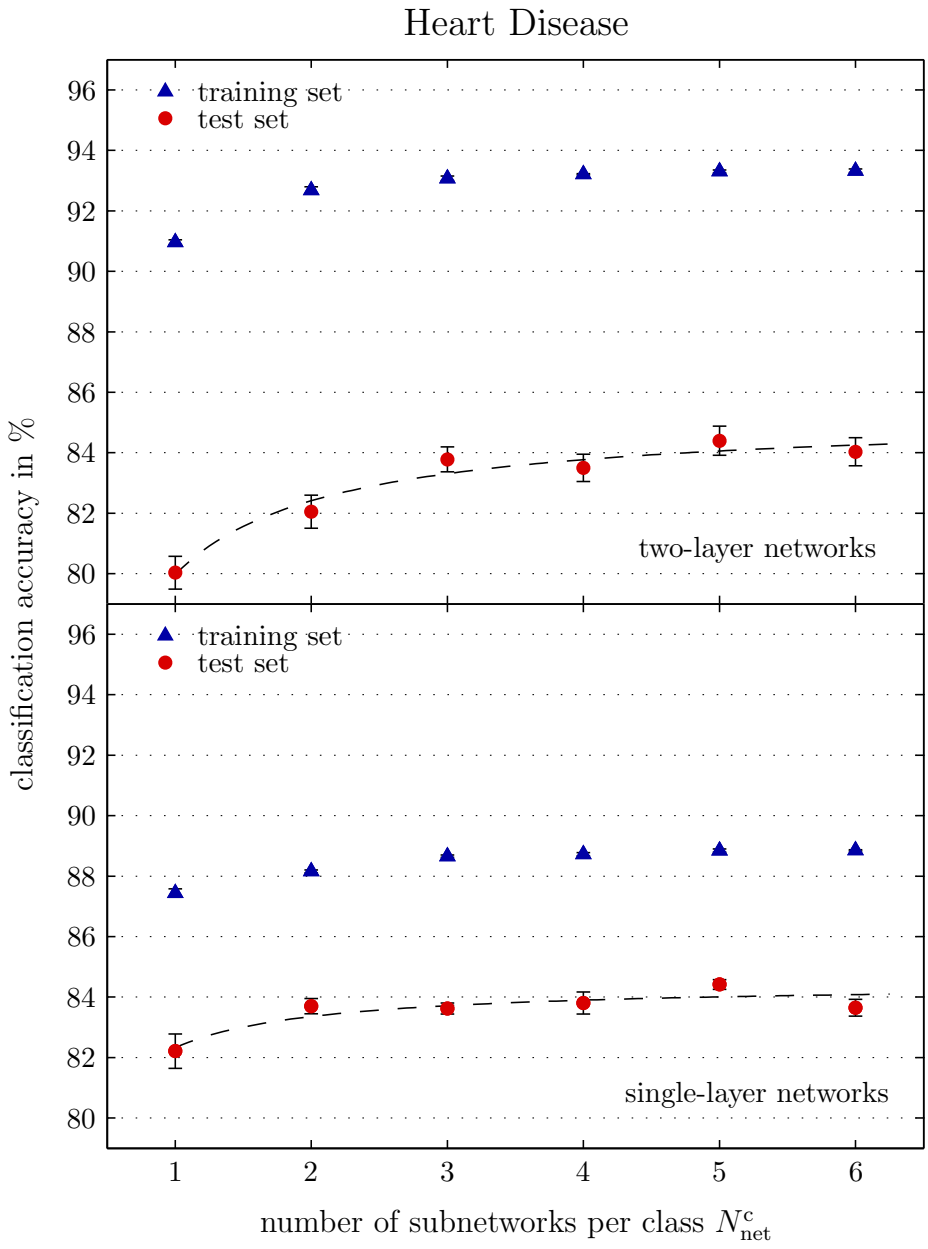


Figure C.7: *The achieved classification accuracies on the training sets A_{t} and test sets A_{g} as a function of the number $N_{\mathrm{net}}^{\mathrm{c}}$ of subnetworks per class for the heart disease problem. The upper diagram presents the results of the two-layer networks, the lower diagram refers to single-layer perceptrons. The shown dashed lines are intended as mere guides to the eye (see section 9.4.1). For the classification accuracies on the training set, the error bars are very small and are partly covered by the markers.*

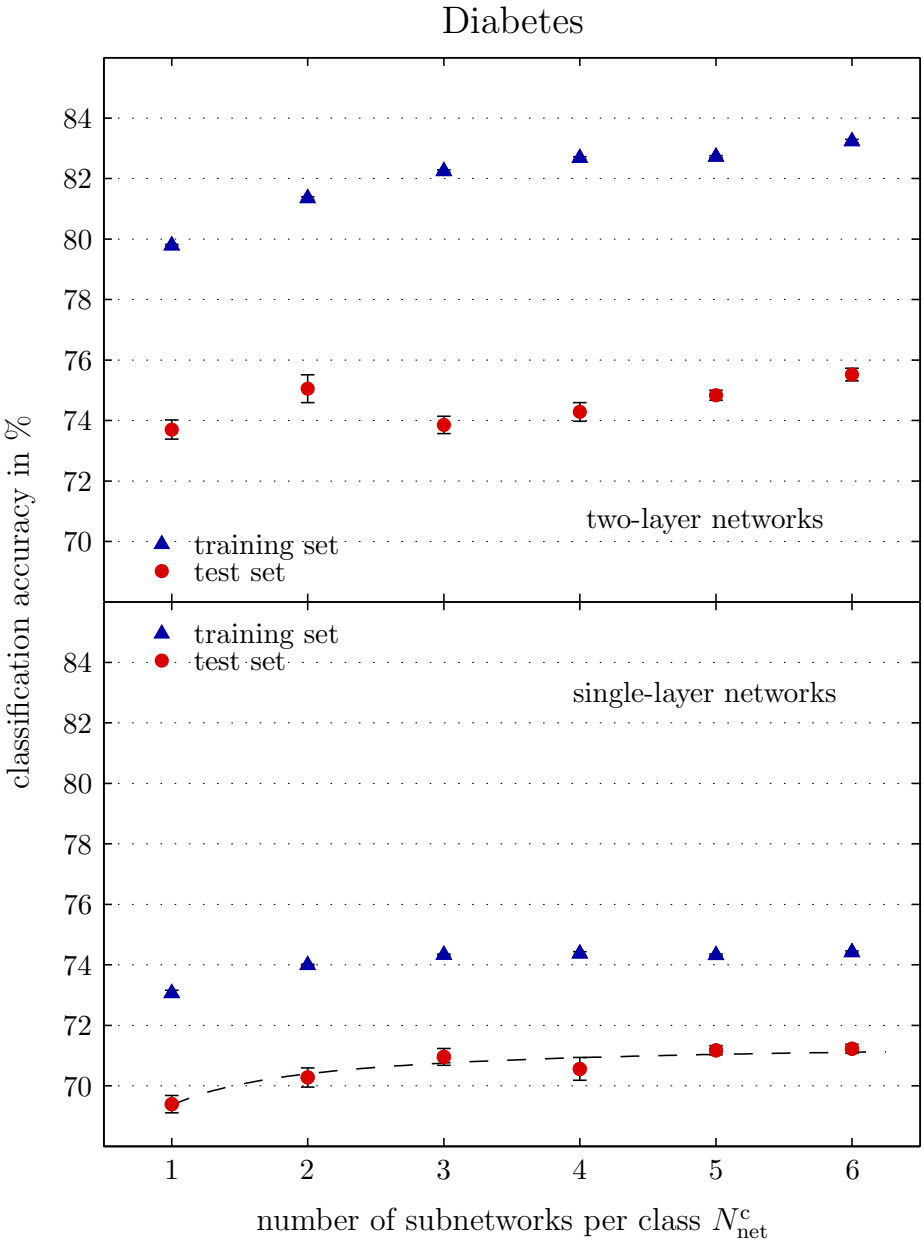


Figure C.8: *The achieved classification accuracies on the training sets A_{t} and test sets A_{g} as a function of the number $N_{\text{net}}^{\text{c}}$ of subnetworks per class for the diabetes problem. The upper diagram presents the results of the two-layer networks, the lower diagram refers to single-layer perceptrons. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1). For the classification accuracies on the training set, the error bars are very small and are partly covered by the markers.*

