RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Johannes Fieres

A Method for Image Classification Using Low-Precision Analog Elements

Doctral Thesis

HD-KIP 06-20

KIRCHHOFF-INSTITUT FÜR PHYSIK

Dissertation
submitted to the
Combined Faculties for the Natural Sciences and for
Mathematics
of the Ruperto-Carola Univertsity of Heidelberg, Germany
for the degree of
Doctor of Natural Sciences

presented by
Dipl. phys. Johannes Fieres
born in Fulda, Germany

Oral examination: November, 29, 2006

A Method for Image Classification Using Low-Precision Analog Computing Arrays

Referees: Prof. Dr. Karlheinz Meier
Prof. Dr. Bernd Jähne

Eine Methode zur Bildklassifikation mit analogen Recheneinheiten beschränkter Genauigkeit

Zusammenfassung

Das Rechnen mit analogen integrierten Schaltkreisen kann gegenüber der weit verbreiteten Digitaltechnik einige Vorteile bieten, z.B.: geringerer Fläche- und Stromverbrauch und die Möglichkeit der massiven Parallelisierung. Dabei muss allerdings aufgrund unvermeidlicher Produktionsschwankungen und analogen Rauschens auf die Präzision digitaler Rechner verzichtet werden. Künstliche neuronale Netzwerke sind hinsichtlich einer Realisierung in paralleler, analoger Elektronik gut geeignet. Erstens zeigen sie immanente Parallelität und zweitens können sie sich durch Training an eventuelle Hardwarefehler anpassen. Diese Dissertation untersucht die Implementierbarkeit eines neuronalen Faltungsnetzwerkes zur Bilderkennung auf einem massiv parallelen Niedrigleistungs-Hardwaresystem. Das betrachtete, gemischt analog-digitale, Hardwaremodell realisiert einfache Schwellwertneuronen. Geeignete gradientenfreie Trainingsalgorithmen, die Elemente der Selbstorganisation und des überwachten Lernens verbinden, werden entwickelt und an zwei Testproblemen (handschrifltiche Ziffern (MNIST) und Verkehrszeichen) erprobt. In Softwaresimulationen wird das Verhalten der Methode unter verschiedenen Arten von Rechenfehlern untersucht. Durch die Einbeziehung der Hardware in die Trainingsschleife können selbst schwere Rechenfehler, ohne dass diese quantifiziert werden müssen, implizit ausgeglichen werden. Nicht zuletzt werden die entwickelten Netzwerke und Trainingstechniken auf einem existierenden Prototyp-Chip überprüft.

A Method for Image Classification Using Low-Precision Analog Computing Arrays

Abstract

Computing with analog micro electronics can offer several advantages over standard digital technology, most notably: Low space and power consumption and massive parallelization. On the other hand, analog computation lacks the exactness of digital calculations due to inevitable device variations introduced during the chip production, but also due to electric noise in the analog signals. Artificial neural networks are well suited for parallel analog implementations, first, because of their inherent parallelity and second, because they can adapt to device imperfections by training. This thesis evaluates the feasibility of implementing a convolutional neural network for image classification on a massively parallel low-power hardware system. A particular, mixed analog-digital, hardware model is considered, featuring simple threshold neurons. Appropriate, gradient-free, training algorithms, combining self-organization and supervised learning are developed and tested with two benchmark problems (MNIST hand-written digits and traffic signs). Software simulations evaluate the methods under various defined computation faults. A model-free closed-loop technique is shown to compensate for rather serious computation errors without the need for explicit error quantification. Last but not least, the developed networks and the training techniques are verified on a real prototype chip.

Contents

Introduction

Most of today's data processing tasks are efficiently and economically solved by digital computing machines based on the principles described by John von Neumann in 1945. In fact, the von Neumann architecture has proven to be so universal that, today, digital computers are found in virtually every sector of private and professional life.

The enormous success of digital technology has pushed the pursue of alternative computing approaches away from the public view, although some of them have the potential to offer advantages in various terms. Examples are computing with analog electric signals, or new emerging technologies like DNA computing or quantum computing. Two of the following limitations of digital computers were foreseen already by von Neumann himself[1]:

Fault-Intolerance *"The desired automatic functioning must, of course, assume that it [i.e., the computer] functions faultlessly. ... Any error may vitiate the entire output of the device"*. Everybody knows this truth from their recent hard drive crash. But also, for example, during the chip production, manufacturing faults constitute a serious issue: With constantly growing chip areas and circuit densities the probability of defects increases, boosting production costs.

Explicit Programming *"The instructions ... must be given to the device in absolutely exhaustive detail. ... [requiring] some code to express the logical and algebraic definition of the problem under consideration"*. A problem the solution of which cannot be put down in exact instructions cannot be solved by a computer. Modern so-called *machine learning* techniques can partly account for this problem. However, for many of these methods, von Neumann computers are actually not the optimal substrate. E.g., the parallel execution in artificial neural networks or the multi-valued logic of fuzzy systems can only be *simulated* with digital processors.

Sequential Computing Algorithms implementable on von Neumann machines are inherently sequential processes. Attempts of simulating truly parallel systems (e.g., hydro-dynamic processes or realistic brain models) on such machines require massive equipment (super computers, cluster computers) and highly sophisticated programming.

Power Consumption The trend to ever higher device densities and clock fequencies goes along with a considerable amount of power dissipation.

[1] Quotes taken from his 1945 publications [42]

> Besides paying the high electricity bill, cooling down the systems requires more and more effort. Today, for the private user this problem becomes apparent mainly by a higher number of air fans in the PC, but for the maintainer of a super computing center, cooling can in fact constitute a major item of expenses.

Computing with analog electric circuits—in contrast to digital—constitutes one of the alternative techniques mentioned at the beginning. Digital computers operate on abstract symbols represented within a complex electronic machinery. In analog computing circuits, basic laws of physics are exploited for performing calculations. Examples are the Kirchhoff rules for currents or the electrical properties of solid matter boundaries. This difference makes analog micro electronics usually occupy way less silicon area and consume factors less power compared to digital devices performing an equivalent task.

Space and power efficiency enable the design of massively parallel systems where a large arrays of equivalent computing elements are integrated on one micro chip. Such devices are quite different from conventional computers and using them requires to leave common patterns of sequential algorithmics behind for the search of new, truly parallel computing techniques.

When performing computations with analog elements, one of the major properties of digital technology is to be sacrificed: preciseness. Quantities coded by analog signals are inherently limited in precision. An upper limit of accuracy is constituted by noise present in the system, e.g., by thermal fluctuations or undesired electro-dynamical side effects. Another source of uncertainty are random device variations inevitably occurring during the manufacturing process. Although some of the variations can be compensated for by sophisticated circuitry and calibration procedures, the software running on analog devices must be tolerant against imperfect calculations.

Artificial neural networks are a promising application for a massively parallel analog implementation: First, they consist of a large number of identical, parallelly working, computing units which makes the employment of a parallel device a natural approach. Second, artificial neural networks are adaptively trainable for a given task, and can thus be supposed to be able to cope with substrate imperfections and precision limitations. Although the computing paradigm is very different from conventional algorithms, the concept has already proven to be successfully applicable to many problems.

So-called *convolutional* neural networks are a special type of artificial neural networks which are applied for recognizing objects in images. Inspired by biological research on the mammalian visual cortex, such networks implement a hierarchical set of feature extraction stages, allowing to learn object representations invariant of many modes of appearance, e.g., position, scale, deformation, or illumination. Convolutional networks require very large network sizes and are thus considered to benefit especially from a massively parallel implementation.

The by far most popular training method for neural networks, backpropagation, works excellent for networks of small and medium sizes. For very large network scales it becomes more and more impractical due to its high computational complexity. It is necessary to development training methods which scale well together with the network size. Self-organization and local learning are promising approaches.

This thesis presents a convolutional neural network application for object recognition and evaluates its feasibility of being implemented in analog hardware. A prototype realization of a massively parallel analog neural network architecture was available for experiments. Topics covered in this thesis include the development of a training method adequate for the network model featured by this particular hardware system, computer simulations of the network behavior under various computation faults, and the investigation of the network when running on the prototype hardware system.

Organization

The thesis is organized in the following chapters.

1 Background The fields relevant for this thesis are shortly reviewed. Preceded by a section about biological foundations of neural networks, the concept of object recognition with convolutional networks is introduced. Previously reported training methods are covered. The chapter is closed by a review of analog hardware implementations of neural networks. p. 7

2 Working Environment The hardware and software setup used to conduct the experiments are described. Special emphasis is given to the software tools developed as part of this thesis. p. 31

3 A Neural Network for Object Recognition This chapter presents the main contribution of this thesis. A convolutional neural network, suited for implementation on the present hardware system is described. Adequate training methods, based on both local self-organization and supervised learning, are developed. p. 53

4 Results With Ideal Neurons The methods described in chapter 3 are tested in a computer simulation on two problems: The recognition of handwritten digits and the recognition of traffic signs on photographs. The properties of the system are investigated and suggestions for further performance improvement are presented. p. 63

5 Robustness Against Computation Faults An application implemented in analog hardware should be robust against computation inaccuracies. The robustness of the developed convolutional network is evaluated. Two chip-in-the-loop training strategies are suggested which take possible errors into account already during training. Simulation results with and without applying the chip-in-the-loop techniques are presented. p. 81

6 Hardware Implementation The methods developed in chapter 3 and 5 are tested on the neuro-chip prototype HAGEN and compared with software simulations. Particular implementation details required by the used hardware are described. The size limitation of the prototype chip imposes restrictions on the implementable networks. Pruning strategies are applied for fitting the networks onto the chip. p. 91

Chapter 1

Background

1.1 Biological Inspiration

1.1.1 The Nervous System

Biological organisms use their perception of the environment to generate behavior, which in turn acts upon the environment. This process requires some sort of information processing taking place within the organism's body. In animals and humans, this task is for a large part accomplished by a special type of cells, the *nervous cells* or *neurons*. These cells are known to be able to generate electric signals in response to external stimuli. Usually, the cell body possesses long appendices, the dendrites and the axon, which are able to propagate the signals over a certain distance (up to in the order of meters) from and to other cells (see Figure 1.1). This way, large and heavily connected networks of neurons are formed. In the human brain, each of the approcimately 5 Billion cells connects on average to about 10,000 other cells. These neural networks are are believed to provide the basis for purposeful behavior and intelligence.

Let us characterize the nature of the neural electrical signals in more detail: A nervous cell in living tissue resides floating in an electrolytic solution. The

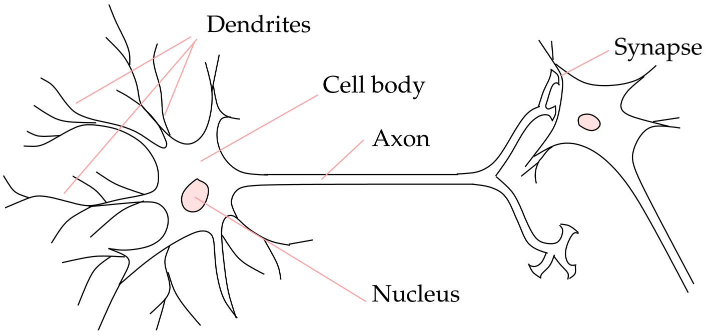


Figure 1.1: *A nervous cell*

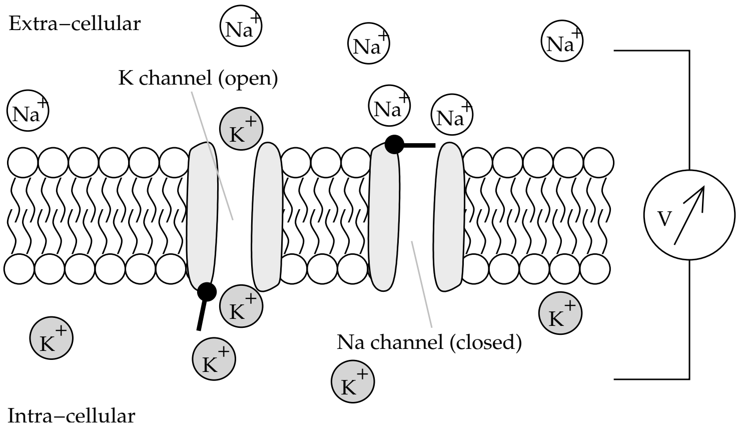


Figure 1.2: *A cell's membrane consists of a double lipid layer. Diffusion of ions through membrane channels results in an electric potential between the inner and the outer space of the cell.*

inner and outer part of the cell are separated by the cell's membrane, a double lipide layer, see Fig. 1.2. Mechanisms which are not covered here maintain a concentration difference of various charged particles between the inside and the outside. Only the most important ions, K^+ (Potassium) and Na^+ (Sodium), are considered in the figure. Ion channels, complex molecules penetrating the membrane, permit or deny certain types of ions to pass and follow their concentration gradient. By this controlled diffusion process, charge is carried across the membrane, resulting in an electric potential which in turn leads to an ion flow in the opposite direction, along the electric field. These two processes settle at a dynamic equilibrium state corresponding to a membrane potential of typically about $U_{\mathrm{in}} - U_{\mathrm{out}} = -70\mathrm{mV}$.

Most ion channels can change between an open and closed state, influenced by external parameters (e.g., the current membrane voltage), or by internal mechanisms (e.g., implicit time dependence). This is the basis for an important feature of nervous cells: The actual membrane potential can vary within a relatively wide range, between -90mV and +40mV, at every point in time defined by the current permeability of the various channel types.

The particular dynamics of the different sorts of ion channels provide the mechanisms for the phenomenon of *action potentials*: As soon as the membrane potential exceeds a given threshold (usually around -55mV) a transient voltage break-out is triggered, as shown in Fig. 1.3. The temporal evolution of the voltage spike is stereotypical: It is exactly the same for every action potential. An action potential causes the voltage in adjacent regions of the membrane to grow above the threshold, too. This way, once triggered, action potentials propagate along the cell's membrane, thus carrying information to remote areas. Because all action potentials are identical, no information is carried in the form of an action potential, but only in the instance of time when it was triggered.

As a mechanism of passing information between cells in neural networks, action potentials are able to influence the membrane potentials of other neurons. This happens primarily at structures called *synapses* (see Fig. 1.1). There are two main sorts of synapses: On arrival of an action potential, excitatory synapses de-polarize (i.e., increase the voltage of) the membrane of the target (or *post-synaptic*) cell, while inhibitory synapses polarize the same. Simply speaking, the post-synaptic cell integrates over all incoming voltage changes received over a period of time, and, if the firing threshold is exceeded, generates an action potential itself. This behavior is described by the *Integrate-and-Fire* neuron model [].

1.1.2 Rate-Based Neuron Model

In order to describe communication in larger networks of neurons, a simple rate-based model is often used which has proven to be adequate for describing many basic phenomena. In this model, the train of spikes produced by a given neuron is represented by a time-dependent continuous value being equal to the current firing rate (e.g., measured in spikes per second). The firing rate is computed by counting the number of spikes occurring within a moving time window. The size of this window is chosen large compared to the mean time between spikes, but small compared to the time scale of the investigated phenomena. Let a spike train be approximated by a sequence of delta peaks $S(t) = \sum_i \delta(t - t_i)$, where t_i is the time of the ith spike. The firing rate r is then defined by:

$$r(t) = \int_{t'=-\infty}^{\infty} k(t' - t)S(t')\mathrm{d}t', \tag{1.1}$$

where $k(t)$ is a bounded kernel function representing the moving window. In the most simple case, k is chosen as a function being equal to 1 in a given interval $[-a, a]$, and 0 otherwise.

A cell usually receives input spike trains from a large number (typically in the order of 10^4) of other cells. If the input rates stay constant over time, the output rate O will saturate at a steady state:

$$O = F\left(\sum w_i I_i\right) \qquad \text{(steady state)}, \tag{1.2}$$

where w_i is the efficacy of the ith input synapse and I_i is the ith input spike rate arriving at the ith synapse. The efficacy w_i of a synapse is also called its *strength* or its *weight*. The activation function F is not explicitely defined here. It allows

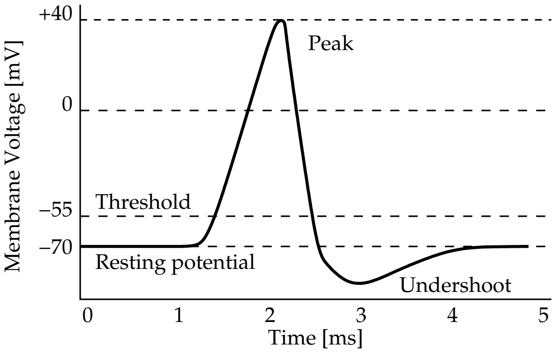


Figure 1.3: *Action potentials are stereotypical fluctuations of the membrane voltage, constituting the basic signals for inter-neuron communication.*

to model non-linear dependence on the input. For a thorough derivation of equation (1.2), read for example chapter 7 of the textbook [6].

1.1.3 Activity-Driven Learning Mechanisms

The nervous system of an organism is under constant development throughout its entire lifespan. Activity-dependent synaptic plasticity is believed to constitute the basic mechanism for learning [6, 12]; synaptic plasticity means that the weight of a given synapse is subject to modification. Many widely accepted models of plasticity are based on a principle formulated by Donald Hebb in 1949: He stated that if a neuron repeatedly contributes to the firing of another neuron, then the synapses between both neurons are strengthened.

The processes of synaptic plasticity are usually much slower than the duration of one spike. If we further assume that the pre-synaptic firing rates change slowly enough, we can well use the steady-state formula (1.2). In addition, we will replace the activation function F by the identity function. This corresponds to looking at a linearized version of (1.2), which is a reasonable simplification as long as the considered changes in w_i are small. Using vector notation where $\mathbf{w} = [w_1, w_2, \ldots]^T$ and $\mathbf{I} = [I_1, I_2, \ldots]^T$, the output firing rate reads:

$$O = \mathbf{w}^T(t)\mathbf{I} \qquad \text{(linearized).} \tag{1.3}$$

The weight vector is written as a function of time in order to emphasize our interest in synaptic changes.

The learning rule of Hebb, stated above, might be now expressed as

$$\epsilon \frac{\mathrm{d}\mathbf{w}(t)}{\mathrm{d}t} = \mathbf{I}O, \tag{1.4}$$

where the constant ϵ determines the speed of learning. This formula implies that simultaneous activity of the pre-synaptic and the post-synaptic cell causes an increase of the connecting synaptic weight. Usually, a cell receives many different input stimuli (patterns) during the learning process. So, the effective weight change may be written as the average change over many patters. Let $\langle \cdot \rangle$ denote the average over many patterns $\mathbf{I}$:

$$\epsilon \Delta \mathbf{w} = \langle \mathbf{I}O \rangle, \tag{1.5}$$

and replacing $O = \mathbf{w}^T\mathbf{I}$,

$$\epsilon \Delta \mathbf{w} = \langle \mathbf{I}\mathbf{w}^T\mathbf{I} \rangle = \langle \mathbf{I}\mathbf{I}^T\mathbf{w} \rangle = \langle \mathbf{I}\mathbf{I}^T \rangle \mathbf{w}. \tag{1.6}$$

From the last line in (1.6), one important property of Hebbian-style learning can be derived. The matrix $\langle \mathbf{I}\mathbf{I}^T \rangle$ represents nothing less than the statistical correlations among the inputs. So, where does the weight vector $\mathbf{w}$ saturate? One theoretical solution is to require $\Delta \mathbf{w} = 0$, which means that $\langle \mathbf{I}\mathbf{I}^T \rangle$ has zero rank and $\mathbf{w}$ is pointing into a direction where the inputs have no correlation. However, this solution can be shown to be unstable against small weight fluctuations [34], and in practice, with large and diverse input pattern sets, $\langle \mathbf{I}\mathbf{I}^T \rangle$ has usually full rank. The remaining solution are diverging weights, where the modulus of $\mathbf{w}$ keeps growing, and the direction of $\mathbf{w}$ will approach the direction of the largest eigenvector of $\langle \mathbf{I}\mathbf{I}^T \rangle$. To see this, suppose the initial weight

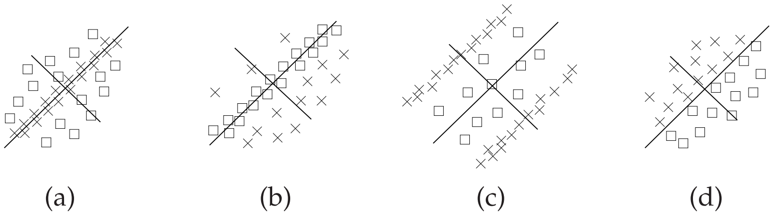


Figure 1.4: *Receptive field geometries, as found by Hubel and Wiesel 1962, of four typical "simple cells". × - areas giving excitatory responses; □ - areas giving inhibitory responses.* *(After [18])*

vector be de-composed into the Eigen-components of $\langle \mathbf{I}\mathbf{I}^T \rangle$. Then, by means of (1.6), the component with the largest eigenvalue will grow faster than all the other components, and, in the limit $t \to \infty$, dominate. This way, a simple rate-based neuron, together with Hebbian-style learning, is capable of performing a principal component analysis of the input data (see also [44]).

The above discussion only accounts for excitatory synapses and synaptic potentiation (weight increase). Various models following the spirit of equation 1.4 have been proposed to include inhibitory synapses and synaptic depression (weight decrease in response to low input/output correlation). The interested reader is referred to textbooks [6, 12]. In this thesis, inhibition and depression are incorporated into (1.4) by allowing all quantities (inputs, outputs, weights) to assume negative values, thereby abandoning a strictly biological model.

1.1.4 Visual Processing in the Brain

The visual systems of human beings and higher animals exhibit a remarkable ability to recognize seen objects. Recognition is accomplished robustly and with high speed, tolerant of many variances in, for example, positional shift, view angle, or illumination conditions, and unaffected by deviations from a learned prototype object or partial occlusion. What mechanisms are these outstanding capabilities based on? In fact, the visual cortex of mammals belongs to the best-known regions of the brain today. In addition to investigating small pieces of dissected living neural tissue, or studying whole brains post-mortem, modern techniques allow to observe neural activity in behaving organisms. Starting with the Nobel-Prize winning work of Hubel and Wiesel with living cats in 1962 [18], elaborate models of early visual processing have been established.

Hubel and Wiesel recorded the activities of cells in a cat's primary visual cortex (anatomical area V1) while presenting stimuli to the respective cell's receptive field. The receptive field of a neuron is the area on the retina which, by stimulation, can influence the activity of this neuron. Hubel and Wiesel classified the cells they observed into several groups. One class, which they called "simple cells", has receptive fields arranged into well-defined excitatory and inhibitory regions (cf., Figure 1.4). Obviously, the stimuli which make these neurons fire strongly are dark/bright straight lines or edges before a bright/dark background. A slight change in the optimal position or orien-

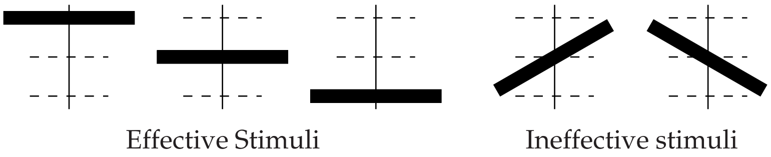


Figure 1.5: *Behavior of a typical "complex" cell: A dark bar of a certain orientation (here: horizontal) evokes activity independently if the exact position. Tilting the stimulus away from its optimal orientation renders it ineffective.* *(After [18])*

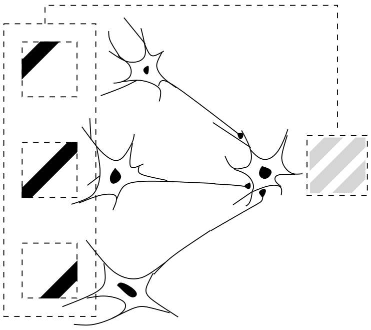


Figure 1.6: *Simple neural model explaining the emergence of invariant recognition: Many cells with simple receptive fields make excitatory connections to a more complex cell. In this example, the cell to the right will respond to an oblique bar, invariant to local shift.*

tation of the stimuli results in a significant decrease in firing activity. Another class of cells responds to specific line orientations, but independent of the exact position (Figure 1.5). These kind of neurons were termed "complex cells". Later, more cell types with increasing complexity were discovered in higher visual areas by the same authors and by other scientists (e.g., [19, 62]). Examples range from neurons responding to line combinations (corners, angles) to neurons responding selectively to shapes as complex as faces. One observation is that neurons responding to more complex shapes usually show a higher degree of invariance to shift, scale, orientation, or illumination of the presented shape.

One possible explanation for how this hierarchical set of feature detectors emerges in the brain is that complex cells receive excitatoty input connections from a range of simpler cells detecting similar features, as shown in Figure 1.6. Although this simple model is not able to explain all mechanisms of visual perception, it is widely accepted as playing an important role for early visual processing [48].

1.1.5 Biological Implications for Artificial Systems

When designing artificial neural vision systems, nature can provide several guidelines about the approach to choose. Usage of hierarchical feature detectors has been discussed in the previous section as one example. Another hint regards the required amount of computation resources [45]: From measure-

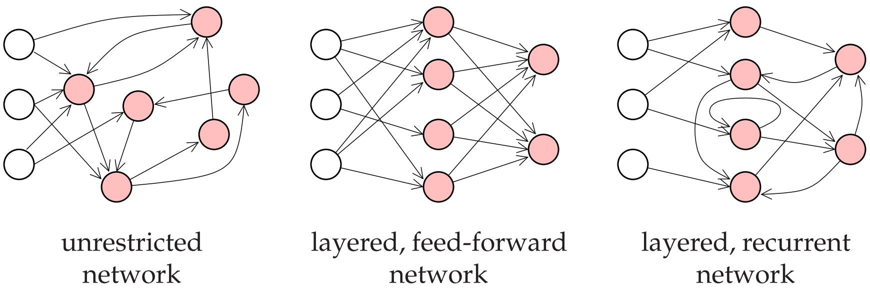


Figure 1.7: *Artificial neural networks are directed graphs of processing units. Input nodes are drawn in white. Depending on the connection topology, different classes of networks can be distinguished.*

ments, the delay between stimulus onset and recognition during the perception process in the brain is known. Dividing this time span by the time necessary for one neuron to transmit a spike to another restricts the number of possible neural relays having occurred during the recognition process to not more than eight. Anatomical studies of the visual areas and their inter-connections show that this is indeed the approximate number of neurons involved in a path from the primary visual cortex to the recognition stage. As a conclusion, basic object recognition cannot rely on lateral or feed-back connections, since this would not be consistent with the measured latency. For an artificial system, a purely feed-forward solution with approximately eight processing stages should be sufficient.

Interestingly, the mentioned time limit does not allow for neurons in the brain to saturate at a defined firing rate. In order to match the measurements, information must be passed on the basis of the first spike. During the first moments of recognition, the neural information seems to be entirely determined by whether a neuron is active or silent, "on" or "off". This implies that in fact a simple threshold neuron model, as employed in this thesis, can be expected to be sufficient to mimic at least some of the recognition capabilities present in the brain.

1.2 Convolutional Neural Networks

In order to read this thesis, basic knowledge about the general concepts of artificial neural networks is assumed. A few facts are shortly reviewed: . Basically, neural networks are directed graphs where the nodes (neurons) are processing units and the edges transport information between them. In the networks used in this thesis, the information is numerical data. A neuron computes a weighted sum of the afferent signals, and provides a scalar function of this sum as its output (cf., equation (1.2)). The scalar function is usually chosen as a bounded sigmoid or a step function, in which case one speaks of threshold neurons. The behavior of such a network is defined largely by the weights of the individual connections.

Networks can be divided into several classes, depending on the connection topology (examples in Figure 1.7). Different update schemes of the neuron outputs are possible, i.e., synchronous or asynchonous update, time-continuous or clocked operation. Further detail are found in a number of excellent text books on this topic [17, 3, 49].

1.2.1 Overview

We have learned in section 1.1.4 that hierarchical feature detectors are believed to play an important role in the brain's visual system. In the following, it will be discussed how the principle of hierarchical structures are taken advantage of in artificial vision systems. In particular, the focus will be on so-called convolutional neural networks, a special type of artificial neural networks with a specific connection topology. Such networks have been successfully employed for industrial image analysis applications (character recognition [32], face identification [29, 67]), but are also used as models in computational neuroscience [34, 38, 48].

Convolutional neural networks belong to the class of layered, feed-forward networks.[1] The first layer usually detects simple features, e.g., oriented line segments. By successive feature extraction through the layer hierarchy, more and more complex shapes, and finally entire objects can be recognized in higher layers. In contrast to many standard feed-forward networks, adjacent layers are not fully connected. Rather, a particular local connectivity scheme implements topology-preserving feature maps which are the basis for the hierarchical image analysis.

Among the many authors who have contributed to the promotion of convolutional networks for image processing applications, the two probably most original are Kunihiko Fukushima and Yann LeCun. Fukushima invented a convolutional neural network for shift-invariant object recognition in 1980, which he named the "Neocognitron". It could be trained by both self-organization and supervision. The supervised training yielded better results but involved time-consuming manual training (cf., section 1.3.4). In 1989, Yann LeCun introduced a system for hand-written digit recognition based on a convolutional neural network trained by back-propagation which was utilized commercially for post office zip-code recognition. Parts of the computations were accelerated by a custom-made hardware device (see also section 1.4).

Although LeCun's work appeared years after the "Neocognitron", LeCun never cited Fukushima (neither did Fukushima cite LeCun in his later work). Therefore it is likely that the two authors, even though their methods have strong overlap, worked completely independent.

1.2.2 Invariant Recognition: From Local to Global Invariance

An ideal vision system is able to identify and classify objects invariant of positional shift, view angle, or illumination conditions, and unaffected by deviations from a learned prototype object. However, experience shows that such functionality is not easily programmed into artificial systems. The reason is that, in pixel space, different views of the same object are usually not close to each other (cf., Figure 1.11). Defining the common characteristics of the various

[1] Although feed-back or lateral connections were sporadically proposed in more theoretical work, most real-life applications do without recurrent connections.

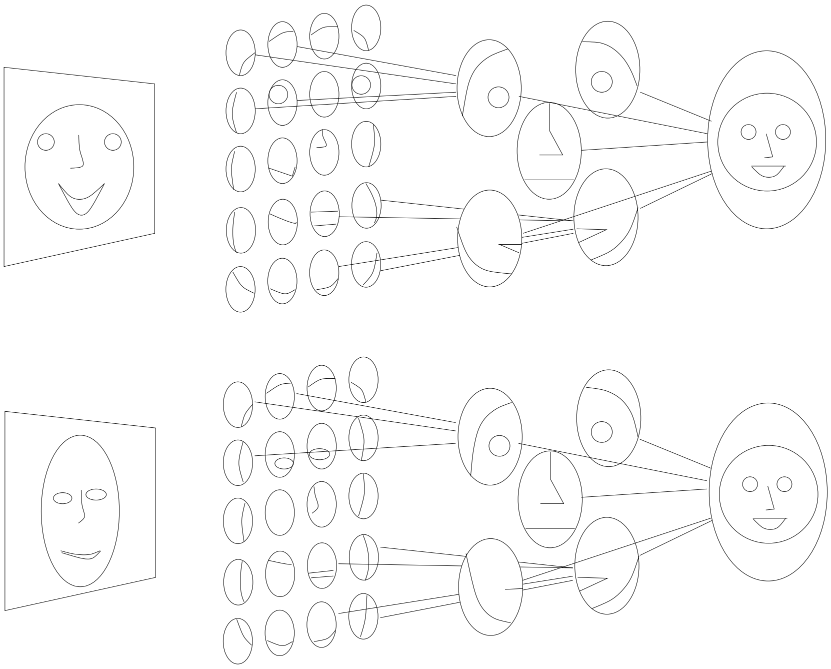
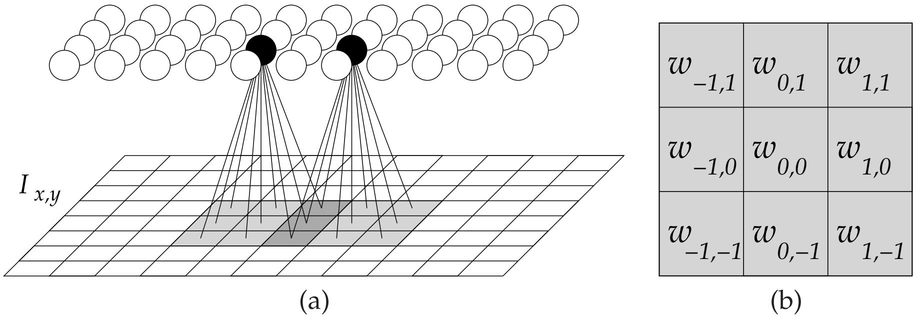


Figure 1.8: *Hierarchical feature detectors facilitate invariant object recognition. Small invariances in each hierarchy level result in a large invariance of the whole system.*

possible views of, say, a face *in terms of pixel values* would require prohibitively many rules of the form: ”If pixel A is brighter than pixel B and pixel C is approximately as bright as pixel D, and ... then the shown object is a face”.