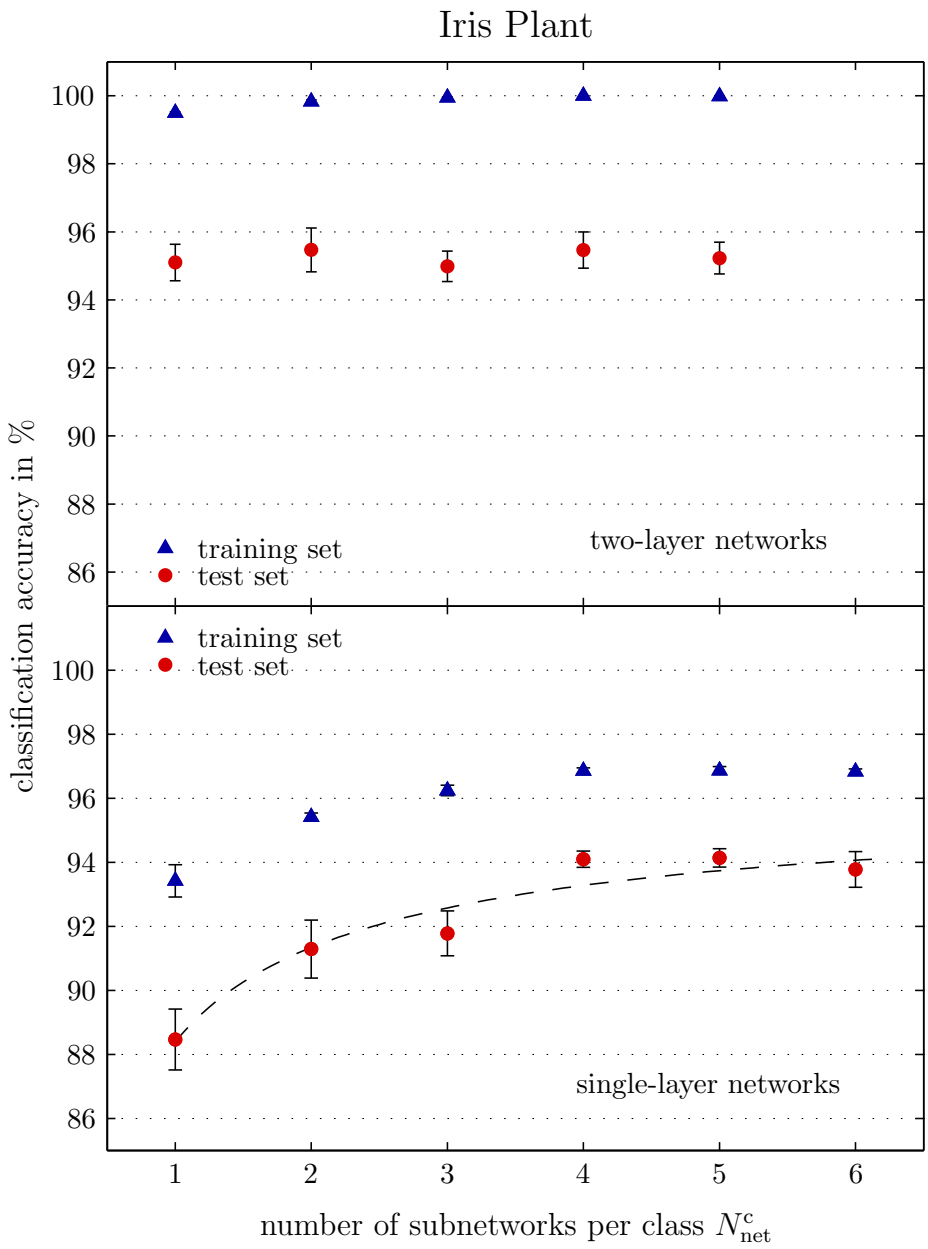


Figure C.9: *The achieved classification accuracies on the training sets A_{t} and test sets A_{g} as a function of the number $N_{\mathrm{net}}^{\mathrm{c}}$ of subnetworks per class for the diabetes problem. The upper diagram presents the results of the two-layer networks, the lower diagram refers to single-layer perceptrons. For two-layer networks, an increase in the number of subnetworks per class does not yield a measurable improvement in generalization. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1). For the classification accuracies on the training set, the error bars are very small and are partly covered by the markers.*

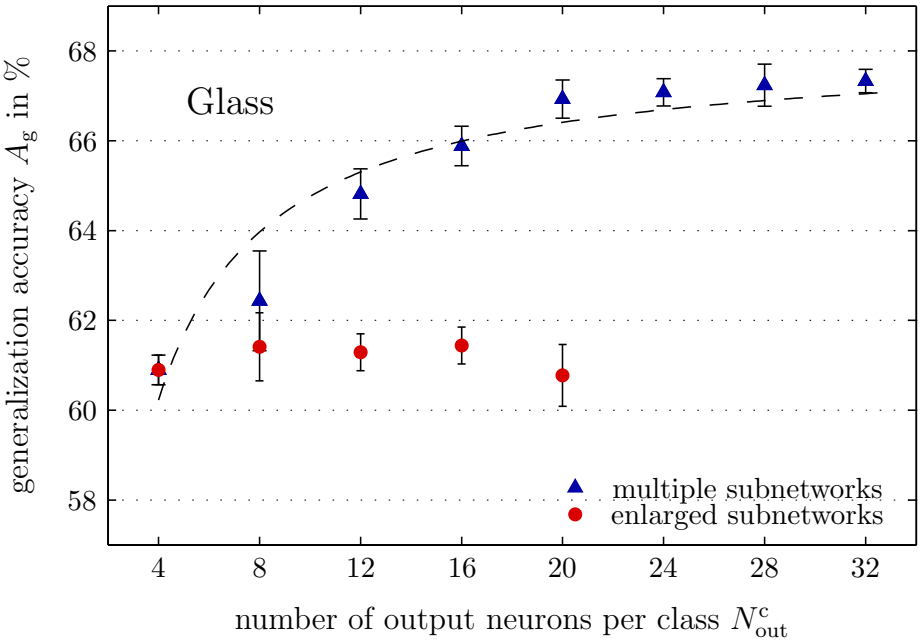


Figure C.10: *The generalization rates A_g on the glass data set that are achieved by networks with increased subnetwork size are compared to the obtained accuracies of networks that contain multiple subnetworks per class (see figure 9.5). Only single-layer networks are considered. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1).*

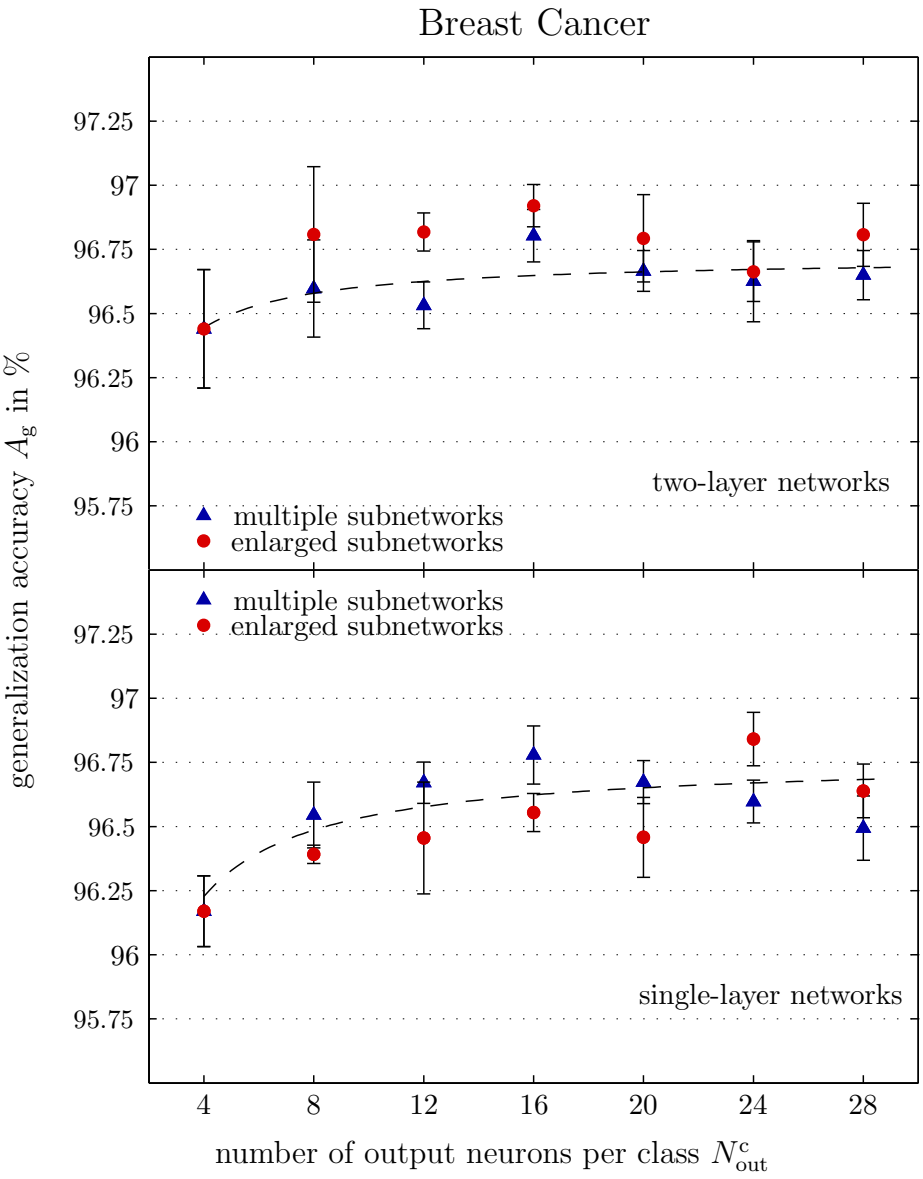


Figure C.11: *The generalization rates A_g on the breast cancer set that are achieved by networks with increased subnetwork size are compared to the obtained accuracies of networks that contain multiple subnetworks per class (see figure 9.8). The upper part refers to two-layer architectures, the lower half represents singe-layer networks. The shown dashed lines are intended as mere guides to the eye (see section 9.4.1). They are fitted to the data that is obtained with the networks that feature multiple subnetworks per class (triangles).*

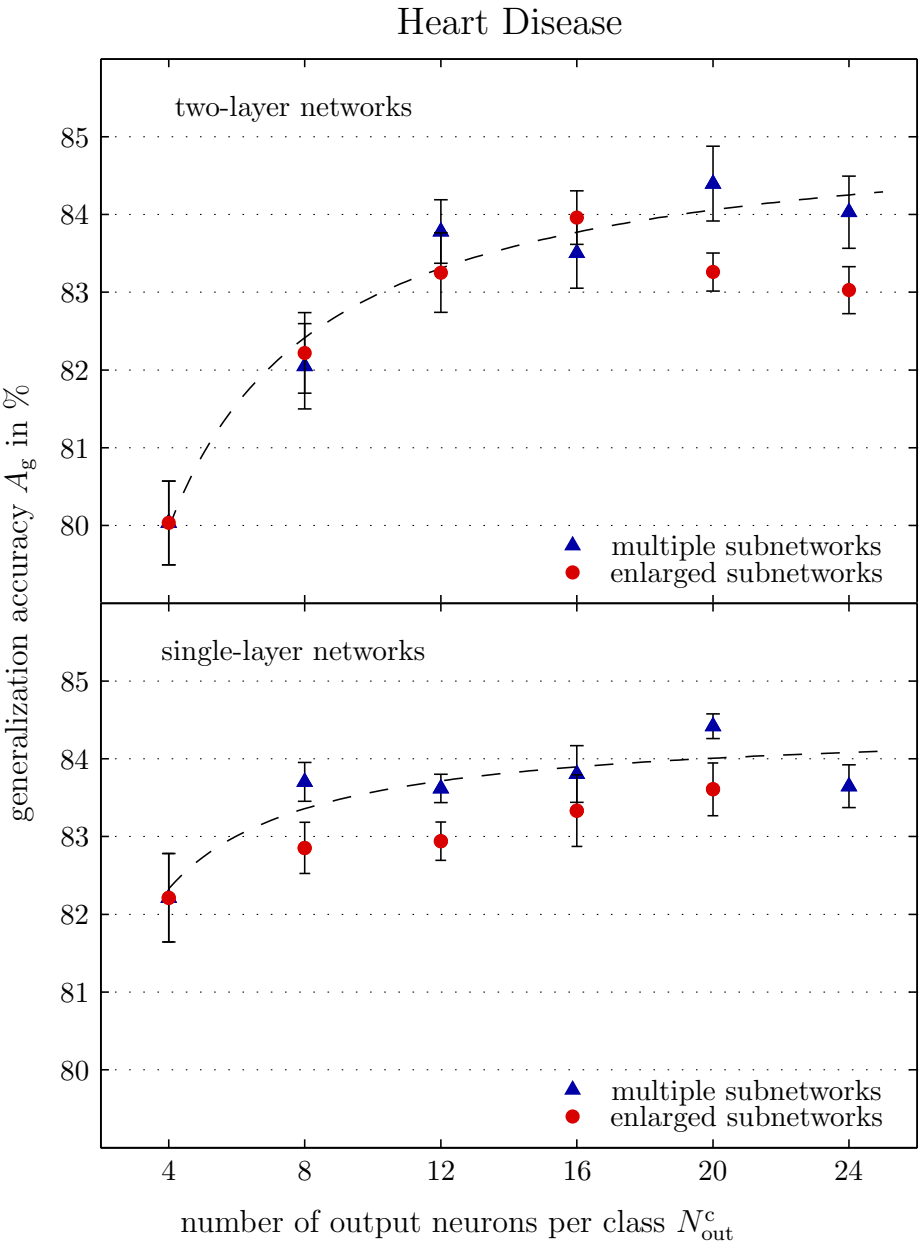


Figure C.12: *The generalization rates A_g on the heart disease data set that are achieved by networks with increased subnetwork size are compared to the obtained accuracies of networks that contain multiple subnetworks per class (see figure 9.9). The upper part refers to two-layer architectures, the lower half represents singe-layer networks. The shown dashed lines are intended as mere guides to the eye (see section 9.4.1). They are fitted to the data that is obtained with the networks that feature multiple subnetworks per class (triangles).*

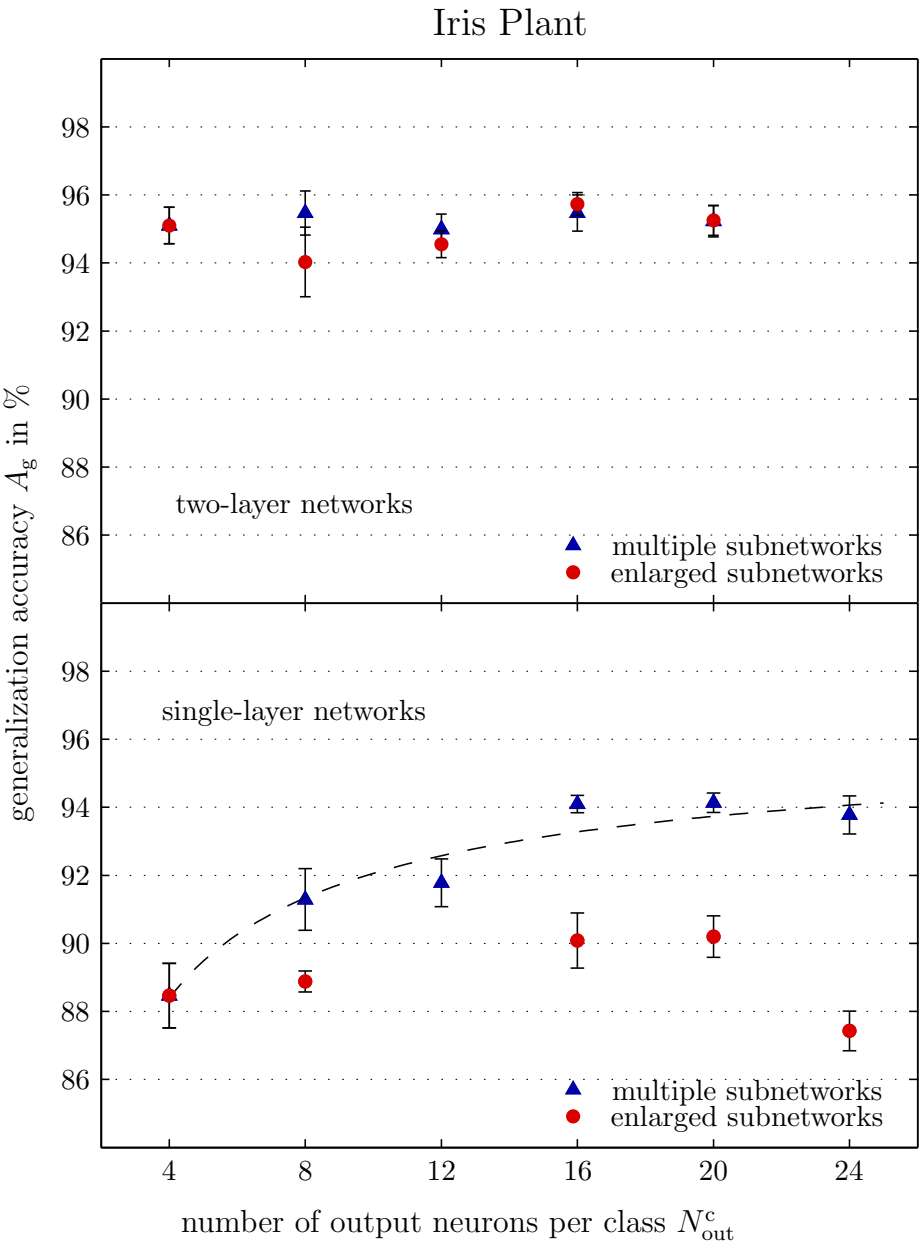


Figure C.13: *The generalization rates A_g on the iris plant data set that are achieved by networks with increased subnetwork size are compared to the obtained accuracies of networks that contain multiple subnetworks per class (see figure 9.9). The upper part refers to two-layer architectures, the lower half represents singe-layer networks. The shown dashed line is intended as a mere guide to the eye (see section 9.4.1).*

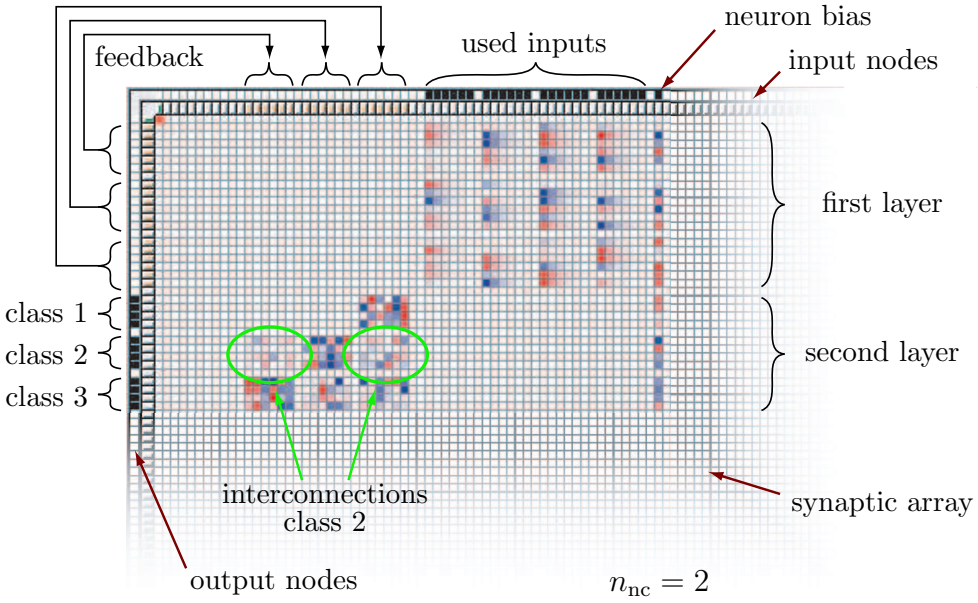


Figure C.14: *Schematical illustration of the upper left corner of the upper left block of a HAGEN ASIC, taken from a screenshot of the HANNEE software (see chapter 7). The input nodes are on top, the neurons are to the left, and the array of colored squares represents the synaptic array (compare also figures 5.2, 5.5, and 8.2 b)). Positive weight values are a shade of red, negative weights are blue, and white squares correspond to deactivated synapses with zero weight. The presented network has a two-layer architecture and is trained for the iris problem. It contains one subnetwork for each class (compare figure 10.3). In the case of the first class, no interconnections are added during the second stage, since the unconnected subnetwork already achieves a classification accuracy of 100 %. In general, the additional interconnections tend to have lower absolute weight values than those trained in the first stage.*