One approach of tackling this combinatorial explosion is to use hierarchical structures [65, 48], where complex features are inferred from the presence or absence of many simpler features (see Figure 1.8). The intuitive idea is that the visual representation of a natural object is composed of a number of smaller shapes which, each taken by themselves, appear more invariant under transformations than the entire object as a whole. In each level of the hierarchy, decisions are based on abstract concepts found in the previous level rather than on raw pixel values. A face, for example, usually consists of two eyes, a nose and a mouth, appearing in a defined relative arrangement. Once it is known that a mouth is present somewhere in the lower part of the picture, the raw pixel values with could have produced this information are not longer of importance.

When using the hierachical approach, only a relatively small amount of invariance must be computed in each hierarchy level. If local feature detectors show invariance against small shifts, this will result in larger shift tolerances in higher layers, and eventually in higher-order invariances like tolerance against scale or deformation (section 1.2.5 and [11]).



$w_{-1,1}$	$w_{0,1}$	$w_{1,1}$
$w_{-1,0}$	$w_{0,0}$	$w_{1,0}$
$w_{-1,-1}$	$w_{0,-1}$	$w_{1,-1}$

Figure 1.9: **(a)** *A convolutional feature-plane: A grid of neurons with identical weights, receives connections from shifted, partly overlapping, input regions. Only the connections of two sample neurons are drawn.* **(b)** *The synaptic weights define the convolution kernel.*

1.2.3 Neural Implementation of Convolutional Filters

Convolution operations are widely used in the field of computational image processing for extracting local information, e.g., edge positions, from digital images. In this context, convolutions are also referred to as *linear filters* or *local neighborhood operations* ([23] for an introduction). Convolutional neural networks use hierarchical sets of convolutions to detect not only edges, but all sorts of abstract shape information, generally called *features*, in a given input image (hence the name).

In convolutional neural networks, convolutions are implemented by particularly interconnected ensembles of neurons. Figure 1.9a shows the essential neural structure: a so-called *feature-plane*. All neurons in the plane are equal with respect to their synaptic weights but they receive their input from shifted, partly overlapping, local regions in the previous network layer. The input region is often chosen as a square with an odd-numbered side length $S \equiv 2s + 1,\ s \in \mathbb{N}^+$. For example, the input region in Figure 1.9b has $S = 3$, or equivalently, $s = 1$. Let us, for a moment, refer to the synaptic weights by double indices according to their spatial arrangement, as shown in Figure 1.9b. Then, in accordance with equation (1.2), the output of a neuron at position x, y writes

$$O_{xy} = F\left(\sum_{i=-s}^{s}\sum_{j=-s}^{s} w_{ij} \cdot I_{i+x,j+y}\right), \tag{1.7}$$

where $I_{x,y}$ is the two-dimensional field of input nodes. The term in parentheses is equivalent to a two-dimensional discrete convolution applied to the function I, using the convolution kernel w (cf. [23]). Thus, a feature plane computes a convolution with the input data, and additionally scales the output by the (generally non-linear) scalar function F.

The fact that all neurons in a feature plane are equal in terms of their weights is a form of the general concept of *weight sharing*, cf., section 1.3.1. Weight sharing drastically reduces the free parameters in a neural network: Although a feature plane has a lot of computable input connections, the actual number of weights to be adjusted during training is only a small fraction thereof.

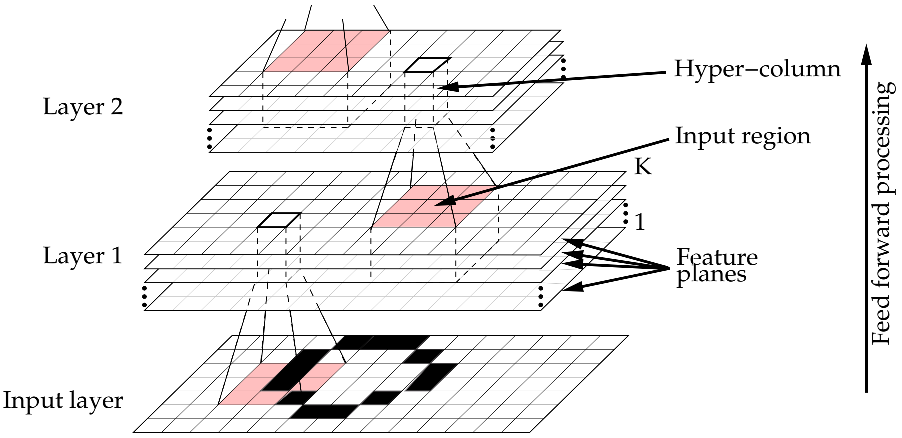


Figure 1.10: *In a convolutional network, each layer consists of a set of feature planes. Each feature plane detects a different feature. A given neuron generally receives input connections from all feature planes in the previous layer.*

1.2.4 Hierarchical Sets of Convolution Filters

Figure 1.10 shows how the convolutional feature planes are used in a complete convolutional neural network. A network layer consists of many equal-size feature planes, each detecting a different feature. An exception is the input layer, which has only a single plane representing the pixels of the image to be processed. The neurons in one layer at the same grid position, but belonging to different planes, are referred to as a *hyper column* throughout this thesis[2]. Neurons usually receive connections from all (or, at least most [30]) planes in the previous layer. The neural activities in a given hyper column form a feature vector, where each component indicates whether the corresponding feature is present in the image at that position. Obviously, the first network layer is transforming the raw image pixels into a topology-preserving feature map. The next layer transforms this feature map into another feature map with a higher degree of abstractness, and so on. Accordingly, the shape features detectable in a given layer are generally composed of features detected in the previous layer. Traversing up the layer hierarchy, more and more complicated shapes can be recognized.

A neuron, depending on its threshold (generally: depending on F in equation 1.7), is usually not only selective to a single point in the input space, but responds to a range of input vectors within an extended volume. This property is also referred to as *generalization*. A neuron in the first network layer will therefore not only fire for one exact pixel combination being present in its input region, but also for image patches whitch look similar. The same holds for neurons in higher layers, where the input consists of abstract features. This neural generalization provides the foundation for the hierarchical principle of invariant recognition discussed in section 1.2.2.

[2] naming after [10]

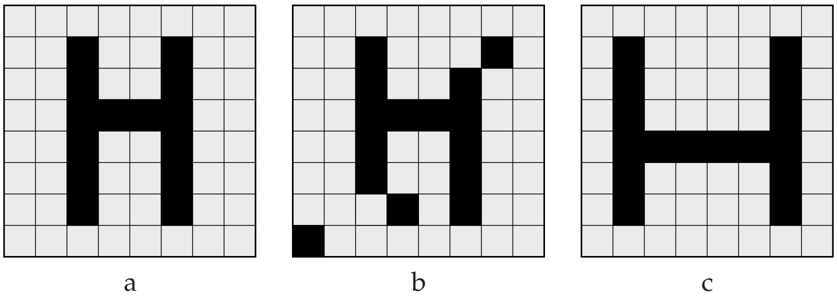


Figure 1.11: *Images* a *and* b *have most pixels in common, so they are close to each other in pixel space. On the other hand, the pixel distance between* a *and* c *is very large, although, on an abstract level, the shapes look similar as well.*

1.2.5 Boosting Invariance by Blurring and Sub-sampling

The invariance computable by one single neuron is restricted to certain modes of variations. Consider, for example, a neuron connected to each pixel of an 8 x 8 image, tuned to respond maximally to the shape "H" shown in Figure 1.11*a*. Presenting this neuron the image in Figure 1.11*b* will result in a similarly strong response, because most pixels are equal in both pictures. In contrast, image *c* will yield a very weak neural response, although, in terms of abstract shape features, the shown object has also much in common with image *a*. We may conclude that a convolutional network as described in the previous section might well cope with noisy images or slight illumination changes, but not at all with shift, scaling, or rotations of the whole shape or parts of it. This investigation gives rise to the introduction of special blurring layers (equivalently called subsampling layers), which in fact generate the real abstraction power of convolutional neural networks.

A blurring layer is very similar to a recognition layer, except for three differences: First, a blurring layer has the same number of feature planes as the preceding layer, and only corresponding planes are connected (e.g., neurons in the third feature plane in the blurring layer receive only inputs from the third feature plane in the preceding layer). Second, the convolution kernels implement blurring filters. In the most simple form, all weights are equal positive numbers. Third: The spatial resolution of the feature planes decreases in the blurring layer. This is usually accomplished by sub-sampling the feature plane grid, i.e., by discarding some rows and columns of neurons.

In a typical convolutional neural network, recognition layers and blurring layers are arranged in alternating order. When a feature is detected at one position in the recognition layer, an extended neighborhood around this position will be active in the following blurring layer. This in turn results in shift-invariance in the subsequent recognition layer. Putting it in other words: As soon as a feature is detected, its exact position does not matter any longer. While traversing up the layer hierarchy, concrete positional information will be gradually transformed into abstract, position-invariant information. Figure 1.13 illustrates how local blurring is used detect deformed instances of the same shape.

A link to the biological paradigm should be emphasized here. A blurring

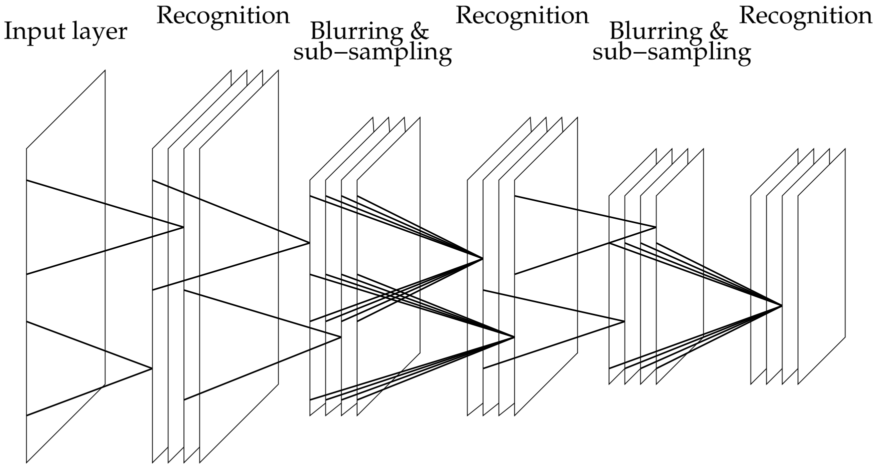


Figure 1.12: *A complete convolutional neural network consists of recognition layers and blurring layers in alternating order. The presence of blurring layers is essential for invariant recognition. Neurons in the blurring layers only receive inputs from the corresponding feature plane in the preceding layer.*

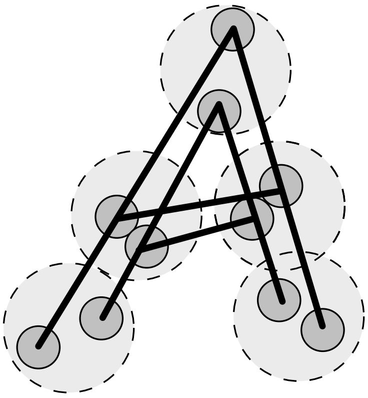


Figure 1.13: *Local blurring can create global invariance. Solid circles represent local features detected by S-neurons. Due to blurring, deviations within dashed circles are tolerated. At the blurred level, the original and the deformed version of the letter "A" can be recognized by the same neuron.*

layer acting on top of a recognition layer exhibits some analogy to the relationship between simple and complex cells observed in the visual cortex (cf., Figure 1.6). According to the terms *simple* and *complex*, recognition layers are often referred to as "S"-layers , blurring layers as "C"-layers in the literature [10, 41].

1.3 Training Methods

The previous section was concerned with the topology and functional principles of convolutional neural networks. However, before being applicable to a given problem, a neural network must be *trained*. in the context of neural networks, *training* usually refers to the adjustment of the synaptic weights with the goal to reach a desired network behavior. Often, training methods are based on a cost function

$$\mathcal{C}(\mathcal{I}, \{w_i\}) \tag{1.8}$$

which depends on a set of training input patterns $\mathcal{I}$ and the weights $\{w_i\}$. It measures the discrepancy between the actual network behavior and the ideal, desired network response. The training algorithm tries to find the set of weights such that $\mathcal{C}$ is minimal ($\mathcal{I}$ is usually constant for a given problem).

In this section, we will only discuss training methods that have been previously applied in the special field of convolutional neural networks. Comprehensive reviews of training algorithms for neural networks in general can be found in text books, e.g., [17].

Before turning to concrete training methods in sections 1.3.2 *et seqq.*, some general considerations about the dimensionality of the problem are of interest.

1.3.1 The Curse of Dimensionality

A typical convolutional neural network possesses a huge number (in the order of 10^6) of synaptic connections. Training a network like this corresponds to finding the global minimum of a function of 10^6 variables, where the function is supposed to be highly non-linear. According to Bellman's notion of the "curse of dimensionality", such a high-dimensional search space is virtually not coverable within reasonable time and resources, considering the exponential growth of the hyper-volume with the number of space dimensions [2]. Training is also impractical from another point of view: Any machine-learning system suffers from the effect of overfitting if the number of free parameters is in the order of, or exceeds, the number of training samples (e.g., [3]). An overfitted system does not generalize, i.e. it may work well on the training set but it will fail when confronted with input patterns not seen during training. For many visual recognition tasks it is difficult to get hold of a sufficiently large number of training samples for proper generalization.

Thus, methods are required for large networks to restrict the search space in advance. Three important strategies, applicable to convolutional networks, shall be mentioned:

1) Weight Sharing This strategy comes for free in a convolutional neural network. The convolutional nature of a feature plane requires that all its neurons have equal weight vectors, in other words: they share the same weights. So, even though there are a lot of computable connections present, the number of actual free parameters is significantly reduced. For each feature plane, only one representative weight vector is necessary. Besides simplifying the training, the shared weights also turn out to enable fast evaluation in the analog hardware environment evaluated in this thesis (cf., section 6.1).

2) Divide and Conquer Even after taking into account the reduction of parameters by weight sharing, a convolutional network can easily possess a few thousand independent weight values. Although nowadays all these parameters can be trained at once by brute computing force (taking hours or even weeks for one training run, [32, 60]), many approaches, including the one described in this thesis, make use of divide and conquer techniques. The network is split into multiple sub-networks which can be trained independently. Assuming that the complexity of training depends exponentially on the number of free parameters, but only linearly on the number of sub-networks to be trained, the divide and conquer approach will greatly simplify computations. However, the challenge remains to find appropriate partitions, such that both the sub-networks are trainable in isolation, and the overall network will be still be close to optimal in the end.

One common partition policy is to divide the network vertically, i.e., to train the network layers independently, one after another. As a consequence, the global network task (e.g., the correct classification of images) cannot directly guide the training, since the global error can only be assessed in the topmost network layer. Therefore, intermediate layers are either trained for solving pre-defined sub-problems [11], or unsupervised methods are employed [10, 67, 41]. Another possibility is to divide the network horizontally. Examples are *ensembles of experts* [17] or partitioning a many-class problem into many 2-class problems [22]. As an extreme example of divide and conquer, one can train each single neuron in a network separately [11] which, however, shifts a large part of the training intelligence from machine learning to the skill of the human operator.

3) Self-Organization The term self-organization refers to phenomena where simple, locally determined processes produce a complex global order. Applied to neural networks, this means that a neuron, or a neighborhood of near-by neurons, change their weight values depending on local parameters as for example their own current activation. An example for a local weight update rule is Hebbian-type reinforcement learning (equation (1.5)). In terms of the principle of splitting the global problem into many local problems, self-organization is similar to the divide and conquer approach outlined above—with the crucial difference that no cost-function (1.8) is explicitely given. Rather, the network converges to a state implicitly defined by the used local learning scheme. Of course, for many problems it is difficult—if not impossible—to define local update rules which result in the desired global network behavior. However, some data processing tasks can be very well accomplished by self-organization techniques. This is done also in this thesis (section 3.3).

1.3.2 Supervised Approaches

Supervised training methods subsume all strategies which rely on some sort of external teacher who has sufficient *a priori* knowledge of the problem to be solved.