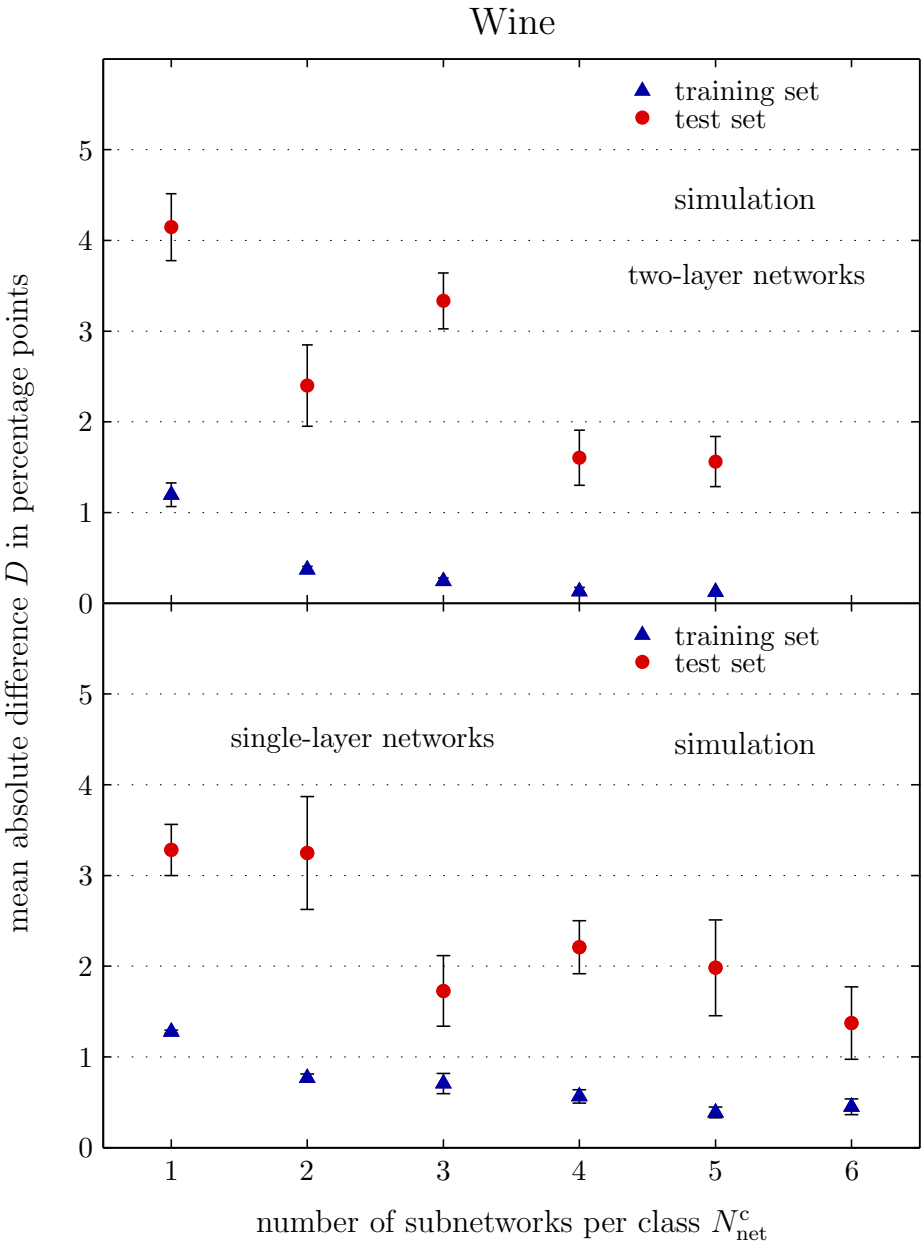


Figure C.15: *The networks that have been trained for the wine benchmark during the experiments presented in section 9.4 are implemented in software using a dedicated chip simulation that is based on the ideal network model defined by equation 5.4. The mean absolute differences in classification accuracy on the training and test sets between the original and the simulated networks (see also equation 10.2) are shown as a function of the number of subnetworks per class $N_{\text{net}}^{\text{c}}$. The results are very similar to those that are obtained when the networks are transferred between different chips (see figure 10.7).*

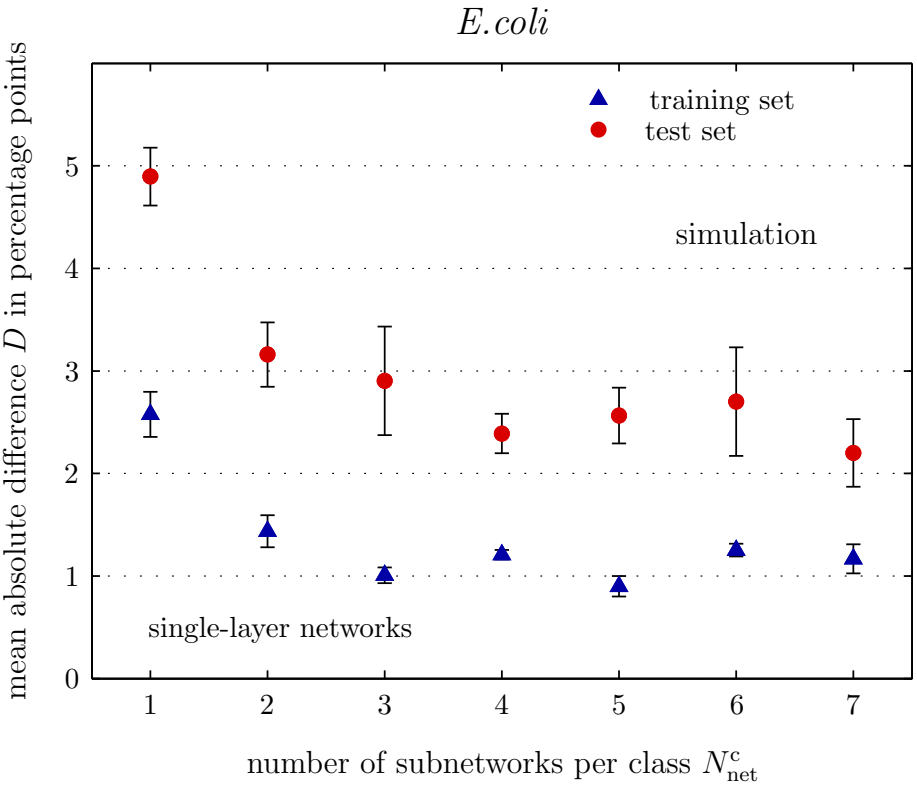


Figure C.16: *The networks that have been trained for the* E.coli *benchmark during the experiments presented in section 9.4 are implemented in software using a dedicated chip simulation that is based on the ideal network model defined by equation 5.4. The mean absolute differences in classification accuracy on the training and test sets between the original and the simulated networks (see also equation 10.2) are shown as a function of the number of subnetworks per class* N^{c}_{net}*. The results are very similar to those that are obtained when the networks are transferred between different chips (see figure 10.8).*

Bibliography

[1] ACE: The Adaptive Communication Environment. Distributed Object Computing (DOC) group, Vanderbilt University, Nashville, Washington University, and University of California, Irvine, `http://www.cs.wustl.edu/ schmidt/ACE.html`.

[2] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

[3] S. Aeberhard, D. Coomans, and O. de Vel. The performance of statistical pattern recognition methods in high dimensional settings. Technical Report 4, James Cook University, Australia, 1993.

[4] P. E. Allen and D. R. Holberg. *CMOS Analog Circuit Design*. Oxford University Press, Inc., 198 Madison Avenue, New York, 2002.

[5] ANSI/TIA/EIA-644. *Electrical Characteristics of Low Voltage Differential Signalling (LVDS)*, March 1996.

[6] P. Auer, H. Burgsteiner, and W. Maass. Reducing communication for distributed learning in neural networks. In *Proceedings of the International Conference on Artificial Neural Networks ICANN'02*, pages 123–128. Springer Verlag, 2002.

[7] Austria Mikro Systeme International AG. *austriamicrosystems* AG. Schloss Premstätten, A-8141 Unterpremstätten, Austria, `http://www.austriamicrosystems.com`.

[8] T. Bäck. Optimal mutation rates in genetic search. In Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 2–8. Morgan Kauffman, June 1993.

[9] T. Bäck, F. Hoffmeister, and H. Schwefel. A survey of evolution strategies. In R. K. Belew and L. B. booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 2–9, San Diego, CA, July 1991. Morgan Kaufmann.

[10] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In J. J. Grefenstette, editor, *Proceedings of the 2nd International Conference on Genetic Algorithms*, pages 14–21, Hillsdale, New Jersey, 1987. Lawrence Erlbaum.

[11] E. B. Baum. On the capabilities of multilayer perceptrons. *Journal of Complexity*, 4:193–215, 1988.

[12] E. B. Baum and D. Haussler. What size nets gives valid generalization? *Neural Computation*, 1:151–160, 1989.

[13] J. Baxter. The evolution of learning algorithms for artificial neural networks. *Complex Systems*, pages 313–326, 1992.

[14] J. Becker. Ein FPGA-basiertes Testsystem für gemischt analog/digitale ASICs. Diploma thesis (german), University of Heidelberg, HD-KIP-01-11, 2001.

[15] V. Beiu. Digitial integrated circuit implementations. In E. Fiesler and R. Beale, editors, *The Handbook of Neural Computation*, New York, January 1997. Institute of Physics Publishing and Oxford University Publishing.

[16] V. Beiu. VLSI implementations of threshold logic – a comprehensive survey. *IEEE Transactions on Neural Networks*, 14(5):1217–1243, 2003.

[17] R. K. Belew, J. McInerney, and N. M. Schraudolph. Evolving networks: Using the genetic algorithm with connectionist learning. Technical Report CS90-174 (Revised), Computer Science & Engr. Dept. (C-104), Univ. of California, San Diego, La Jolla, CA 92093, USA, February 1991.

[18] N. Bertschinger and T. Natschläger. Real-time computation at the edge of chaos in recurrent neural networks. *Neural Computation*, 16(7):1413 – 1436, July 2004.

[19] B. Bhanu and Y. Lin. Learning composite operators for object detection. In W. Langdon et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2002*, pages 1003–1010. Morgan Kaufmann Publishers, July 2002.

[20] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Walton Street, Oxford, 1995.

[21] T. Blickle and L. Thiele. A comparison of selection schemes used in genetic algorithms. Technical Report 11, Computer Engineering and Communications Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Gloriastrasse 35, 8092 Zurich, Switzerland, 1995.

[22] H. D. Block. The perceptron: a model for brain functioning. *Reviews of Modern Physics*, 34:123–135, 1962.

[23] E. K. Blum and L. K. Li. Approximation theory and feedforward networks. *Neural Networks*, 4(4):511–515, 1991.

[24] S. Bornholdt and D. Graudez. General asymmetric neural networks and structure design by genetic algorithms. *Neural Networks*, 5(1):327–334, 1992.

[25] Boser et al. An analog neural network processor with programmable topology. *IEEE Journal of Solid-State Circuits*, pages 2017–2025, December 1991.

[26] J. Branke. Evolutionary algorithms in neural network design and training – A review. In J. T. Alander, editor, *Proceedings of the First Nordic Workshop on Genetic Algorithms and their Applications (1NWGA)*, number 95-1, pages 145–163, Vaasa, Finnland, 1995.

[27] A. Breidenassel, K. Meier, and J. Schemmel. A flexible scheme for adaptive integration time control. In *Proceedings of the Third IEEE Conference on Sensors*, pages 280–283, Vienna, October 2004.

[28] D. Brüderle. Implementing spike-based computation on a hardware perceptron. Diploma thesis, University of Heidelberg, HD-KIP-04-16, 2004.

[29] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[30] A. H. Cannon, L. J. Cowen, and C. E. Priebe. Approximate distance classification. In *Proceedings of the 1998 Symposium on the Interface between Computer Science and Statistics*, number 30-1, pages 544–549, 1998.

[31] B. Carr, W. Hart, N. Krasnogor, J. Hirst, E. Burke, and J. Smith. Alignment of protein structures with a memetic evolutionary algorithm. In W. Langdon et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2002*, pages 1027–1034. Morgan Kaufmann Publishers, July 2002.

[32] J. P. Cater. Successfully using peak learning rates of 10 (and greater) in backpropagation networks with the heuristic learning algorithm. In M. Caudill and C. Butler, editors, *Proceedings of the IEEE First International Conference on Neural Networks*, volume II, pages 645–651, San Diego, 1987. IEEE.

[33] T. P. Caudell and C. P. Dolan. Parametric connectivity: Training of constrained networks using genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 370–374. Morgan Kaufmann, 1989.

[34] G. Cauwenberghs. A fast stochastic error-descent algorithm for supervised learning and optimization. *Advances in Neural Information Processing Systems*, 5:244–251, 1993.

[35] G. Cauwenberghs. Learning on silicon: A survey. In G. Cauwenberghs and M. A. Bayoumi, editors, *Learning on Silicon: Adaptive VLSI Neural Systems*, pages 1–29, Norwell, MA, 1999. Kluwer Academic Publisher.

[36] D. J. Chalmers. The evolution of learning: An experiment in genetic connectionism. In D. S. Touretzky, J. L. Elman, and G. E. Hinten, editors, *Proceedings of the 1990 Connectionist Models Summer School*, pages 81–90, San Mateo, CA, 1990. Morgan Kaufmann.

[37] C. A. Coello Coello, D. A. Van Veldhuizen, and G. B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer Academic/Plenum Publishers, New York, 2002.

[38] J. P. Cohoon, S. U. Hedge, W. N. Martin, and D. Richards. Punctuated equilibria: A parallel genetic algorithm. In J. J. Grefenstette, editor, *Proceedings of the 2nd International Conference on Genetic Algorithms*, pages 148–154, Hillsdale, New Jersey, 1987. Lawrence Erlbaum.

[39] J. P. Cohoon, W. N. Martin, and D. S. Richards. Genetic algorithms and punctuated equilibria in VLSI. In H.-P. Schwefel and R. Männer, editors, *Proceedings of the 1st International Conference on Parallel Problem Solving from Nature*, volume 496, pages 134–141. Springer Verlag, 1991.

[40] C. Cortez and V. Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995.

[41] T. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Trans. on Electronic Computers*, EC-14:326–334, 1965.

[42] C. R. Darwin. *On the Origin of Species*. John Murray, London, 1859.

[43] L. Davis. Adapting operator probabilities in genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 61–69. Morgan Kaufmann, 1989.

[44] P. Dayan and L. F. Abott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. The MIT press, Cambride, Massachusetts, London, England, 2001.

[45] K. Deb and D. E. Goldberg. An investigation of niche and species formation in genetic function optimization. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 42–50, San Francisco, 1989. Morgan Kaufmann.

[46] K. DeJong. The analysis and behaviour of a class of genetic adaptive systems. *PhD thesis*, 1975.

[47] A. Dembo and T. Kailath. Model-free distributed learning. *IEEE Transactions on Neural Networks*, 1(1):58–70, 1990.

[48] B. Denby, P. Garda, B. Dranado, C. Kiesling, J.-C. Prévotet, and A. Wassatsch. Fast triggering in high-energy physics experiments using hardware neural network. *IEEE Transactions on Neural Networks*, 14(5):1010–1026, 2003.

[49] Design Automation Standards Committee of the IEEE Computer Society, New York. *VHDL Language Reference Manual, IEEE Std. 1076.1*, 1997.

[50] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer Verlag, Berlin, Heidelberg, New York, 2003.

[51] L. J. Eshelmann, R. A. Caruana, and J. D. Schaffer. Biases in crossover landscape. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 10–19. Morgan Kaufmann, 1989.

[52] J. Fieres, A. Grübl, S. Philipp, K. Meier, J. Schemmel, and F. Schürmann. A platform for parallel operation of VLSI neural networks. In *Proc. of the 2004 Brain Inspired Cognitive Systems Conference (BICS2004)*, University of Stirling, Scotland, UK, 2004.

[53] G. Fischer. *Lineare Algebra*. Friedrich Vieweg und Sohn Verlagsgesellschaft mbH, Braunschweig, Wiesbaden, 1986.

[54] T. Fließbach. *Statistische Physik*. Spektrum Akademischer Verlag GmbH, Heidelberg, 1995.

[55] B. Flower and M. Jabri. Summed weight neuron perturbation: An $\mathcal{O}(n)$ improvement over weight perturbation. *Advances in Neural Information Processing Systems*, 5:212–219, 1993.

[56] D. B. Fogel. *Evolutionary Computation – Toward a New Philosophy of Machine Intelligence*. IEEE Press, New York, 1995.

[57] D. B. Fogel, editor. *Evolutionary Computation: the Fossil Record*. IEEE Press, Piscataway, NJ, 1998.

[58] D. B. Fogel, L. J. Fogel, and V. W. Porto. Evolving neural networks. *Biological Cybernetics*, 63:487–493, 1990.

[59] O. Forster. *Analysis 2*. Friedrich Vieweg und Sohn Verlagsgesellschaft mbH, Braunschweig, Wiesbaden, 1984.

[60] O. Forster. *Analysis 3*. Friedrich Vieweg und Sohn Verlagsgesellschaft mbH, Braunschweig, Wiesbaden, 1992.

[61] O. Forster. *Analysis 1*. Friedrich Vieweg und Sohn Verlagsgesellschaft mbH, Braunschweig, Wiesbaden, 2000.

[62] M. Frean. The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2:198–209, 1990.

[63] K. Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural Networks*, 1:119–130, 1988.

[64] K. Fukushima, S. Miyake, and T. Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13:826–834, 1983.

[65] G. Fung and O. L. Mangasarian. Proximal support vector machine classifiers. In F. Provost and R. Srikant, editors, *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 77–86, 2001.

[66] D. J. Futuyma. *Evolutionary Biology*. Sinauer Associates, Inc., Sunderland, Massachusetts, USA, 1986.

[67] The GNU Compiler Collection. Free Software Foundation, Inc., 59 Temple Place, Boston, MA, USA, `http://gcc.gnu.org/`.

[68] R. Genov. Kerneltron: Support vector "machine" in silicon. *IEEE Transactions on Neural Networks*, 14(5):1426–1434, 2003.

[69] W. Gerstner and W. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[70] D. E. Goldberg. Genetic algorithms with sharing for multimodal function approximation. In J. J. Grefenstette, editor, *Proceedings of the 2nd International Conference on Genetic Algorithms*, pages 41–49, Hillsdale, New Jersey, 1987. Lawrence Erlbaum.

[71] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley Longman Inc., 1989.

[72] M. Gorges-Schleuter. ASPARAGOS an asynchronous parallel genetic optimization strategy. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 422–427, San Francisco, 1989. Morgan Kaufmann.

[73] P. M. Granitto, H. Navone, and H. A. Ceccatto. A stepwise algorithm for construction of neural network ensembles. In *VII Argentine Congress of Computer Science*, Calafate, Argentina, 2001.

[74] G. W. Greenwood. Training partially recurrent neural networks using evolutionary strategies. *IEEE Transactions on Speech and Audio Processing*, 5(2):192–194, 1997.

[75] J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-16(1):122–128, 1986.

[76] J. J. Grefenstette. Deception considered harmful. In L. D. Whitley, editor, *Foundations of Genetic Algorithms*, volume 2, pages 75–91, 1993.

[77] J. J. Grefenstette and J. E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 20–27. Morgan Kaufmann, 1989.

[78] A. Grübl. Eine FPGA-basierte platform für neuronala netze. Diploma thesis (german), University of Heidelberg, HD-KIP-03-2, 2003.

[79] G. Han and E. Sánchez. CMOS transconductance multipliers: A tutorial. *IEEE Transactions on Circuits and Systems II; Analog and Digital Signal Processing*, 45(12):1550–1563, 1998.

[80] P. J. B. Hancock. Genetic algorithms and permutation problems: A comparison of recombination operators for neural net structure specification. In D. Whitley, editor, *Proceedings of the IEEE Workshop on Combinations of Genetic Algorithms and Neural Network*, pages 108–121. IEEE Press, 1992.

[81] P. J. B. Hancock. Recombination operators for the design of neural nets by genetic algorithm. In R. Männer and B. Manderick, editors, *Proceedings of the Conference on Parallel Problem Solving from Nature*, volume 2, pages 441–450. Elsevier Science Publishers B.V., 1992.