Manual Weight Adjustment The most basic, albeit very laborious, supervised "training" method is to set each and every weight value by hand. This method requires detailed knowledge about both the problem and about the strategy by which the network will solve it. With convolutional neural networks, the solution strategy is known: Detect simple shape features in the first layer and more complex features in subsequent layers. So, given a hierarchical set of features, the corresponding network weights can be constructed analytically. The actual work consists in defining the features detectable in each layer. This task requires a high degree of skill and intuition of the human supervisor. Usually, a lot of trial-and-error adjustments are involved, since for many problems it is not obvious which features are optimally suited for the detection of the object(s) in question. Nevertheless, some authors were able to achieve sound results with this method [11, 41].

Interactive Feature Learning This method adds some level of automation to the plain manual setting of weights discussed above. The supervisor is still responsible for defining the features detected in each layer, but it is no longer necessary to think of raw pixel representations and to specify individual network connections. Instead, the supervisor marks examples of the features in the training images (e.g., using a mouse pointer on a graphical display), and some sort of automatic learning identifies the optimal synaptic weights. In one of Fukushima's approaches, the supervisor explicitely appoints a single neuron for each example, which is to alter its weights [11], i.e., the supervisor assigns features to neurons beforehand. In other techniques, a winner-take-all learning scheme automatically finds a suitable feature-to-neuron mapping, and additionally may insert new feature planes as deemed necessary [64, 67]. Training normally proceeds bottom-up, i.e., complex features are learned after the training for simple features is complete.

Global Training With Back-Propagation Back-propagation [51] is one of the most prominent automatic supervised methods applied in the scope of neural networks in general. It belongs to the family of gradient-based optimization methods. A set of training input patterns $\mathcal{I}$ for which the correct network response is known in advance must be given as a prerequisite. Then, the cost function is defined as a distance measure between the desired (O_d) and actual network output (O_a):

$$\mathcal{C}(\mathcal{I}, \{w_i\}) = \sum_{j \in \mathcal{I}} |O_d(j) - O_a(j, \{w_i\}|^2. \tag{1.9}$$

In each iteration, the weights of the network w_i are updated by following the negative gradient of the cost function:

$$w_i \leftarrow w_i - \epsilon \frac{\partial \mathcal{C}}{\partial w_i} \quad , \quad \text{for each } i. \tag{1.10}$$

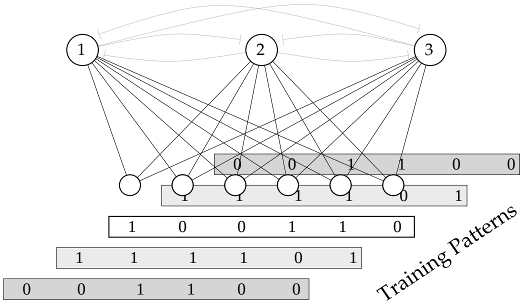


Figure 1.14: *Competitive learning. n neurons are connected to the same inputs ($n = 3$ in this drawing). For each training pattern, only the neuron with the strongest response is reinforced. Sometimes this winner-take-all policy is implemented by lateral inhibitory connections (drawn shaded).*

By application of the chain formula, the gradient (1.10) can be evaluated not only for synapses belonging to output neurons but also for neurons in hidden network layers, provided the transfer function F in (1.2) is differentiable. This technique was termed "error back-propagation" by its inventor D. Rumelhart in 1986 [51]. In fact, back-propagation was the first algorithm to provide a universal formula to train a multi-layer neural network by global supervised training. For convolutional neural networks, this means that only the mapping from full-scale images to object labels must be given by the supervisor. All the details of the features detected in the hidden layers are learned automatically by the algorithm.

Examples of successful application of back-propagation in the field of convolutional neural networks include hand-writing recognition by LeCun and successors [30, 60] and face recognition by another research group [29].

Although back-propagation is extremely popular and yields successful solutions for many problems, it suffers from the following shortcomings: First, like any gradient-based method, it is prone to get stuck in local minima of the cost function. Second, back-propagation can get computationally very expensive for large networks. Training times as long as many hours or even days for just one training run are often reported. Third, as mentioned above, the neural transfer function (F in (1.2)) and its derivative must be exactly specified. These preconditions can be easily realized in software simulations, but they can constitute an issue in hardware implementations.

1.3.3 Un-Supervised Approaches

Un-supervised learning refers to training approaches which do not rely on *a priori* knowledge about the task to be solved. Only the implicit structure contained in the training data is used. Un-supervised methods are often used for dimensionality reduction or for the automatic identification of pattern classes implicitly defined by clusters. In the field of neural networks, un-supervised is also called self-organization and is often based on Hebbian-style reinforcement.

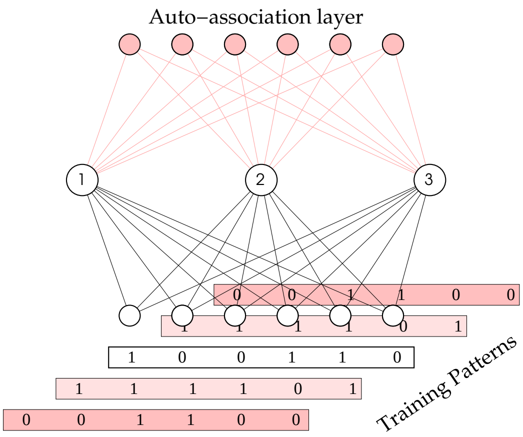


Figure 1.15: *Learning by auto-encoding: A 3-layer network is trained such that the temporary auto-association layer (shaded) reproduces the input for all training patterns. After training, the hidden neurons (1-3) represent the most significant components of the input data.*

Competitive Learning The term "competitive learning" was introduced by Rumelhart et al. in 1985 [50], denoting a learning scheme where only the most active neuron of an ensemble has its weight altered in each learning step. The other neurons do not change their weights in this step. If topological information is to be learned, neurons in the neighborhood of the winner neuron can be updated as well, as, for example, done in Kohonen's self-organizing maps.

The update rule itself is Hebbian-style reinforcement learning (cf. equation (1.4)), followed by some sort of weight normalization. All neurons in the ensemble receive the same inputs (see Figure 1.14). Ideally, for different training inputs, a different neuron is the one with the strongest activation, so, during training, every neuron in the ensemble gets the chance to be reinforced. After a neuron was reinforced, the next time a similar training pattern is presented, the same neuron will again be the one with the strongest activation. Gradually, the network will partition the input space into several classes, where each neuron responds to patterns from its corresponding class (see also Figure 3.2, p. 56). The success of this method can vary depending on the initial weight values and the order in which the training patterns are presented. Competitive learning is one example of vector quantization and it is very similar to the *K-means* clustering method.

In his early work, Fukushima used a kind of competitive learning in his "Neocognitron" in order to identify suitable feature sets detected by the feature planes [10]. The training proceeded sequentially layer after layer. Fukushima focused on supervised methods later because better results could be obtained. Competitive learning is a key technique for the training procedure described in this thesis (see section 3.3).

Auto-Encoding / Principal Components Like competitive learning, auto-encoding is a method for dimensionality reduction. The aim is to transform the N-dimensional input data into M-dimensional feature vectors ($M < N$) while preserving as much of the information as possible. In Figure 1.15, the 7-neuron input layer is to be reduced to a 3-dimensional feature vector defined by the activity of the neurons labeled "1", "2", and "3". During training, a temporary N-dimensional association layer (shaded) is introduced, and the resulting feed-forward multi-layer network is trained to reproduce the input as close as possible. For this purpose, back-propagation is used in [41], but other training methods are possible. After the training, the association layer is removed. This training strategy ensures that a maximum of the information is preserved in the middle layer. It can be shown that for a linear transfer function F (cf., equation (1.2)) the activities of the N neurons in the middle layer correspond to the first N principal components of the training data [3]. In [41], auto-encoding is used to identify the features for the hidden network layers.

Reinforcement Without Competition Some authors use self-organization by reinforcement where, in contrast to competitive learning, the reinforcement is not limited to the most active neuron in an ensemble. The decision whether a neuron is reinforced by the current input or not depends on whether its activation exceeds a given threshold, either a global or a dynamically adjusted one. Such a technique is useful if it is not known in advance how many distinct features are needed in each layer (adaptive network size), or if multiple objects should be detectable simultaneously in the same field of view. Examples are the Cresceptron [67], the multiple-object recognizing network of McQuoid [38], or the association network by Teichert [64].

1.3.4 Hybrid Approaches

Hybrid approaches, mixing supervised and un-supervised methods, are often utilized. One approach is to train the lower convolutional network layers by self-organization, and to train a classifier layer on top by a supervised method. The classifier can be either another neural network layer [41] or other non-neural machine learning systems [32, 29]).

An interesting hybrid approach is described by Weng et al. [67], where the supervisor marks high-level objects in example images and assigns them class labels. For each new class label, a new feature plane is added in the top-most network layer while a self-organization process based on the marked input field alters the structure of the network below. Thereby, the number of feature planes in each layer can grow adaptively when new features are detected. Training is based on bottom-up analysis rather than on back-propagating errors. Similar approaches are described in [38, 64]

In this thesis, the hidden layers are trained by self-organization without any explicit class information, and only the output layer is trained by a supervised optimization method.

1.4 Analog VLSI[2] Implementations

1.4.1 Motivation

The majority of today's computing machines rely on digital general-purpose processors, mostly based on the von Neumann architecture [42]. In contrast, analog electronic circuits are not widely applied for complex computing tasks[3] although they have the potential to offer advantages in various terms (cf., [33, 26, 66]):

Space efficiency. In analog computing circuits, basic laws of physics are exploited for performing calculations. Examples are the Kirchhoff rules for currents or the electrical properties of solid matter boundaries (see [66] and section 6). In contrast, digital solutions operate on abstract symbols represented within a complex electronic machinery. This difference makes analog VLSI implementations usually occupy way less silicon area compared to equivalent digital devices.

Power efficiency. For the same reason, many analog solutions consume less power than a digital implementation performing the equivalent computation. This holds especially for so-called *sub-threshold* designs where currents are low and are determined by Bolzmann statistics.

Massive parallelization. Space and power efficiency enable the design of massively parallel implementations where a large number of equivalent computing elements are integrated on one micro chip.

However, when designing analog computers, the following drawbacks relating to the analog nature must be taken into account:

Limited computing precision. Quantities coded by analog signals are inherently limited in precision. An upper limit of accuracy is constituted by noise present in the system, e.g., by thermal fluctuations or electrodynamical side effects, e.g., crosstalk. Other sources of uncertainty are random device variations which are characteristic for the manufacturing process.

Limited application scope. Not all types of applications are equally well suited for an analog implementation. First, precision requirements must match the capabilities offered by the used analog system. Second, in applications relying on algebraic computations, parallel processing can only be exploited if the problem is partitionable into independent, simple, computing steps.

Feed-forward neural networks seem to be a promising application for analog computing devices. The high degree of regularity and the natural fine-grained partition into independent computing units makes a parallel implementation straight-forward. This is true even more in the special case of convolutional networks (see next section).

[2] Very Large-Scale Integration

[3] involving more than a single multiplication or addition

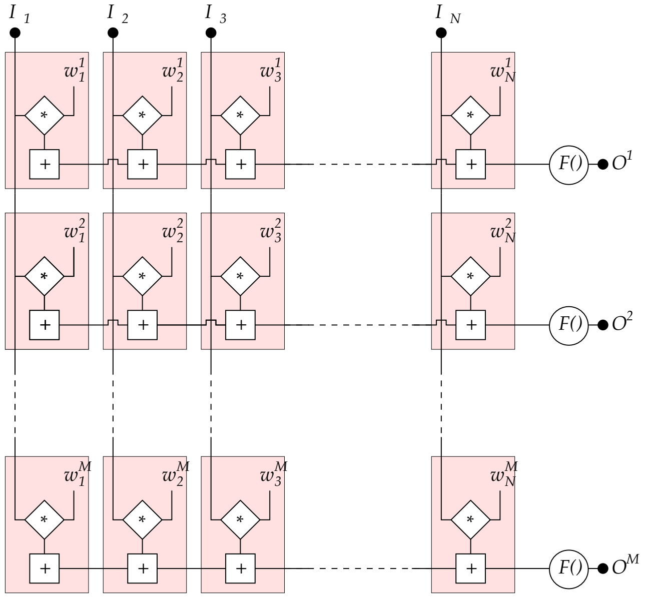


Figure 1.16: *2D parallel computing array. Each of the M rows constitutes a neuron with N inputs. The input signals $I_1 ... I_N$ are shared by all neurons. Shaded boxes represent the atomic multiply-and-accumulate elements, the synapses. The weights are stored in the synapses. The non-linear scaling function $F()$ is computed once per neuron.*

Whether limited computing precision is an issue remains to be assessed in each individual case. Techniques of limiting the precision problem exist on the hardware level, e.g., using analog computing but digital signal transmission [16, 4, 54], or incorporating calibration routines ([22] and section 6.2.3). It is a main contribution of this thesis to tackle the issue of precision on the training level. Methods are developed which yield reliable networks independent of hardware inaccuracies (chapter 5).

1.4.2 Massively Parallel Computing Arrays

As pointed out in the previous section, massively parallel computing is one of the applications where analog VLSI technology can promise efficient alternatives to digital processors. Neural networks are particularly well suited for parallel implementations, since (at least for layered neural networks) each neuron can be computed independently. Moreover, even the computation within a single neuron can be further parallelized. Recall that a neuron computes the dot-product with a given weight vector and scales the result (equation 1.2 repeated):

$$O = F\left(\sum_{i=1}^{N} w_i \cdot I_i - t\right) . \tag{1.11}$$

The N multiplications $w_i \cdot I_i$ can be performed independently and can thus be processed by parallel computing elements. Depending on the number of inputs N, this can imply a significant performance gain compared to sequential processing. Kramer states in [26] that the most dense arrangement of such multiplication units is a 2-dimensional array, as shown in Figure 1.16. The multiplication units (synapses) are laid out in a grid, where the shared inputs are along one dimension ($I_1 ... I_N$), and the outputs are along the other dimension ($O^1 ... O^M$). Each synapse holds a locally stored (but re-configurable) weight and thus implements a *single-instruction multiple-data* element.

If reconfiguring of weights is to be avoided during the evaluation phase, one row must be reserved for each neuron in the network. Convolutional neural networks usually possess a huge absolute number of computable neurons, but due to *weight sharing* (p. 20), large groups of neurons have identical weights. If a distinct computing row is reserved only for every *unique* neuron in the network, the required array size is thus limited to practical dimensions. This makes convolutional networks especially well suited for parallel array implementations. Indeed, all serious applications of array-based analog VLSI known to the author are located in the field of image processing and involve computing convolutions [5, 26, 31, 37].

1.4.3 Recent Array-Based Neuro Chips

A plethora of analog neural network implementations have been developed by various research groups. Rather exhaustive overviews are given in [1] or [33]. Here, we will review some of the most-cited array-based VLSI solutions of the recent years. Some of them are reviewed also in the book [68] and the paper [25]. All these chips have configurable weights, but they do not include learning circuitry, so training is done either completely in software or in a closed software/hardware loop. Except for the ETANN chip, which is purely analog, the described chips combine analog computing with digital data transmission. This *mixed-signal* approach enables convenient interfacing by standard digital hardware and confines the noisy analog calculations into closed units.

ETANN (1989) The first commercially available analog chip was the Intel i80170NX, or ETANN (Electrically Trainable Analog Neural Network), containing 64 neurons and 10280 weights (there are 128 inputs plus 32 bias weights per neuron). Multiple configurations including multilayer networks using internal feedback can be realized. The array can be split into two 64x64 banks. The programming of the weight values is very slow compared to the other chips drescribed below. The ETANN chip was used in various commercial devices, including a music synthesizer.

NET32K (1990) The NET32K chip [16], developed by AT&T, implements 256 neurons with 8,192 synapses. The chip uses only 1-bit resolution for weights and data, but several synapses can be used in conjunction to realize effectively up to 4-bit weights and/or data. The chip was used for example on a commercial extension board for use with standard workstations [5]. The board, containing 2 NET32K chips and digital support logic was used to accelerate convolutions for image processing tasks. An interesting development by the same research group was a similar architecture by Satyanarayana et al. (1992) which allowed for more flexible connection topologies by equipping each synapse with an own neuron body circuit.

ANNA (1992) Another development by AT&T is the ANNA chip (Analog Neural Network Arithmetic) which implements 4,096 synapses which can be configured as 8 neurons with 256 inputs each [4]. Other configurations with 64 or 128 inputs per neurons are possible as well. The activation function is piecewise linear, roughly approximating a sigmoid. The digital interface allows 6 bit resolution for the weights and 3 bit for the input/output data lines. The ANNA chip was used for the evaluation of convolutional networks for hand-written character recognition [52].