[82] S. A. Harp, T. Samad, and A. Guha. Towards the genetic synthesis of neural networks. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 360–369. Morgan Kaufmann, 1989.

[83] J. C. Hay, F. C. Martin, and C. W. Wightman. The MARK I perceptron - design and performance. In *IRE National Convention Record*, volume 2, pages 78–87, 1960.

[84] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, Upper Saddle River, New Jersey, 1999.

[85] D. O. Hebb. *The Organization of Behaviour*. Wiley, New York, 1949.

[86] R. Hecht-Nielsen. *Neurocomputing*. Addison-Wesley, 1989.

[87] J. N. H. Heemskerk. Overview of neural hardware. In: Neurocomputers for Brain-Style Processing. Design, Implementation and Application, *PhD thesis*, 1995.

[88] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the theory of neural computation*. Addison Wesley Publishing Company, Redwood City, CA, 1991.

[89] J. Hesser and R. Männer. Towards an optimal mutation probability for genetic algorithms. In H.-P. Schwefel and R. Männer, editors, *Proceedings of the 1st International Conference on Parallel Problem Solving from Nature*, volume 496, pages 23–32. Springer Verlag, 1991.

[90] S. Hettich and S. D. Bay. The UCI KDD archive. University of California, Department of Information and Computer Science, Irvine, USA, `http://kdd.ics.uci.edu`, 1999.

[91] R. Hinterding, Z. Michalewicz, and A. E. Eiben. Adaptation in evolutionary computation: A survey. In *IEEECEP: Proceedings of The IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, 1997.

[92] A. Hohmann and W. Hielscher. *Lehrbuch der Zahntechnik, Band I.* Quintessenz Verlags-GmbH, Berlin, 2001.

[93] S. Hohmann, J. Schemmel, F. Schürmann, and K. Meier. Exploring the parameter space of a genetic algorithm for training an analog neural network. In W. e. a. Langdon, editor, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2002*, pages 375–382. Morgan Kaufmann Publishers, July 2002.

[94] S. Hohmann, J. Fieres, K. Meier, J. Schemmel, T. Schmitz, and F. Schürmann. Training fast mixed-signal neural networks for data classification. In *Proceedings of the 2004 International Joint Conference on Neural Networks (IJCNN'04)*, pages 2647–2652. IEEE Press, 2004.

[95] S. G. Hohmann, J. Schemmel, F. Schürmann, and K. Meier. Predicting protein cellular localization sites with a hardware analog neural network. In *Proceedings of the Int. Joint Conf. on Neural Networks*, pages 381–386. IEEE Press, July 2003.

[96] J. H. Holland. Genetic algorithms and the optimal allocation of trials. *SIAM J. of Computing*, 2:88–105, 1973.

[97] M. Holler. VLSI implementation of learning and memory systems: A review. *Advances in Neural Information Processing Systems*, 3, 1991.

[98] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554–2558, 1982.

[99] J. J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences*, 81:3088–3092, 1984.

[100] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 4(2):359–366, 1989.

[101] P. Horton and K. Nakai. Better prediction of protein cellular localization sites with the k nearest neighbors classifier. In *Proceedings of the 5th International Conference on Intelligent Systems in Molecular Biology*, pages 147–152. AAAIPress, 1997.

[102] J. Hromkovič. *Algorithmics for Hard Problems.* Springer Verlag, Berlin, Heidelberg, New York, 2004.

[103] P. Husbands. Distributed co-evolutionary genetic algorithms for multi-criteria and multi-constraint optimisation. In T. C. Fogarty, editor, *Evolutionary Computing: Proceedings of the AISB workshop*, pages 150–165, Berlin, Heidelberg, New York, 1994. Springer Verlag.

[104] Intel Compiler for Linux. Intel Inc., `http://www.intel.com/software/products/compilers/clin/`.

[105] M. M. Islam, X. Yao, and K. Murase. A constructive algorithm for training cooperative neural network ensembles. *IEEE Transactions on Neural Networks*, 14(4):820–834, 2003.

[106] M. Jabri and B. Flower. Weight perturbation: An optimal architecture and learning technique for analog VLSI feedforward and recurrent multilayer networks. *IEEE Transactions on Neural Networks*, 3(1):154–157, 1992.

[107] R. A. Jacobs, M. I. Jordan, and A. G. Barto. Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks. *Cognitive Science*, 15:219–250, 1991.

[108] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3:79–87, 1991.

[109] R. Jacobs. Increased rates of convergence through learning rate adaption. *Neural Networks*, 1:295–307, 1988.

[110] H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks. Technical Report GMD Report 148, German National Research Center for Information Technology, 2001.

[111] D. Jimenez and N. Walsh. Dynamically weighted ensemble neural networks for classification. In *Proceedings of the 1998 International Joint Conference on Neural Networks (IJCNN)*, 1998.

[112] T. Joachims. Text categorization with support vector machines: learning with many relevant features. In C. Nédellec and C. Rouveirol, editors, *Proceedings of ECML-98, 10th European Conference on Machine Learning*, pages 137–142, Chemnitz, DE, 1998. Springer Verlag, Heidelberg, DE.

[113] L. K. Jones. A simple lemma on greedy approximation in hilbert space and convergence rates for projection pursuit regression and neural network training. *Annals of Statistics*, 20(1):608–613, 1992.

[114] Jungo Ltd., Netanya. *Windriver 6 User's Manual*, 2003.

[115] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall International Inc., London, 1978.

[116] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[117] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4(4):461–476, 1990.

[118] C. Koch. *Biophysics of Computation: Information Processing in Single Neurons*. Oxford University Press, 1999.

[119] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *The 1995 International Joint Conference on Artificial Intelligence IJCAI*, pages 1137–1145, Montreal, Quebec, Canada, August 1995.

[120] T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59–69, 1982.

[121] J. R. Koza, F. H. Bennet III, D. Andre, and M. A. Keane. *Genetic Programming III - Darwinian Invention and Problem Solving*. Morgan Kaufmann Publishers, San Francisko, CA, USA, 1999.

[122] A. Kramer. Array-based analog computation. *IEEE Micro*, pages 20–29, October 1996.

[123] T. Kristensen and R. Patel. Classification of eukariotic and prokariotic cells by a backpropagation network. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1718–1723. IEEE Press, 2003.

[124] A. Krogh and J. Vedelsby. Neural network ensembles, cross validation and active learning. *Advances in Neural Information Processing Systems*, 7:299–314, 1995.

[125] J. Langeheine, K. Meier, and J. Schemmel. Intrinsic Evolution of Quasi DC Solutions for Transistor Level Analog Electronic Circuits Using a CMOS FPTA chip. In *Proceedings of the 2002 NASA/DoD Conference an Evolvable Hardware*, 2002.

[126] C. Lindsey and T. Lindblad. Survey of neural network hardware. *SPIE*, 1995(2492):1194–1205, April 1995.

[127] R. P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4(2):4–22, 1987.

[128] R. Lohmann. Application of evolution strategy in parallel populations. In H.-P. Schwefel and R. Männer, editors, *Proceedings of the 1st International Conference on Parallel Problem Solving from Nature*, volume 496, pages 198–208. Springer Verlag, 1991.

[129] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.

[130] R. Maclin and J. W. Shavlik. Combining the predictions of multiple classifiers: Using competitive learning to initialize neural networks. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 524–530, Montreal, Canada, 1995.

[131] S. W. Mahfoud. Niching methods for genetic algorithms. *PhD thesis*, 1995.

[132] B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 428–433, San Francisco, 1989. Morgan Kaufmann.

[133] O. L. Mangasarian and W. H. Wolberg. Cancer diagnosis via linear programming. *SIAM News*, 23(5):1–18, September 1990.

[134] MATLAB. version 6, release 12.1, The Mathworks Inc., 3 Apple Hill Drive, Natick, MA, USA, `http://www.mathworks.com/products/matlab/`.

[135] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, pages 127–147, 1943.

[136] M. R. J. McQuoid. Neural ensembles: Simultaneous recognition of multiple 2-D visual objects. *Neural Networks*, 6:907–917, 1993.

[137] C. A. Mead. *Analog VLSI and Neural Systems*. Addison Wesley, Reading, MA, 1989.

[138] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, and A. H. Theller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[139] M. Mézard and J.-P. Nadal. Learning inf feedforward layered networks: the tiling algorithm. *Journal of Physics A*, 22:2191–2203, 1989.

[140] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programms*. Springer Verlag, Berlin, Heidelberg, New York, 1999.

[141] G. F. Miller and P. M. Todd. Designing neural networks using genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384. Morgan Kaufmann, 1989.

[142] M. Minsky. Neural nets and the brain model problem. *PhD thesis*, 1954.

[143] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[144] P. Moerland and E. Fiesler. Neural network adaptions to hardware implementations. In E. Fiesler and R. Beale, editors, *The Handbook of Neural Computation*, New York, January 1997. Institute of Physics Publishing and Oxford University Publishing.

[145] D. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 762–767, San Mateo, CA, 1989. Morgan Kaufmann.

[146] J. Moody and C. J. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1:281–294, 1989.

[147] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[148] D. Moriarty and R. Miikkulainen. Discovering complex othello strategies through evolutionary neural networks. *Connection Science*, 7(3/4):195–210, 1995.

[149] Y. S. Mostafa and J. St. Jaques. Information capacity of the hopfield model. *IEEE Transactions on Information Theory*, IT-31(4):461–464, 1985.

[150] H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 416–421, San Francisco, 1989. Morgan Kaufmann.

[151] D. Niedenzu. Aufbau eines binären Neocognitrons. Diploma thesis (german), University of Heidelberg, HD-KIP-03-11, 2003.

[152] A. R. Omondi. Neurocomputers: A dead end ? *International Journal of Neural Systems*, 10(6):475–481, 2000.

[153] D. W. Opitz and R. Maclin. Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:337–353, 1999.

[154] D. W. Opitz and J. W. Shavlik. Actively searching for an effective neural network ensemble. *Connection Science*, 8(3/4):337–353, 1996.

[155] M. Page-Jones. *Fundamentals of Object-Oriented Design in UML*. Addison Wesley, Reading, MA, 2000.

[156] R. Parekh, J. Yang, and V. Honavar. Mupstart - a constructive neural network learning algorithm for multi-category pattern classification. In *Proceedings of the IEEE International Conference on Neural Networks (ICNN'97)*, pages 1924–1929, Houston, TX, USA, 1997.

[157] R. Parekh, J. Yang, and V. Honavar. Pruning strategies for the mtiling constructive learning algorithm. In *Proceedings of the IEEE International Conference on Neural Networks (ICNN'97)*, volume 3, pages 1960 – 1965, Houston, TX, USA, 1997.

[158] E. Pasero and M. Perri. Hw-sw codesign of a flexible neural controller through a fpga-based neural network programmed in vhdl. In *Proceedings of the International Joint Conference on Neural Networks IJCNN'04*, pages 3161–3166. IEEE Press, 2004.

[159] G. O. Penokie. *Working Draft SCSI Parallel Interface-2 (SPI-2).* American National Standard of Accredited Standards Committee NCITS, Washington, DC, revision 20b edition, April 1998.

[160] M. P. Peronne and L. N. Cooper. When networks disagree: Ensemble methods for hybrid neural networks. *Artificial Neural Networks for Speech and Vision*, pages 126–142, 1993.

[161] C. B. Pettey, M. R. Leuze, and G. J. J. A parallel genetic algorithm. In J. J. Grefenstette, editor, *Proceedings of the 2nd International Conference on Genetic Algorithms*, pages 155–161, Hillsdale, New Jersey, 1987. Lawrence Erlbaum.

[162] PLX Technology, Inc., Sunnyvale. *PLX 9054 Data Book*, version 2.1 edition, January 2000.

[163] L. Prechelt. Proben1 — A set of neural network benchmark problems and benchmarking rules. Technical Report 21/94, 38 pages, Fakultät für Informatik, Universität Karlsruhe, 1994.

[164] A. Prügel-Bennet and A. Rogers. Modelling GA dynamics. *Proceedings of Theoretical Aspects of Evolutionary Computation*, pages 59–86, 2001.

[165] A. Prügel-Bennet and J. L. Shapiro. Analysis of genetic algorithms using statistical mechanics. *Physical Review Letters*, 72(9):1305–1309, February 1994.

[166] J. Pujol and R. Poli. Evolving neural networks using a dual representation with a combined crossover operator. In *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC)*, pages 416–421, 1998.

[167] E. T. Ray. *Learning XML.* O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA, 2001.

[168] L. Reyneri. Implementation issues of neuro-fuzzy hardware: Going toward HW/SW codesign. *IEEE Transactions on Neural Networks*, 14(1):176–194, January 2003.

[169] P. Rojas. *Theorie der neuronalen Netze: Eine systematische Einführung.* Springer Verlag, Berlin, Heidelberg, New York, 1996.

[170] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[171] F. Rosenblatt. Perceptron simulation experiments. In *Proceedings of the IRE*, pages 301–309, 1960.

[172] G. Rudolph. Global optimization by means of distributed evolution strategies. In H.-P. Schwefel and R. Männer, editors, *Proceedings of the 1st International Conference on Parallel Problem Solving from Nature*, volume 496, pages 209–213. Springer Verlag, 1991.

[173] G. Rudolph. Convergence of evolutionary algorithms in general search spaces. In *Proceedings of the IEEE Conference on Evolutionary Computation*, pages 50–54, Piscataway, NJ, 1996. IEEE Press.

[174] G. Rudolph. *Convergence Properties of Evolutionary Algorithms*. Verlag Dr. Kovač, Hamburg, 1997.

[175] D. E. Rumelhart, G. E. Hinton, and W. R.J. Learning internal representations by error propagation. *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, I:318–362, 1986.

[176] D. E. Rumelhart, G. E. Hinton, and W. R.J. Learning representations of back-propagation errors. *Nature*, 323:533–536, 1986.

[177] J. D. Schaffer, R. A. Caruana, and L. J. Eshelmann. Using genetic search to exploit the emergent behavior of neural networks. *Physica D*, 42:244–248, 1990.

[178] J. Schemmel. personal communication, 2005.

[179] J. Schemmel, S. Hohmann, K. Meier, and F. Schürmann. A mixed-mode analog neural network using current-steering synapses. *Analog Integrated Circuits and Signal Processing*, 38(2-3):233–244, 2004.

[180] J. Schemmel, F. Schürmann, S. Hohmann, and K. Meier. An integrated mixed-mode neural network architecture for megasynapse ANNs. In *Proceedings of the 2002 International Joint Conference on Neural Networks IJCNN'02*, pages 2704–2710. IEEE Press, 2002.

[181] T. Schmitz. personal communication, 2005.

[182] T. Schmitz, S. Hohmann, K. Meier, J. Schemmel, and F. Schürmann. Speeding up Hardware Evolution: A Coprocessor for Evolutionary Algorithms. In A. M. Tyrrell, P. C. Haddow, and J. Torresen, editors, *Proceedings of the 5th International Conference on Evolvable Systems ICES 2003*, pages 274–285. Springer Verlag, 2003.

[183] R. Schüffny, A. Graupner, and J. Schreiter. Hardware for neural networks. In *Proceedings of the 4th International Workshop on Neural Networks in Applications*, pages 1–6, 1999.

[184] F. Schürmann, K. Meier, and J. Schemmel. Edge of Chaos Computation in Mixed Mode VLSI – "A Hard Liquid". In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, Cambride, 2005. MIT Press.

[185] F. Schürmann. *PhD thesis*, University of Heidelberg, in preparation, 2005.

[186] F. Schürmann, S. G. Hohmann, K. Meier, and J. Schemmel. Interfacing binary networks to multi-valued signals. In *Supplementary Proceedings of the Joint International Conference ICANN/ICONIP*, pages 430–433. IEEE Press, 2003.

[187] T. J. Sejnowski and C. R. Rosenberg. NETtalk: a parallel network that learns to read aloud. Technical Report JHU/EECS-86/01, John Hopkins University, Electrical Engineering and Computer Science, 1986.

[188] R. S. Sexton, R. E. Dorsez, and J. J. D. Toward global optimization of neural networks: A comparison of the genetic algorithm and backpropagation. *Decision Support Systems*, 22(2):171–185, 1998.

[189] A. J. C. Sharkey. On combining artificial neural nets. *Connection Science*, 8(3/4):299–314, 1996.

[190] A. J. C. Sharkey and N. E. Sharkey. Combining diverse neural nets. *Knowlegde Engineering Review*, 12(3):1–17, 1997.

[191] A. A. Siddiqi and S. M. Lucas. A comparison of matrix rewriting versus direct encoding for evolving neural networks. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation (ICEC'98)*, pages 392–397, Piscataway, NJ, 1998. IEEE Press.

[192] H.-J. Siegert and U. Baumgarten. *Betriebssysteme: Eine Einführung*. R. Oldenbourg Verlag, München, 1998.

[193] A. Sinsel. Linuxportierung auf einen eingebetteten powerpc 405 zur steuerung eines neuronalen netzwerkes. Diploma thesis (german), University of Heidelberg, HD-KIP-03-14, 2001.

[194] R. E. Smith, C. Bonacina, P. Kearney, and W. Merlat. Embodiment of evolutionary computation in general agents. *Evolutionary Computation*, 8(4):475–493, 2001.

[195] R. E. Smith and J. E. Smith. New methods for tunable, random landscapes. In W. N. Martin and W. Spears, editors, *Foundations of Genetic Algorithms*, volume 6, pages 47–67, 2001.

[196] W. M. Spears. Simple subpopulation schemes. In A. V. Sebald and L. J. Fogel, editors, *Proceedings of the Third Annual Conference on Evolutionary Programming*, pages 296–307. World Scientific, 1994.

[197] W. M. Spears and K. A. De Jong. On the virtues of parametrized uniform crossover. In R. K. Belew and L. K. booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 230–236, San Diego, CA, 1991. Morgan Kaufmann.

[198] W. M. Spears and K. DeJong. Dining withGAs: Operator lunch theorems. In W. Banzhaf and C. Reeves, editors, *Foundations of Genetic Algorithms*, volume 5, pages 85–101, 1999.

[199] K. O. Stanley and R. Miikkulainen. Continual coevolution through complexification. In W. Langdon et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2002*, pages 113–120. Morgan Kaufmann Publishers, July 2002.

[200] K. O. Stanley and R. Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In W. Langdon et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2002*, pages 569–577. Morgan Kaufmann Publishers, July 2002.

[201] V. Storch, U. Welsch, and M. Wink. *Evolutionsbiologie*. Springer Verlag, Berlin, Heidelberg, New York, 2001.

[202] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, MA, August 1997.

[203] G. Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9, San Francisco, 1989. Morgan Kaufmann.

[204] R. Tanese. Parallel genetic algorithm for a hypercube. In J. J. Grefenstette, editor, *Proceedings of the 2nd International Conference on Genetic Algorithms*, pages 177–183, Hillsdale, New Jersey, 1987. Lawrence Erlbaum.

[205] J. Teichert and R. Malaka. An association architecture for the detection of objects with changing topologies. In *Proceedings of the International Joint Conference on Neural Networks IJCNN'03*, pages 125–130. IEEE Press, July 2003.

[206] D. Thierens, J. Suykens, J. Vandewalle, and B. DeMoor. Genetic weight optimization of a feedforward neural network controller. In *Proceedings of the Conference on Artificial Neural Nets and Genetic Algorithms*, pages 658–663, Berlin, Heidelberg, New York, 1993. Springer Verlag.

[207] R. F. Thompson. *The Brain*. W. H. Freeman and Company, New York and Oxford, 1985.

[208] S. J. Thorpe, D. Fize, and C. Marlot. Speed of processing in the human visual system. *Nature*, 381:520–522, 1996.