MoneyPen (1999) One example of a VLSI neural network development which matured into a commercial product is the chip presented by Masa et al. [37]. The application-specific chip has a fixed connection topology with three different-sized synapse arrays corresponding to three convolutional network layers. Image data is directly fed in from an optical sensor. The system is marketed as a mobile character recognition system for check reading ("MoneyPen") through a company in Switzerland (`www.csem.ch/fs/microelectronics.htm`).

HAGEN (2003) The HAGEN chip, developed by J. Schemmel in Heidelberg (HAGEN = Heidelberg AnaloG Evolvable Network) [54], implements 256 neurons with 128 inputs each. The neurons are arranged in 4 synapse arrays of 64 neurons. Input/output lines have 1 bit precision, weights can be written with 10 bit precision (+ 1 sign bit). Of all reviewed chips, HAGEN is manufactured using the most modern technology and outperforms its predecessors in terms of operating speed, weight configuration speed, and power efficiency. This chip is described in more detail in section 2.2.1. It is used for the hardware experiments in chapter 6. Previous studies for utilizing this chip for image recognition are described in [43].

Chapter 2

Working Environment

This chapter covers the software tools and the hardware equipment used for conducting the experiments described in chapters 3 through 9. The software environment, comprising more than 100,000 lines of code, has been created in the course of this thesis, in collaboration with other members of the research group. The prototype neuro chip and a PC-based control system were available as the result of previous research.

2.1 Software

The development of new algorithms commonly involves a considerable amount computer programming. Constantly, new ideas must be implemented, tested, and refined. In order to spend as much time as possible on conceptual research rather than on crafting machine code, programming should be made as simple as possible. Moreover, the software produced should exhibit a rich set of convenience features, and also facilitate documentation of the research history. Features considered most important are summarized here:

Modularity It should be easy to add and remove functional modules to and from the software. Different variations of the same module should be easily interchangeable. Standardized ways for interaction among modules are required.

Runtime Control / Interactivity Interactive use of the sofware is desirable. For example, it may be necessary to test different parameter settings in order to get a feeling for resulting effects. This ideally involves graphical control and display.

Serialization and Logging In order to facilitate documentation and reproducibility, settings and results must be storable in a persistent form.

Automation In practice, a thorough evaluation of new algorithms requires automated control over the software, typically in the form of batch scripts.

Rapid prototyping It should be possible to implement new ideas fast and with a minimum of effort while still including all the above features. Ideally,

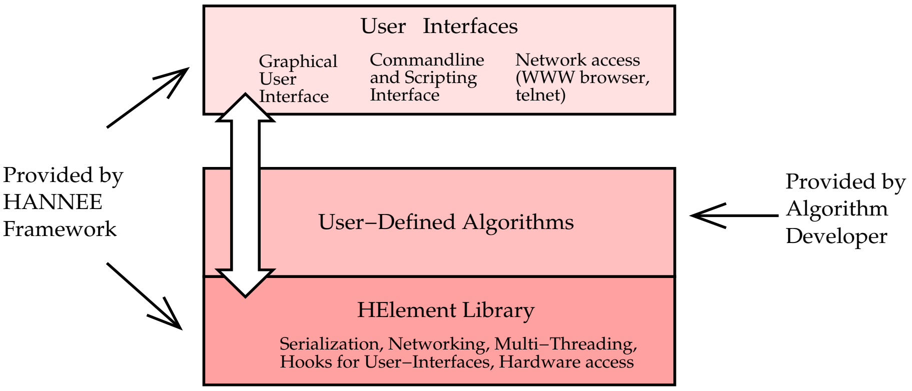

User Interfaces
Graphical User Interface
Commandline and Scripting Interface
Network access (WWW browser, telnet)
Provided by HANNEE Framework
User−Defined Algorithms
Provided by Algorithm Developer
HElement Library
Serialization, Networking, Multi−Threading, Hooks for User−Interfaces, Hardware access

Figure 2.1: *Overview of the HANNEE software framework. Higher-level components make use of the components below. The programmer focuses solely on algorithm development. User interfaces, hardware access, and other required functionality are provided by HANNEE.*

the researcher focuses on the algorithms rather than the implementation of convenience functionality.

Efficiency Last but not least, performance issues cannot be neglected. The speed of development progress is often limited by the computation time needed for testing algorithm implementations.

Commercial tools providing some of the desired features are available, e.g., MATLAB. Nevertheless, in order to remain most flexible and still being able to produce high-performing code, an own C++ software-framework, called HANNEE[1], was created in the course of this thesis. This happened in close collaboration with fellow lab members, most notably Steffen Hohmann. HANNEE provides a general basis for the development and the evaluation of new algorithms.

The overall structure of the software is shown in Fig. 2.1. The HElement library which constitutes the core of HANNEE consists of useful base classes and tools for developing algorithm modules. It realizes many of the desired features discussed above, ready to be used by the algorithm programmer, and it contains functionality particularly suited for interfacing the various custom-developed hardware devices used by the research group.

Another part of HANNEE provides various user interfaces for controlling the user-written algorithms graphically or text-based, possibly via a network connection. As long as the programmer makes proper use of the HElement library, the user interfaces are generated automatically without any further programming effort.

The rest of the software section covers the components of the HANNEE framework in more detail. Therefore, it is assumed that the reader is familiar with object-oriented programming techniques. For a comprehensive docu-

[1] Originally HANNEE was an acronym for “Heidelberg Analog Neural Network Evolution Environment”. In the meanwhile, it has developed into a general algorithm development environment.

mentation the reader is referred to the documents in the hannee++/docs/[2] and hannee++/hannee++-api/ directories and, of course, to the source code of the software.

2.1.1 The HElement C++ Library[2]

Library Features

The HElement library constitutes the core of the HANNEE software. It defines general functionality useful for algorithm development. Most of this functionality is contained in the classes derived from `HElement`. A class diagram is shown in Fig. 2.2.

`HElement` objects can be arranged in tree-like graphs. `HValues`, constituting the leaves, represent actual information, while tree nodes are represented by objects derived from `HGroup`. Each `HElement` has a name by which it can be identified, and it can be assigned a (human-understandable) description. It provides methods to manipulate itself and sub-elements through a string-based control API. Moreover, `HElements` can be set up to notify each other of certain events. All these functions will be covered in detail below. The particular relationship between `HElement`, `HValue`, and `HGroup` is a straight-forward implementation of the canonical *Composite* design pattern [15].

The benefits of using the `HElement` library are best illustrated by means of a demonstration. Therefore, in the rest of the software section, it will be shown how a simple sorting algorithm, depicted in Fig. 2.3, is implemented using the HElement library. This example will be discussed from several points of view, illuminating the various benefits of the library:

Runtime Control

Most algorithms depend on a set of parameters defining the actual behavior. Thus, when developing new algorithms, a range of parameter settings must be evaluated in order to assess a method's capabilities. The straight-forward (or, better: quick-and-dirty) method is to hard-code parameter values in the program source code. When exercising this practice, every time a parameter is varied, the program source must be changed, possibly requiring re-compilation.

[2] All paths refer to the workgroup's CVS repository. Please contact the author for material.

[2] This section is part of the software documentation. Knowledge about object-oriented programming techniques is assumed. The reader interested in the results may skip this section.

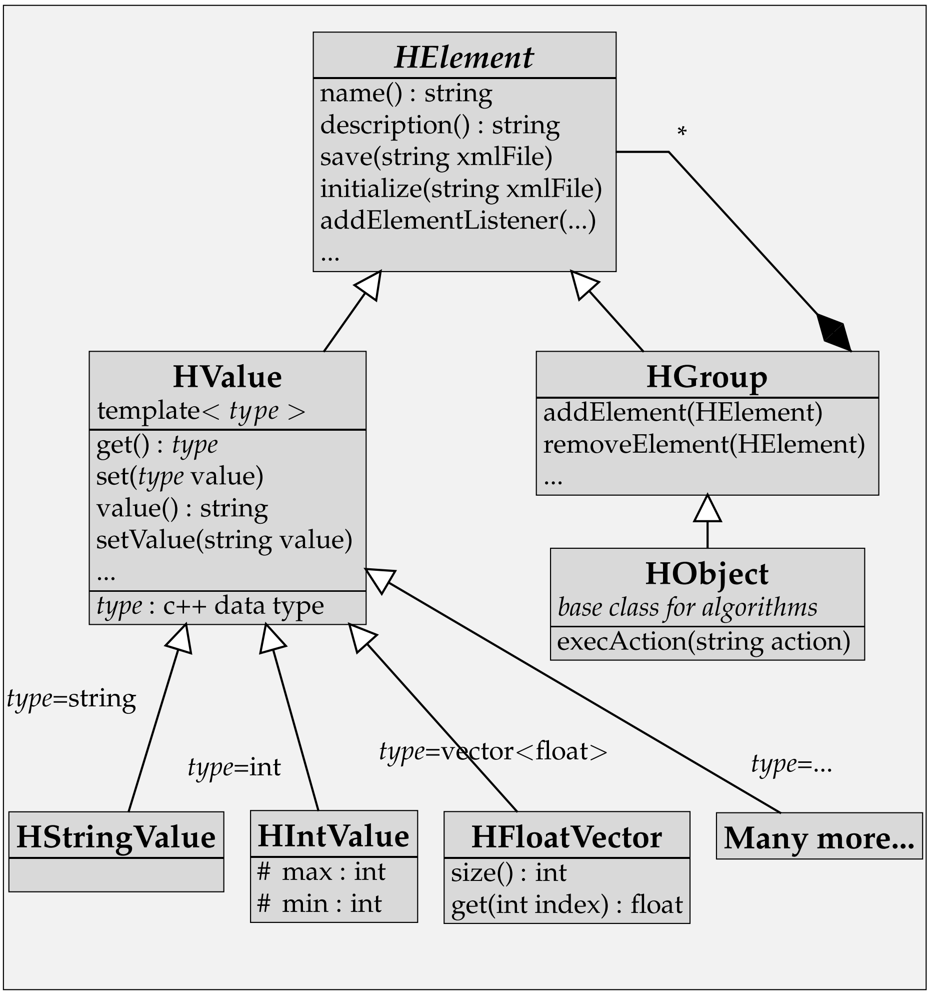


Figure 2.2: *Core classes of the HElement library.* `HObject` *serves as base class for new algorithms.* `HValues` *encapsule parameters. The* `HElement` *classes are a realization of the classic* composite *design pattern [15]*

```
Algorithm              Parameters

BEGIN                  - Input File
   LOAD data file      - Output File
   SORT data           - Data compression (yes/no)
   SAVE data file      - Column by which to sort
END                    - Sorting order (ascending, descending)
                       - Sorting method (Quicksort, Heapsort, ...)

                       ... more params, depending on sorting method
```

Figure 2.3: *Example sorting algorithm with parameters*

Therefore, parameters are preferably made specifiable at run-time: by command line parameters, via a settings file, or in some interactive fashion, e.g., through the command prompt or a graphical interface.

The HValue classes provide a convenient infrastructure for working with parameters that are changeable during run-time. The example sorting algorithm specified in Fig. 2.3 might be declared like this:

```
/* Listing 2.1:  Declaring an algorithm using HElements */
  class SortingAlgo : public HObject{
    // declare parameters as HValues
    HBoolValue* compression;
    HIntValue*  column;
    HChoice*    order;
   public:
    SortingAlgo(){
      // define parameters (name and initial value)
      compression=new HBoolValue("compression",true);
      column=new HIntValue("column",0);
      order=new HChoice("order","asc","asc,desc");
      // add them as children to this SortingAlgo
      addElement(compression);
      addElement(column);
      addElement(order);
    }
    void sort(){
       // do the work
    }
    ...
  };
```

SortingAlgo is declared to inherit HObject. As such, is it a HGroups and thus can have children. Instead of using native int, float, or std::string member variables, the algorithm's parameters are declared in the variables compression, column, and order which are of types derived from HValue. Each HValue encapsules one native variable which is accessed by get() and set() methods:

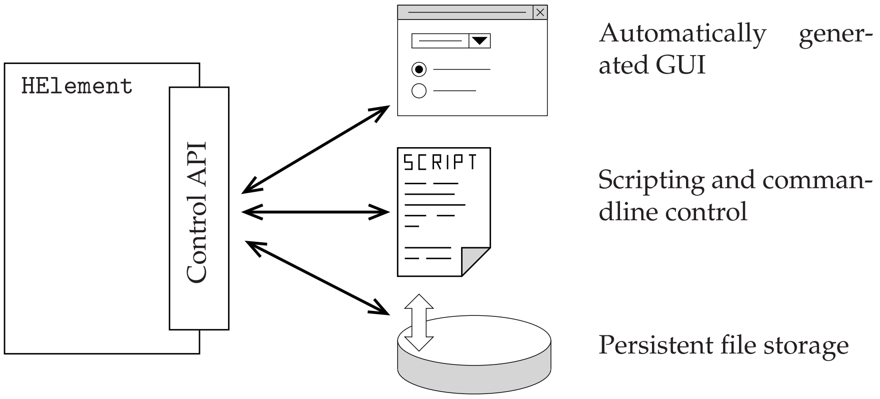

HElement
Control API
SCRIPT
Automatically generated GUI
Scripting and commandline control
Persistent file storage

Figure 2.4: *HElements provide a standardized control API which is the basis for automatically created graphical user interfaces, script control, or file I/O.*

```
/* Listing 2.2:  Set and get methods */

  void SortingAlgo::sort(){
    // ...
    if(column->get()>99)
      column->set(99);
    // ...
  }
```

Accessing HValues by their get() function is nearly as efficient as accessing a native variable.

The main feature is that HValues, like all HElements, provide a standardized control API which can be thought of as "control handles" to the outside world (see Figure 2.4). This API constitutes the basis for automatically generating interactive graphical user interfaces, for controlling the HValues via batch scripts during runtime, or for storing algorithm settings in structured data files. The string-based access functions setValue() and value() are example members of the control API. They allow to change, respectively read, all types of HValues via the same string-based interface:

```
/* Listing 2.3:  Control API example */

  SortingAlgo a;
  // string argument is parsed and converted to bool
  a.setValue("compression","true");
  // string argument is parsed and converted to int
  a.setValue("column","3");
  // value of column is returned as string
  std::string order=a.value("column");
```

Other control API methods, not shown here, include querying the type of an HElement, accessing children (for HGroup), or, in the case of HObject (see nect section), invoking defined actions.

Actions

Specifying parameters alone does not make a useful program. Functions must be invoked interactively, too. For this purpose there exists the class `HObject`, which is also the recommended base class for new algorithms. `HObjects` are `HGroups` with the additional feature that so-called *actions* can be declared. An action binds a name to a no-argument member function, by which this function can be referred to from the user interfaces. Actions are declared by macros, one in the header file, the other in the source file:

/* Listing 2.4: Declaring actions */

```
/// h-file ///
class SortingAlgo : public HObject{
  DECLARE_ACTIONS // this object defines actions
 public:
  void sort();
};

/// cpp-file ///
DEFINE_ACTIONS(SortingAlgo,HObject)
  ACTION("sort",sort) // bind name to function
  // ... more actions ad lib.
END_DEFINE_ACTIONS
```

The `DEFINE_ACTIONS` macro expects the class name and the name of the superclass as arguments. Figure 2.6 and the commandline interpreter example on page 45 demostrate how the `sort()` function can be invoked interactively.

Modularity

Using the `HGroup` element, which can have `HValues` or other `HGroup` objects as children, hierarchical tree structures can be arranged using the `HGroup::addElement()` function. Every element in the tree is uniquely identified by a hierarchical name of the form `grandparent.parent.element` with an arbitrary number of parent levels. These names can for example be used in the get/set functions introduced in listing 2.3. Thus, following object oriented design principles, semantically related entities can be pooled into integrated units.

Turning to our `SortingAlgo` example, let us assume the actual sorting method is to be encapsuled in such an independent unit. We define an abstract interface `SortingMethod`, which can be realized by various implementations. The `SortingAlgo` only interacts with the general interface.

/* Listing 2.5 */

```
// abstract interface:
class SortingMethod : public HObject {
  public:
     virtual void sort() = 0;
};

// one possible concrete implementation:
```

```
class HeapSort : public SortingMethod {
    // method-specific parameter:
    HIntValue* numHeaps;
  public:
    HeapSort(){
      numHeaps=new HIntValue("numHeaps",2);
      addElement(numHeaps());
    }
    // implement SortingMethod::sort()
    void sort();
};

// sorting algo class
class SortingAlgo : public HObject {
    SortingMethod* my_method;
  public:
    SortingAlgo(){
       my_method=getSortingMethod();
       addElement(my_method);
    }
    sort(){
       // call actual sorting method:
       my_method->sort()
    }
};
```

Note that implementations of `SortingMethod` may define own method-specific parameters (here: `numHeaps`). The HElement control API allows these parameters to be dynamically queried and accessed from the `SortingAlgo` without further programming effort.