[209] M. Trefzer, J. Langeheine, K. Meier, and J. Schemmel. New genetic operators to facilitate understanding of evolved transistor circuits. In *Proceedings of the 2004 NASA/DoD Conference an Evolvable Hardware (EH2004)*, 2004.

[210] Trolltech AS. The Qt application development framework. Waldemar Thranes gate, 98, NO-0175 Oslo, Norway, `http://www.trolltech.com/products/qt/`.

[211] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.

[212] M. Valle. Analog VLSI implementations of neural networks with supervised on-chip-learning. *Analog Integrated Circuits and Signal Processing*, 33:263–287, 2002.

[213] D. van Heesch. The doxygen documentation system. 2004, `http://www.stack.nl/~dimitri/doxygen`.

[214] E. van Nimwegen, J. P. Crutchfield, and M. Mitchell. Statistical dynamics of the Royal Road genetic algorithm. *Theoretical Computer Science*, 229:41–102, 1999.

[215] E. A. Vittoz. Analog VLSI signal processing: Why, where and how? *Analog Integrated Circuits and Signal Processing*, 8(1):27–44, 1994.

[216] E. A. Vittoz. Analog VLSI implementation of neural networks. In E. Fiesler and R. Beale, editors, *The Handbook of Neural Computation*, New York, January 1997. Institute of Physics Publishing and Oxford University Publishing.

[217] M. D. Vose and G. E. Liepins. Punctuated equilibria in genetic search. *Complex Systems*, 5(1):31–44, 1992.

[218] R. L. Watrous. Learning algorithms for connectionist networks: Applied gradient methods for nonlinear optimization. In M. Caudill and C. Butler, editors, *Proceedings of the IEEE First International Conference on Neural Networks*, volume II, pages 619–627, San Diego, 1987. IEEE.

[219] H. White. Connectionist nonparametric regression: Multilayer feedforward networks can learn arbitrary mappings. *Neural Networks*, 3(5):535–549, 1990.

[220] P. M. White and C. C. Pettey. Double selection vs. single selection in diffusion model GAs. In T. Bäck, editor, *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 174–180, San Fransiso, 1997. Morgan Kaufmann.

[221] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. Morgan Kaufmann, 1989.

[222] D. Whitley. Cellular genetic algorithms. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*, page 658, San Francisco, 1993. Morgan Kaufmann.

[223] D. Whitley and T. Hanson. Optimizing neural networks using faster, more accurate genetic search. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 391–395, 1989.

[224] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, 14(3):347–361, 1990.

[225] L. D. Whitley. Fundamental principles of deception in genetic search. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, volume 1, pages 221–241, 1991.

[226] B. Widrow and M. A. Lehr. 30 years of adaptive neural networks: Perceptron, Madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.

[227] B. Widrow and M. E. Hoff. Adaptive switching circuits. In *IRE WESCON Convention Record*, pages 96–104, New York, 1960. IRE.

[228] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[229] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124-3400, USA. *Virtex E Datasheet*.

[230] Xilinx, Inc., www.xilinx.com. *Virtex-II Pro Platform FPGA Handbook*, 2002.

[231] Xilinx, Inc., www.xilinx.com. *PowerPC processor reference guide*, September 2003.

[232] Xilinx, Inc., www.xilinx.com. *PowerPC 405 processor block reference guide*, August 2004.

[233] Xilinx Inc., San Jose. *Xilinx XC9536XL High Performance CPLD*, September 2004.

[234] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[235] X. Yao and Y. Liu. Ensemble structure of evolutionary artificial neural networks. In *Proceedings of the Third IEEE International Conference on Evolutionary Computation (ICEC'96)*, pages 659–664, Nagoya, Japan, May 1996.

[236] X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, 8(3):694–713, May 1997.

[237] X. Yao and Y. Liu. Making use of population information in evolutionary neural networks. *IEEE Transactions on Systems, Man and Cybernetics, Part B: Cybernetics*, 28(3):417–425, 1998.

Danksagung (Acknowledgements)

We are not an endangered species ourselves yet,
but this is not for lack of trying.

Douglas Adams, Last Chance to See

Abschließend möchte ich all jenen, die zum Gelingen dieser Arbeit beigetragen haben, ein herzliches Dankeschön aussprechen. Mein Dank gilt vor allem

- Herrn Prof. Dr. K. Meier für die freundliche Aufnahme in seine Arbeitsgruppe und die Möglichkeit, an einem so interessanten Projekt mitarbeiten zu können.
- Herrn Prof. Dr. F. A. Hamprecht, der freundlicherweise das Zweitgutachten übernommen hat.
- Dr. Johannes Schemmel, der nicht nur mit der Entwicklung des HAGEN Chips einen wesentlichen Grundstein für diese Arbeit gelegt hat, auf dessen umfassendes fachliches Wissen und visionären Ideenreichtum ich auch in unzähligen Diskussionen zurückgreifen durfte, der immer auf alle Probleme eine Antwort wusste und dessen freundliche Art mir die Arbeit stets angenehm machte.
- Felix Schürmann für die jahrelange kollegiale Zusammenarbeit, viele fruchtbare Diskussionen, bereitwillige Hilfe in allen Hardware-Fragen, wertvolles inspirierendes Material für die Kapitel 5 und 6 und die immer geduldige Unterstützung bei unzähligen Linux-Problemen.
- Johannes Fieres für weitaus mehr als ich hier aufzählen könnte, vor allem jedoch für enorme Mengen genialen C++ Codes, objekt-orientiertes Denken in allen Lebenslagen, viele erhellende fachliche und (mindestens ebenso viele) nichtfachliche Diskussionen, des öfteren Schokolade, Abendessen im Botanik, gute Stimmung im Büro, diverse Schülertage, bei denen ich nicht zu erscheinen brauchte, die aufmerksame Korrektur des Manuskripts und vor allem seine trotz alledem andauernde Freundschaft.
- Eilif Mueller, nicht nur weil sein ehemaliges `VisWidget` immer noch toll aussieht, sondern auch für seinen wesentlichen Beitrag zur netten Atmosphäre im Büro, viele inspirierende Gedanken zu theoretischen Fragen und dafür, dass er mich so oft an seinen Englischkenntnissen hat teilhaben lassen.
- Tillmann Schmitz, der es nicht nur geschafft hat, einen schnellen evolutionären Koprozessor zu zaubern und stetig zu verbessern, sondern der ihn auch bereitwillig selbst den abstrusesten Vorstellungen der Jungs aus dem Softwarezimmer anzupassen bereit war.

- Allen Mitgliedern der Electronic Vision(s) Gruppe für die vielen lustigen Arbeitsgruppen-Meetings, die immerwährende Hilfsbereitschaft — egal, wen man fragt — ein harmonisches Arbeitsklima, das nahezu vollständige Erscheinen auf allen WG- und Geburtstagsparties und dass man auch, wenn man nicht um 12 Uhr essen geht und nur ein klapperiges altes Schrottfahrrad fährt, das Gefühl hat dazuzugehören.
- meinem ersten und leider einzigen Diplomanden Dominik „Bon“ Niedenzu für seine vielfältige konstruktive Kritik am HANNEE Programm, seine erfrischend andere Sichtweise der Dinge und dafür, dass er es geschafft hat, Diplomand zu sein und gleichzeitig Kumpel zu bleiben.
- Kristoffer Lerch und Ulf Bissbort, deren vorbildliches Engagement sie im Rahmen ihrer Projektarbeiten wertvolle Beiträge zur HANNEE Software bzw. der Forschung an den schrittweisen Trainingsmethoden hat liefern lassen.

Abgesehen von der fachlichen und anderweitig arbeitsbezogenen Unterstützung, die mir im Laufe dieses Projekts zuteil wurde, haben insbesondere auch der freundschaftliche und emotionale Beistand mehrerer lieber Menschen die letzten Jahre für mich zu einer wertvollen Erfahrung werden lassen, die ich nicht mehr missen möchte. An dieser Stelle will ich auch hierfür ein herzliches Dankeschön loswerden an:

- alle Mitglieder (die ehemaligen wie die aktuellen) der Kultband „Fake It“: Ani, Hanni, Kranky, Mad und Merd dafür, dass sie ihren despotischen „Frontman“ auch nach Jahren immer noch ertragen, für einmal in der Woche rumschreien dürfen, tolle Musik, bombastische Auftritte, jede Menge Spass, kein Rauchen im Proberaum und überhaupt das ganze Rockband-Lebensgefühl.
- Martin „Desboth“ Both, Gunnar „the Gunner“ Schramm, Thomas „the Killer“ Künsting und Blizzard Entertainment für wiederholtes wildes gezocke, Prosecco und viel zu viel Pizza. Ab demnächst bin ich wieder voll dabei: „That's it. I'm dead.“
- den Mensaclub: Annika, Franzi, Fred, Gunnar, Johannes, Kristin, Kristine, Martin, Nicole, Stefan, Thomas und alle gelegentlichen Besucher für das unangefochtene soziale Highlight meines Arbeitstages, interessante und nicht immer so bierernst zu nehmende Diskussionen über Gott und die Welt und das Tolle Gefühl, Mitglied in einem der erlesensten und elitärsten Intellektuellenclubs der Welt zu sein.
- meine WG — Angelika, Beate und Michael — sowie meine Lieblings-Ex-Mitbewohner Nicole und Sven für ein trautes und harmonisches Heim, auch wenn ich nicht oft genug dort war um immer mein Zeug abzuspülen.
- meine allerliebste Verena für ihre nicht enden wollende Geduld mit ihrem ständig arbeitenden Freund, ihre aufopferungsvolle Unterstützung in den letzten Wochen und vor allem dafür, dass mit ihr das Leben so viel schöner ist.
- meine lieben Eltern, die inhaltlich vielleicht nicht so viel zu dieser Doktorarbeit beigetragen haben, die es mir aber durch ihre stete liebevolle Unterstützung in jeglicher Hinsicht überhaupt erst ermöglicht haben, es bis hierhin zu schaffen.