Up to this point, the choice of the actual sorting method is done in the source code (for instance in the constructor of `SortingAlgo`). Using `HElements`, we can go one step further and choose the method dynamically at run-time. The necessary code is here:

/* Listing 2.6: Using HObjectChooser */

```
class SortingAlgo : public HObject {
    // object chooser for dynamically switching modules
    HObjectChooser* my_method;
  public:
    SortingAlgo(){
      // allow to choose among SortingMethod subclasses:
      my_method=new HObjectChooser("method",
         HObjectMap::classNames("SortingMethod"));
      addElement(my_method);
    }
    sort(){
       ((SortingMethod*)my_method->object())->sort();
    }
};
```

```
SortingAlgo a;
// sorting with HeapSort
a.setValue("method","HeapSort"); a.sort();
// sorting with QuickSort
a.setValue("method","QuickSort"); a.sort();
```

In the graphical interface (section 2.1.2), the `HObjectChooser` will appear as a drop-down box. The phrase `HObject::classNames("SortingMethod")` in the constructor tells the object chooser to allow all known sub-classes of `SortingMethod` as possible choices (this construct relies on the HObject runtime type information system, see page 42). For this purpose, the c-style cast in `SortingAlgo::sort()` is safe. As soon as the value of the `HObjectChooser` changes, a new object of the chosen class will be created (and added as a child of `SortingAlgo`). From within the `SortingAlgo`, the current `SortingMethod` instance is accessed via `my_method->object()`.

Serialization

The control API allows entire `HElement` trees to be converted into a serial form, i.e., a sequence of bytes, and restored again. This feature is useful for persistent archiving of experimental setups, but also for temporarily storing `HElement` structures in the system clipboard, enabling copy/paste functionality.

The *Extensible Markup Language* (XML) was chosen as the serial data format for its natural ability to represent tree-like data structures. Here is an example showing how our `SortingAlgo` object looks like in its serial form. Every `HElement` is represented by a corresponding XML tag:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<HObject class="SortingAlgo" name="SortingAlgo">
  <Bool name="compression" value="true" />
  <Integer name="column" value="3" />
  ...
  <HObject class="HeapSort" name="HeapSort">
     <Integer name="numHeaps" value="2" />
  </HObject>
</HObject>
```

Due to the hierarchical structure of the the XML language, old setup files remain readable even if the software has changed in the meanwhile. The deserialization routines ignore unrecognized XML elements, and data elements that are present in the software but which cannot be found in in the XML file are assigned default values. Compatibility of setup files between software versions turns out to be of great value while algorithms are still under construction.

Last but not least, XML files are human-readable, making in-place edits possible without the need of starting the software program.

The data overhead imposed by the XML data structure can result in relatively large file sizes. Therfore, if space requirements are an issue, the produced setup files may be compressed using the *gzip* method [14]. Input streams are automatically de-compressed if gzip-compression is detected.

Event Handling

The HElement library implements a modern, simple-to-use, event handling system. For example, whenever a `HValue` is assigned a new value it emits a signal which notifies registered callback object, so-called *listeners*, about this change. In general, `HElements` provide a standardized way to register user-defined callback functions which are notified in pre-defined situations. In terms of canonical design patterns, this technique corresponds to the *Observer* pattern [15].[3] An event handler must inherit the library class `EventListener`[4]:

/* Listing 2.7: EventListener */

```
class MyListener : public EventListener{
  public: void elementChanged(ElementEvent &e);
    // implement callback functionality here
  }
};
// ...
HIntValue wert("wert",0); // initialize with 0
MyListener listener;
// register the listener
wert.addElementListener(&listener);
```

In this example, the `elementChanged()` function of `MyListener` will be invoked every time the value of `wert` changes. The `ElementEvent` passed as argument to the callback function contains information about the type of the event (e.g., value changed, child added) and about the `HElement` which sent the event. An event can be marked as handled by calling `ElementEvent::consume()` in the callback function. This will prevent the `ElementEvent` to be sent to any further listeners. Listeners are automatically unregistered when the emitting element or the listener itself is destroyed.

If an `HElement` is part of an `HElement` tree, generated events are by default passed up the hierarchy. Therefore, a listener registered with the root `HElement` will receive all events generated within the sub-tree below. This feature is very convenient, but on the other hand it can produce a lot of overhead function calls when `HValues` are frequently changed. Methods are provided to temporarily suspend event sending within `HElement` sub-trees.

One important application of the event handling system is the separation of the graphical user interface (GUI) from the `HElement` worker classes. Since the communication between the `HElements` and the GUI is implemented via the standardized event system, in most cases the programmer of a new `HElement` does not need to care for user interface issues. Moreover, the `HElement` classes can be used with or without a GUI present. For more details see section 2.1.2.

[3] Not to be confused with the HANNEE class `Observer` which provides fast read-only access to internal algorithm parameters.

[4] Usage of the HElement event handling is similar to the event handling known from the Java(TM) programming language.

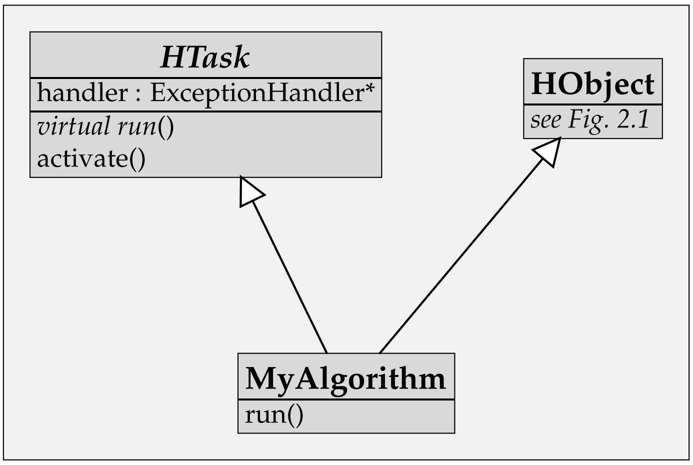


Figure 2.5: *Inheriting* `HTask` *is a convenient way to write multi-threaded applications. Invoking activate() will start the run() function of MyAlgorithm in a separate thread.*

Logging

A convenient and flexible logging system is provided. Messages of various types (e.g. warning, info) can be created and directed to files, onto the screen, or over network connections. The originating program module and the time of the message are automatically recorded in the style of the following short example:

```
14:50:15, SortingAlgo: calling sort routine...
14:50:15, SortingAlgo.Heapsort: starting sort
14:50:17, WARNING, SortingAlgo.Heapsort: data already sorted
```

In the source code, creating log messages is done by sending data to a dedicated `std::ostream` object named `log`. Each `HObject` has its own log stream object.

`/* Listing 2.8: Writing to the log stream */`

```
class HeapSort {
  void sort(){
     // ...
     log<<"data already sorted"<<hwarn;
     // ...
  }
}
```

Multi-Threading

The HElement library is designed to be thread-safe. To be precise, event handling, logging, HElement access, and the various user interfaces are correctly synchronized respectively serialized, and thus can be used within multi-threaded programs without interference A convenient interface is provided to set up new threads: If HTask class is inherited and its run() function is implemented, a call to activate() will start run() in an own thread (cf., Figure 2.5) . An optional exception handler defines how exceptions occuring during the execution should be treated. The default is displaying them to the user (see also the paragraph about exception handling below). Only a small set of rules must be obeyed when writing multi-threaded applications using the HElement library. Detailed information can be found in the file Multithread_HOWTO in the hannee++/ directory.

Exception Handling

The HElement library provides a convenient way to report runtime errors to the user. Within algorithm source code, objects of type Exception can be thrown to indicate an erroneous situation:

/* Listing 2.9: throwing an Exception */

```
void sort(){
  if(column->get()<0)
    throw Exception("Negative column index in sort()");
  // ...
}
```

If an exception is encountered, the normal program flow will stop and the program will return to the point of the most recent user interaction. Before user interaction can be resumed, the the error message will be by default presented to the user in a way depending on the context: If the graphical user interface was used to issue the most recent command an error dialog will pop up showing the message. If the user is interacting via the command line interpreter the error message will be written to the terminal together with the source (e.g, file name and line number) of the erroneous line. More details on error reporting can be found in the document hannee++/docs/commandlinemode_doc.pdf. In addition to the default behavior, user-defined exception handlers can alter the way exceptions are handled.

Run-time Type Information

Many of the HElement convenience functions, most notably: serialization, dynamic class instantiation, and automatic generation of user interfaces, rely on run-time type information (RTTI). The particular class of an HValue object can be queried by using its tag() function. For HObject subclasses, a more elaborate RTTI mechanism based on the library class HObjectMap is provided. This class features static member functions for querying the class name of a given HObject and for obtaining information about inheritance dependencies. HObjectMap also serves as an object factory for instantiating objects by class name. In conjunction with the GuiMap class, the appropriate graphical representation for a given HObject can be determined.

For the RTTI to work, each newly implemented HObject subclass must be registered with the HObjectMap (similarly, each custom GUI must be registered with GuiMap). The registration process is very simple. It involves only a one-line definition of one global variable per

HObject, respectively per GUI class, indicating the class name and its inheritance chain. For example, to register the SortAlgo object defined in Listing 2.1, add the following line to the source file:

/* Listing 2.10: Registering HObjects */

```
#include "hobjectmap.h"
HObjectMapEntryImpl<SortALgo> _SortAlgo_("SortAlgo","HObject");
```

2.1.2 User Interfaces

A number of user interfaces to the HElement library exist, suited for different modes of operation. While for interactive work, the graphical user interface is the preferred choice, script control is appropriate for systematic, time-consuming experiments. On embedded systems not featuring any graphics support, the slim commandline interpreter can be used in interactive mode. For controlling the software in distributed environments, two network-based interfaces are provided.

Thanks to the HElement's control API (cf., page 36), all the mentioned user-interfaces are generated automatically at run-time for any HElement. The user interfaces allow to monitor and manipulate HValues and to invoke functions that have been declared as *actions* (see page 37). For most applications, these default interfaces are sufficient. Nevertheless, if special interface functionality is required, custom user-interfaces can be implemented for any HElement subclass.

Graphical User Interface

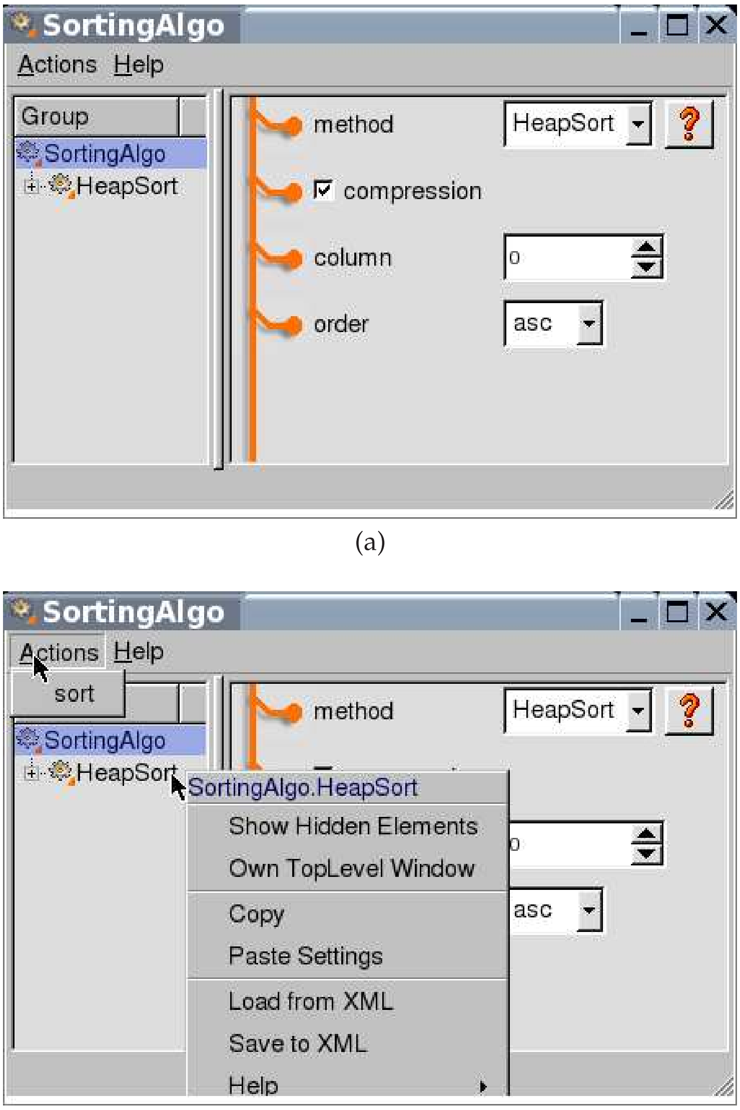
The graphical user interface is implemented on top of the commercial open-source Qt library [47] which is available for most common operating systems. The example in Figure 2.6a shows the screen representation of the SortingAlgo declared in listings 2.1 and 2.6. The hierarchical structure is reflected in the expandable tree-view to the left. The HValues of the currently selected (sub-)HGroup are displayed in a type-specific way (e.g., check-boxes for boolean values, text fields for string values) on the right-hand side. If actions are defined for the current object, they can be invoked via a drop-down or context menu (Figure 2.6b).

Commandline Interpreter

Additionally to the access to HElement functions, the commandline interpreter provided by HANNEE features control structures (for, while, if...else constructs), the definition of user-defined functions and local variables, inclusion of nested batch files, system calls, access to system environment variables, and many other convenience functions. In the following short demonstration, HANNEE is started in console mode,

```
fieres@botanik:~> hannee -c
Welcome to the Hannee Console!
H->
```

a new SortingAlgo is instantiated,



method HeapSort

☑ compression

column 0

order asc

(a)

(b)

Figure 2.6: *Automatically generated graphical representation of the example algorithm defined in listings 2.1 through 2.6. (a) (Sub-)* `HGroups` *are selected in the left area of the window. Parameters of the selected* `HGroup` *are displayed to the right. (b) Defined actions are invoked in the main menu. A context-menu is available for each* `HGroup`*, providing copy/paste functionality among other things.*

```
H-> set sortalgo new SortingAlgo()
```

its children are listed,

```
H-> sortalgo children
 Groups:
 -------
  HeapSort
 Values:
 -------
  method
  compression
  column
  order
```

and a few things are done with it:

```
H-> sortalgo compression = true
H-> sortalgo method = "HeapSort"
H-> sortalgo HeapSort numHeaps = 2
H-> sortalgo sort()
```

The commandline interpreter can be run either in interactive mode as shown above or in batch mode from a script file. Script files can be invoked from the interactive prompt, from the GUI, or scripts can passed to HANNEE as commandline parameters:

```
fieres@botanik:~> hannee -b myscript.txt
```

A more detailed documentation of the commandline syntax and a function reference can be found in the file commandline_doc.pdf in the hannee++/docs/ directory.

Network Access via Http and Telnet

A running instance of HANNEE can act as a network server, providing control via IP. The network interface has been designed to rely only on standard software on the client side, so no special application has to be installed on the terminal machine. The following commands set up HANNEE as a server:

```
fieres@botanik:~> hannee -c
Welcome to the Hannee Console!
H-> new Httpserver() connect()
Listening for connections on port 2080
H-> new CliServer() connect()
Listening for connections on port 2000
H->
```

Of course, the server modules can also be set up in GUI mode or directly from C++ code.

A remote console session is initiated simply by opening a telnet session on a specific network port to the computer running HANNEE (default port is 2000). Since the usual commandline interpreter (p. 43) is involved, the remote console

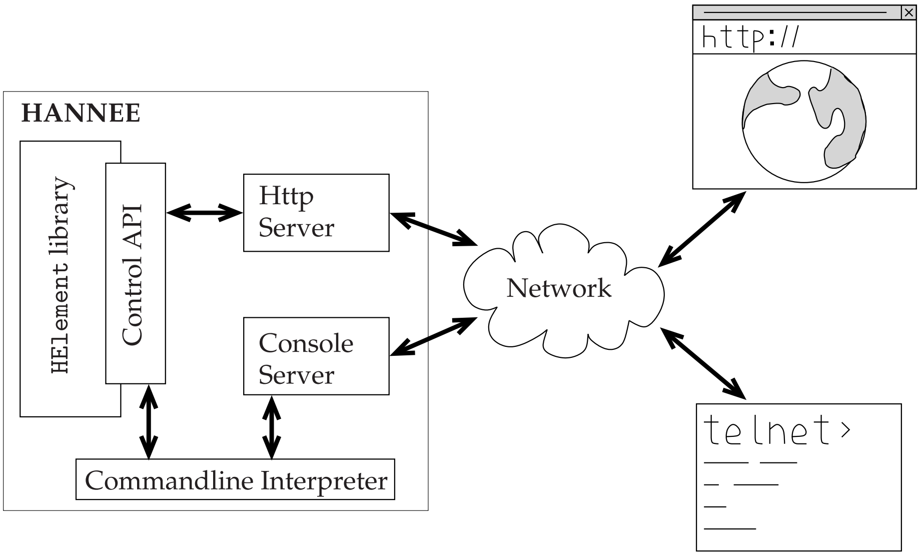
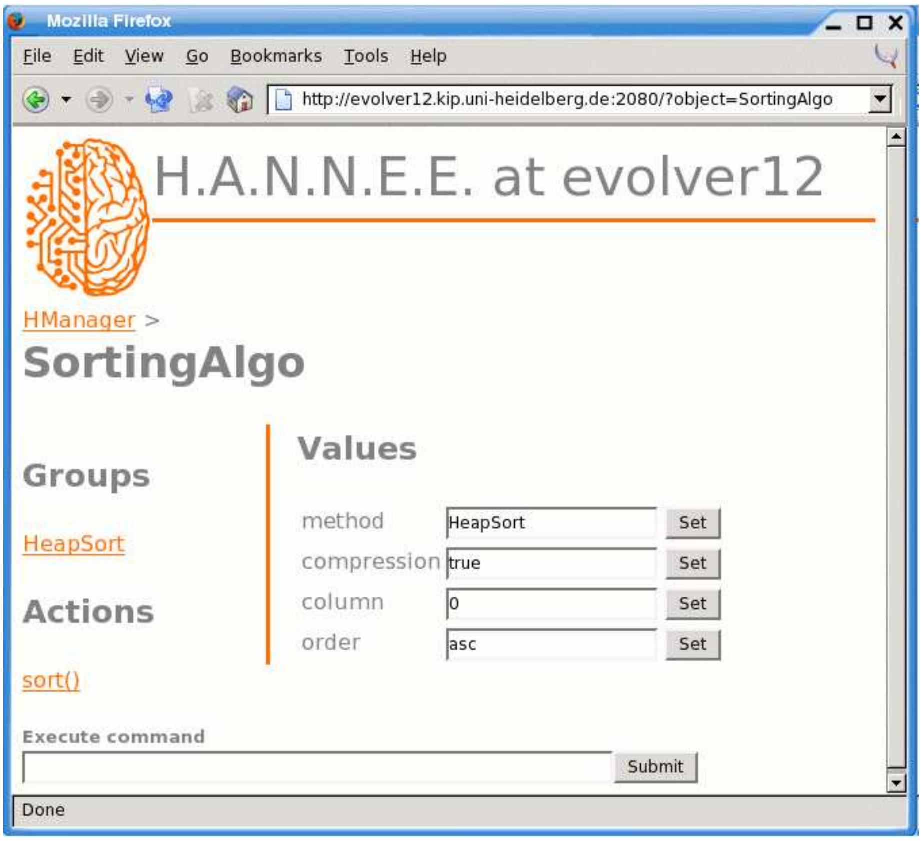


Figure 2.7: *Network interface. The server modules communicate with the HElement library via the control API and the commandline interpreter. On the remote side, standard client software, as present on almost every PC, is used.*

is, from the user's view, effectively the same as is this software was run locally in console mode.

Connecting via the Http protocol is evenly simple. Just direct a web browser to the computer running HANNEE on port 2080. Figure 2.8 shows the web representation of our `SortingAlgo` example.

Besides enabling remote control (which can be essential if the software runs on an embedded system without an own terminal), the network access can be used to create a duplicate user interface. To see why this may be useful, consider the situation in which the regular user interface (console or GUI) is blocked because an active calculation takes much longer than expected. Usually the only chance is to kill the program, losing all the results so far. If, however, network access is enabled, a new user interface is quickly created allowing the user to call a possible interrupt function, or at least to save settings and results to disk before shutting the program down. Each network connection runs in an own thread, so it is not affected by the main thread being blocked.



http://evolver12.kip.uni-heidelberg.de:2080/?object=SortingAlgo

H.A.N.N.E.E. at evolver12

HManager >

SortingAlgo

Groups

HeapSort

Actions

sort()

Values

method	HeapSort
compression	true
column	0
order	asc

Execute command

Figure 2.8: *The Http interface allows to control the software with a standard web browser.*

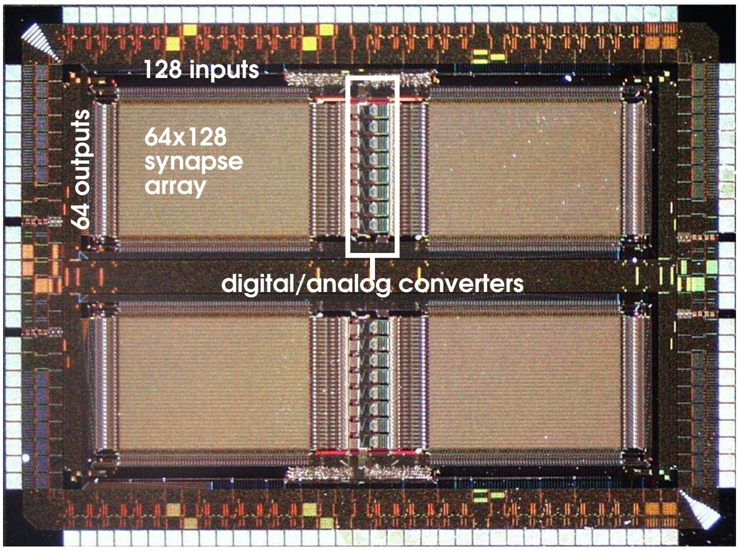


Figure 2.9: *The neural network prototype chip "HAGEN" (micro-photograph). Each of the four synapse arrays constitutes a single-layer perceptron fully connecting 128 inputs to 64 outputs. Neural computations are carried out by analog electronics.*

2.2 Hardware

2.2.1 The HAGEN Chip

The neuro chip used in this thesis is a prototype implementation of a general array-based analog VLSI[5] architecture [54]. The chip, named by the acronym HAGEN (Heidelberg AnaloG Evolvable Neural network), is divided into four synapse arrays, each containing 64 binary threshold neurons, computing their output according to

$$O = \theta\left(\sum_i w_i I_i\right) \quad \text{(cf. eq. 1.11)} \tag{2.1}$$

where $\theta(\cdot)$ is the step function. A neuron threshold t, as present in equation 1.11, is not explicitely defined, but it can be be realized by using one of the regular inputs as a constant bias. The neurons of one array share the same 128 binary inputs, forming a fully connected single-layer Perceptron. The 8,192 synapses of each array are laid out in a 64 x 128 array (see Figures 2.9 and 2.10, as well as Figure 1.16 on p. 27 for more detail).

The neural computations are carried out by analog electronics. The idea is to take advantage of simple laws of physics, e.g., the Kirchhoff laws for currents, to perform the calculations, rather than employing the complex machinery of a digital processor. That is why analog circuits can be designed to require less chip area compared to digital solutions doing the same calculations. For the HAGEN chip this pays off in a high synapse density. Moreover, analog im-

[5] Very Large-Scale Integration

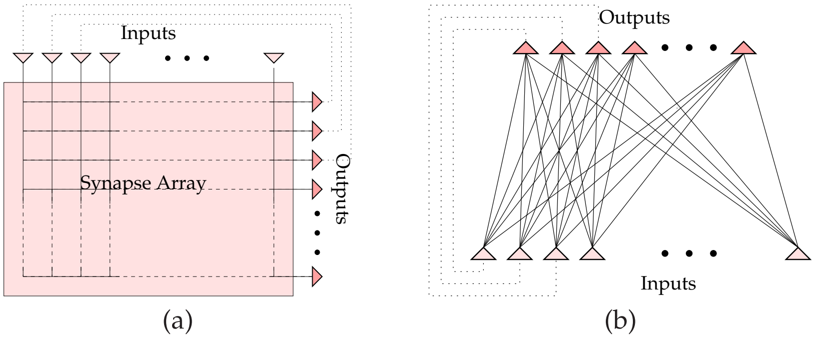


Figure 2.10: *One synapse array is equivalent to a fully connected single-layer network. a) Hardware schematic. b) Equivalent "usual" abstract network representation. The hardware system features on-chip and off-chip configurable feedback connections (dotted lines) which allow for time-discrete recurrent operation. In this thesis however, only feed-forward networks are considered.*

plementations are usually more economic in terms of power dissipation (see also section 1.4).

As a major drawback of analog computing, the results are usually not exact due to device variations and statistical noise influencing the computation results in unpredictable ways. In order to avoid the accumulation of errors, the analog computations are confined within the synapse arrays. Input and output connections are interfaced digitally, and the analog synaptic weights are generated by digital-to-analog converters on the chip. Thus, all communication outside the synapse array is digital, ensuring data integrity while still exploiting the advantage of fast and highly integrated analog computing units. The analog circuits in the computing arrays

In order to make the architecture scalable, the network operates at a constant frequency, with the outputs of all neurons being updated synchronously once each network cycle. For setting up multi-layered or recurrent networks, some of the outputs can be fed back to inputs of the same or of a different array via physical on-chip, or virtual off-chip, configurable feedback lines. In such synchronously updated networks, a feedback connection introduces a time delay of multiples of a network cycle. All hardware feedbacks within the chip have a delay of one cycle. External inter-chip connections can be set up with delays of arbitrary multiples of one cycle.

Operation Principle

Figure 2.11 illustrates the operation principle of the analog neuron computations. The figure displays the block diagram of one neuron, corresponding to one row of a synapse array. The main idea is that the inhibitory and excitatory activations are summed as currents in the two electric lines I^{pos} and I^{neg}. A comparison of the two signals in the neuron body circuit determines the output state of the neuron.

The synapses act as current sinks, connected either to the I^{pos} or the I^{neg} line, depending on the sign of the synaptic weights. If the input signal is off, the sink is connected to a third line, I^{park} to keep the synapse circuit at its

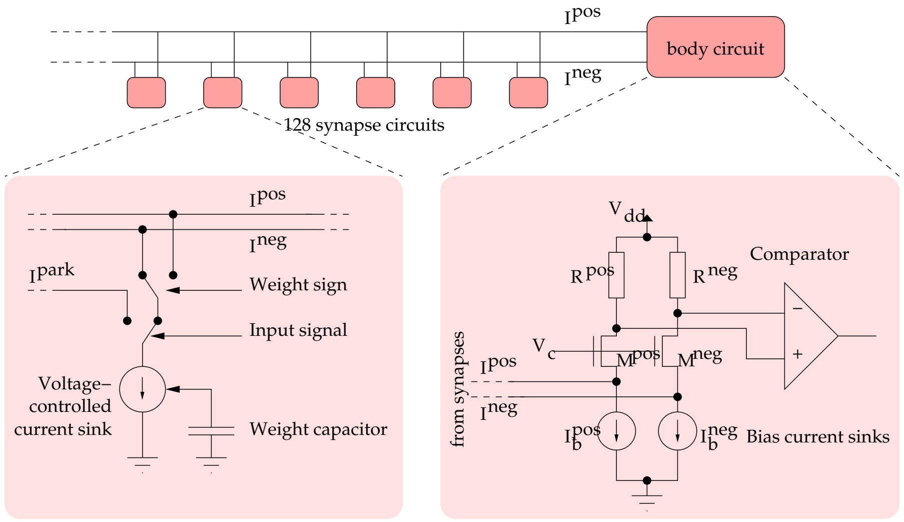


Figure 2.11: *Operation principle of the the analog neuron computations in the HAGEN chip.* *(After [54])*

operating point. The current sink is implemented as a current memory. The controlling capacitor is charged during the programming phase such that the a current proportional to the modulus of the weight value is sustained.

In the neuron body circuit, the resistors R^{pos} and R^{neg} convert the inhibitory and excitatory current into voltages which are compared in order to determine wether the neuron should fire or not. Two equal bias currents I_b keep the body circuit at its working point even if no input synapse is active. The cascode transistors M decouple the comparator from the large capacitances of the current lines and thus enables faster operation.

2.2.2 Distributed Operation of Multiple Chips

HAGEN's modular block-based design and its digital interface make the architecture well scalable. Larger scales can be realized either by building a new chip with more or larger network arrays, or by using several of the existing chips in combination. The latter path is followed by some members of the research group who developed a platform for the distributed operation of large networks using up to 256 neural net chips in parallel [7, 13]. The system both extends the capabilities of the current HAGEN chip and is flexible enough to be used with future chips of various kinds.

The central components of the parallel system are highly integrated network modules (Figure 2.12), each containing all necessary components to operate one neural network chip: programmable logic, a local CPU, memory, and the HAGEN chip itself.

In order to use the network modules in parallel, groups of 16 are connected via a high-speed backplanes. Since the HAGEN network chip accepts digital inputs and produces digital outputs, the data can be transported by digital communication technologies. This way, the outputs of one chip can be fed to another one on a different network module, each for example implementing

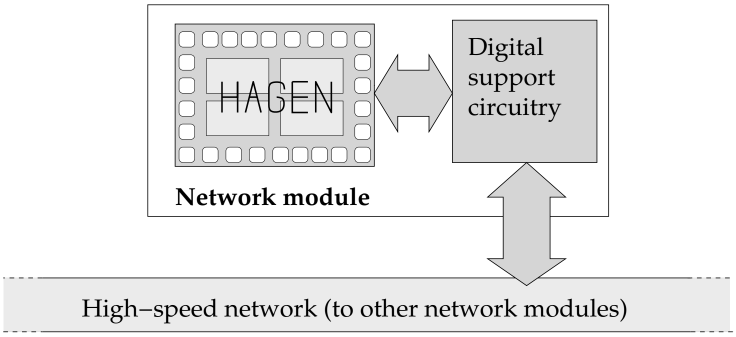

HAGEN
Digital support circuitry
Network module
High–speed network (to other network modules)

Figure 2.12: *Parallel operation of multiple HAGEN chips. Each single chip resides on a compact network module featuring digital support circuity, including memory and a small power-PC processor. A high-speed network carries neural network data and configuration data between up to 16 connected modules.*

one layer of a larger network. The connectivity has been carefully designed in order to maintain a continuous signal stream necessary for the smooth operation of a distributed neural network. Figure 2.13 shows a fully equipped backplane residing in its rack-mounted case. The distributed hardware operation does not play a central role in this thesis. Nevertheless, the possibility of distributing networks over parallel units should be kept in mind when thinking about future applications of the HAGEN architecture.

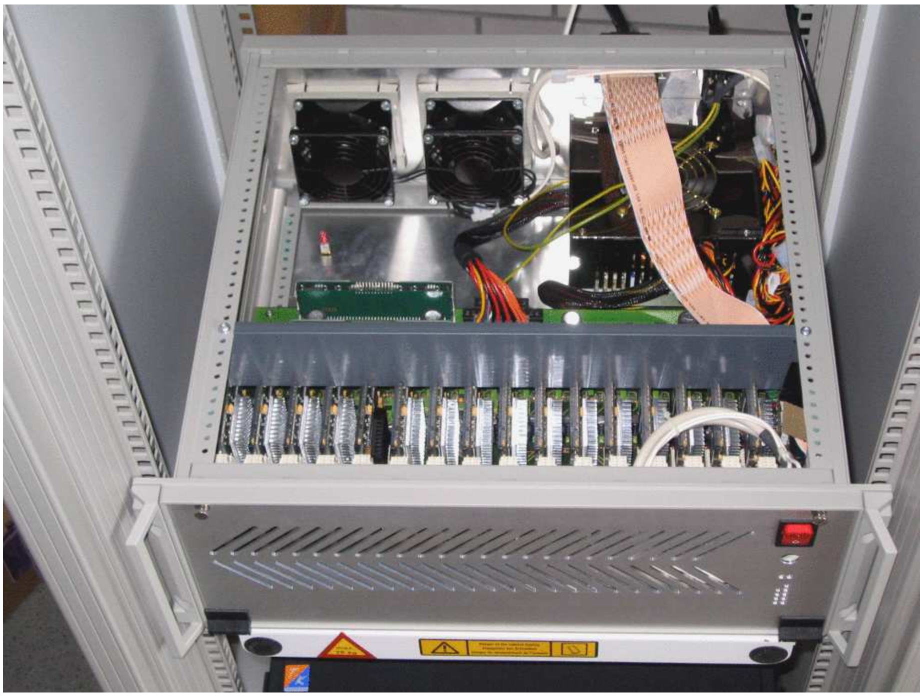


Figure 2.13: *The distributed system with 16 network modules mounted on a backplane in a 19″ rack. For the experiments presented in this thesis, a different system with only a single HAGEN chip was used.*

Chapter 3

A Neural Network for Object Recognition

3.1 Neuron Model

Using a simple threshold neuron model is one of the approachess for realizing the high synapse density, evaluation speed, and low power requirements found in the the HAGEN hardware system (section 2.2.1). Binary signals can be conveniently and reliably transmitted, and the multiply-accumulate operations for the dot product between the weight vector and the binary input data boil down to conditional additions. Moreover, a threshold activation function can be realized by simple, fast, and reliable electronic circuits.

While offering advantages for the hardware implementation, a threshold neuron model does not necessarily imply restrictions for possible applications: In principle, more elaborate, e.g., continuous valued, neuron models can be approximated by ensembles of binary neurons (see [58]). Moreover, in the special case of convolutional neural networks, there is evidence that a simple threshold model might in fact be sufficient for object recognition (cf., section 1.1.5). Nevertheless, a threshold activation function imposes certain restrictions on the applicable training methods. Since a step function lacks a meaningful derivative, well-proven gradient-based methods like back-propagation cannot be applied. Most other supervised and non-supervised automated training methods reported in the field of convolutional networks assume continuous neuron outputs and therefore cannot be applied without heavy modifications as well (cf., section 1.3). Developing a new training method suited for convolutional networks of threshold neurons is one of the main contributions of this thesis.

All neurons in the proposed convolutional network compute their output according to

$$O = \beta\left(\mathbf{w} \cdot \mathbf{I} - t\right), \tag{3.1}$$

where $\mathbf{w} = [w_1, \ldots, w_N]$ and t are the neuron's weights resp. threshold, $\mathbf{I} = [I_1, \ldots, I_N]$ are the current input values $I_i \in \{-1, 1\}$, and $\beta(x)$ is the bipolar step function (1 for $x > 0$, -1 otherwise). The hardware architecture featured by the HAGEN chip uses $\{0, 1\}$ as possible neuron states. This is no contradiction to the bipolar $\{-1, 1\}$ neurons used in the formulas, since by a simple

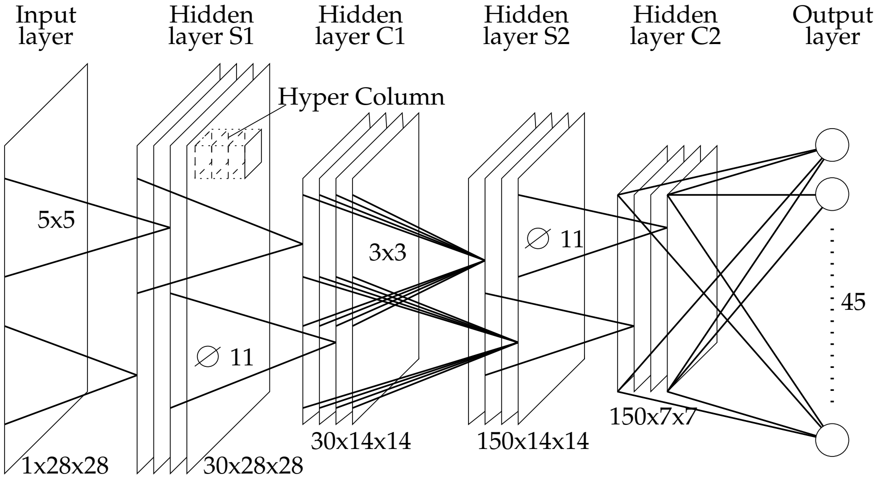


Figure 3.1: *Detailed network topology. The network consists of the input layer, four hidden layers, and one output layer. Layers are locally connected. Shown layer dimensions and convolution kernel sizes refer to the MNIST experiments.*

transformation of the weights and thresholds both neuron types can be converted into each other (see section 6.2.1). Before transferring a trained network onto the hardware, the weights and thresholds are converted accordingly.

3.2 Topology

The network developed in this thesis is a straight-forward implementation of the general scheme of convolutional networks (section 1.2). It consists of the input layer, four hidden layers, and one output layer, as shown in Figure 3.1. Of the four hidden layers, two are recognition layers (or S-type layers), and two are blurring layers (C-type), arranged in alternating order. The alternating feature-extraction and sub-sampling stages carried out by the two layer types enable invariant recognition, as explained in section 1.2.5. Each network layer is divided into a number of equal-size feature planes which are regular grids of neurons. A group of neurons from different planes within the same layer, located at the same grid position, is referred to as a *hyper column*. The neurons in a given hyper column receive input connections from the same local neighborhood of neurons in the preceding layer (which we call their *input region*), but are each tuned to detect a different feature.

All hyper-columns within a given layer are identical with respect to their neurons' synaptic weights (*weight sharing*), while receiving inputs from shifted input regions, depending on their own grid position. In order to compute cells at the layer border, two different strategies are applied: For the handwritten digits (section 4.1.1), parts of the input region falling outside the previous layer's dimensions are padded with the background value (-1). For the traffic sign images 4.1.2, where pseudo line-endings occur at the image border, border neurons in the S-layers are discarded. C-layers still use the the padding technique.

S-layers. The purpose of an S-layer plane is to detect a specific feature. In the current implementation, the input region of an S-layer neuron is chosen to be a square portion of the hyper column grid in the previous layer. The side length of the input region, denoted by d_S, is uniform within a layer, but can generally differ between layers S1 and S2. S-layer neurons connect to to all the planes within their input regions, thus being able to combine features found in preceding stages to find more complex features. The synaptic weights are set by an unsupervised clustering process, driven by training data (details in section 3.3.1).

C-layers. C-layers consist of the same number of feature planes as present in the preceding S-layer. In contrast to S-layers, which are fully connected to their input region, C-neurons receive input from corresponding planes only. All weights of a C-neuron are fixed to 1. Such a uniform convolution kernel results in spatial blurring of the previous S-layer's activation pattern. The threshold is set to a constant value $T_C \geq 1$, being the same for all planes in a layer, but chosen separately for each of the two C-layers. The constant T_C specifies the neural sensitivity. The shape of the C-layer's input regions are circles with diameter d_C.

In addition to blurring, C-layers subsample the spatial resolution of the layer grid by a scaling factor α. This corresponds to transforming concrete, local information to abstract, location-invariant information. Sub-sampling is applied in the following way: First, a virtual full-size C-layer is constructed, and then a fraction of the grid rows and columns is removed. For instance, at a scaling factor of $\alpha = 1/2$, each 2$^{\text{nd}}$ row and column is removed.

Output layer The output layer is fully connected to the last C-layer and is trained in a supervised fashion to distinguish the desired image classes. The coding scheme relating the firing pattern to image classes is chosen differently for the two tested problems (details in section 3.3.2). From the topological point of view, the output layer is just another S-layer with a 1 x 1 hyper column "grid", and an input region size equal to the dimensions of the previous layer. The correspondence between planes (here: single neurons) and learned object classes depends on the chosen coding scheme (see section 3.3.2).

3.3 Training

3.3.1 Hidden Layers: Self-Organization by Clustering

Of the hidden layers, only the S-layers are subject to training. The C-layers have their weights fixed to 1, corresponding to a uniform blurring kernel (see section 3.2). Training proceeds bottom-up, i.e., layer S2 is trained after the training of layer S1 is complete. Assume that an S-layer consisting of K feature planes is to be trained. Because of the weight sharing policy by which all neurons in a plane are equal, only the weights of one prototype hyper column are identified which is duplicated to the full layer dimensions after training.

Given a training image and having evaluated all preceding layers, there is one input vector to each hyper column in the trained layer. (Remember: in S-layers, all neurons within a hyper column receive the same input vector.)

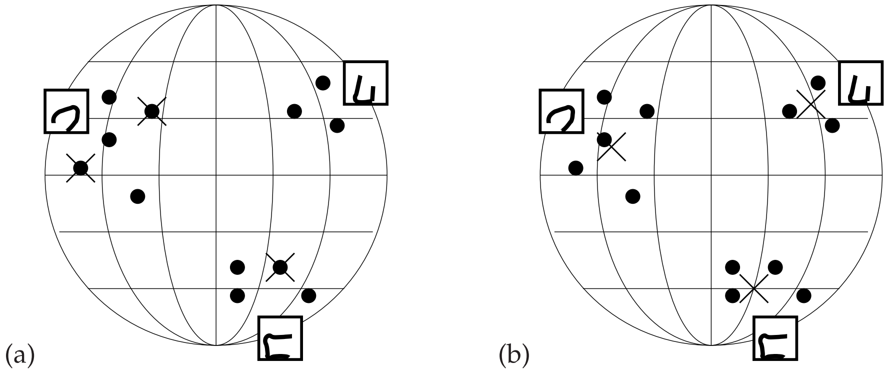


Figure 3.2: *Schematic visualization of the clustering process. Dots represent the input vectors $\mathcal{I}$, located on a hyper sphere, crosses represent the weight vectors w_k. Input vectors are clustered around typical shape features.* **(a)** *Before training, w_k are initialized with random members from $\mathcal{I}$.* **(b)** *After training, w_k have settled in cluster centers.*

The dimension of this vector is the number pf pixels in the input region times the number of feature planes in the preceding layer. We use the notation $\mathbf{I}^j_{x,y}$ for the input vector to the hyper column at grid position (x, y), resulting from training image j. Let $\mathcal{I} := \{\mathbf{I}^j_{x,y}/||\mathbf{I}^j_{x,y}||\}$ be the union set of all (normalized) input vectors for all training images and all positions. These vectors lie on a unit hyper-sphere in the input space, and, since the training images contain the objects to be recognized, the vectors are likely to be clustered around shape features which are typical for these objects and which can be recognized in that layer. Consequently, the proposed training method identifies K clusters in $\mathcal{I}$ (remember, K is the number of feature planes) and the cluster centers are used as the weight vectors of the K neurons in the prototype hyper column. Figure 3.2 illustrates the idea. This consideration does not depend on certain shape features to appear always at the same positions in the training images, since $\mathcal{I}$ includes data from all grid positions. Input vectors where all components equal -1 are not considered for clustering, since they are not assumed to contain meaningful features.

A variation of the K-Means algorithm is used for clustering (see e.g. [24] or [39]). The original version of this algorithm assumes that data points are distributed in space like the sum of K Gaussians, where the positions of the Gaussians are unknown. The positions of the Gaussians are also called "cluster centers" in this context. The K-Means algorithm identifies the cluster centers maximally consistent with the given data set. The data we are interested in lies on the surface of a hyper-sphere, so we choose the angle between two vectors as the distance measure (abandoning the straight K-Means which uses Euclidian distance). In detail, the procedure is as follows: At the start, the cluster centers $\mathbf{w}_k$, $k = 1 \dots K$, are initialized with vectors randomly drawn from $\mathcal{I}$. Then, repeatedly, the following two steps are performed in each training epoch:

1. For each vector $\mathbf{I} \in \mathcal{I}$, assign $\mathbf{I}$ to the cluster $\tilde{k}$ the center of which has the smallest angle to $\mathbf{I}$ ($\tilde{k} = \text{argmax}_{k=1}^{K}(\mathbf{I} \cdot \mathbf{w}_k)$).

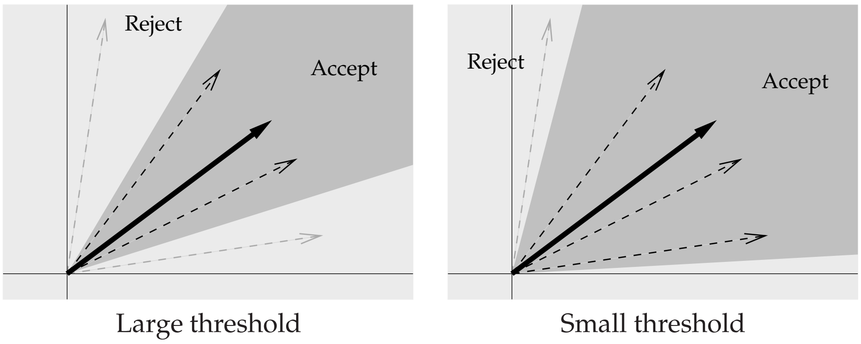


Figure 3.3: *A neuron's threshold determines its selectivity. Bold arrows depict the optimal stimulus for one sample neuron in the input space. Dashed arrows represent candidate input vectors. Only input vectors within the acceptance region make the neuron fire.*

2. Update each cluster center $\mathbf{w}_k$ to be the center of mass of all vectors being assigned to the kth cluster. Re-normalize $\mathbf{w}_k$ to unit length.

This variation of the K-Means algorithm is equivalent to competitive Hebbian learning first described by Rumelhart 1985 [50] (see also section 1.3.3).

The algorithm stops if either only a small fraction (0.5%) of patterns had their cluster assignments altered in the previous epoch or a maximum of epochs (100) has elapsed. Experience collected during the experiments suggests that the exact definition of the termination criterion is not very critical.

Interestingly, the clustering process requires only a comparably small number of training images. An evaluation of the performance of the trained networks using varying training set sizes is included in section 4.2.3.

Neuron Thresholds

The described training method identifies the weight vectors for the S-neurons, but does not specify their thresholds. In other words, the optimal input vector is given but the maximum allowed deviation from this vector still activating the neuron remains to be specified (cf., Figure 3.3). Choosing high selectivities (large thresholds) may prevent the network from responding to objects which differ slightly from the learned prototype (bad generalization), choosing the thresholds too low will decrease the network's discrimination power. Finding the thresholds for which the network performs optimally is a non-trivial task. Extensive methods to adjust the thresholds in a similar convolutional network were studied previously [35].

In this thesis, the neuron threshold t for S-neurons is set to a fraction of the respective neuron's maximally achievable activation:

$$t = T_S \sum_i |w_i|. \tag{3.2}$$

The meta parameter T_S equal for all planes within a given layer. It is defined separately for each of the two S-layers. T_S does not need to be specified with excessive accuracy in order to achieve good results (cf., Table 4.1).

3.3.2 Output Layer: Supervised Perceptron Learning

The output layer is a set of linear classifiers, trained in a supervised manner where for each training image the desired output of every neuron is given. Two different coding schemes for mapping the image class to an activation pattern are used in this thesis.

Pairwise classifiers. Each output neuron is, trained to discriminate only two classes. For n pattern classes, and considering every possible combination of two classes, there are $(n^2 - n)/2$ output units. When evaluating an unseen pattern, each unit votes for one of the two classes it was trained with. The class receiving the most overall votes wins. This voting scheme is applied for the digits network (section 4.1.1).

Ensemble voting. For each of the n image classes, m neurons are trained to distinguish this class from all the other classes, making an overall number of $n \cdot m$ neurons. Due to the stochastic nature of the training, the m neurons of the same class will develop sightly differently. In the recognition phase, each neuron votes for the class is was trained with, either with 1 or with 0. The class receiving the most overall votes wins. This voting scheme is applied for the traffic sign network network (section 4.1.2).

In both coding schemes, whenever two or more classes receive an equal amount of votes the respective image is counted as misclassified.

Training is done using the well-known Perceptron learning rule, where after each pattern presentation, a neuron's weights and threshold $\mathbf{v} = [w_1, \ldots, w_N, t]$ are updated according to:

$$\mathbf{v} \leftarrow \begin{cases} \mathbf{v} - O\mathbf{J}, & \text{if } O \text{ is incorrect} \\ \mathbf{v}, & \text{if } O \text{ is correct} \end{cases}, \tag{3.3}$$

where $\mathbf{J} = [I_1, \ldots, I_N, -1]$ is the current input vector plus an additional constant component to account for the bias t in $\mathbf{v}$, and O is the neuron's current output. Here, the weight vector dimension N equals the total number of neurons in the last hidden layer (C2). The training patterns are presented in random order and the procedure is terminated if the output is correct for a predefined number of consecutive pattern presentations, or if a preset number of iterations has passed.

Note that no explicit normalization step is involved in the training. The weights and t are allowed to grow without constraint. However, as the *length* of $\mathbf{v}$ in (3.3) increases, its change in *direction* will tend to decrease since vectors $\mathbf{J}$ of fixed length $\sqrt{N+1}$ are added in each iteration. Only after the entire training process, $\mathbf{v}$ is re-normalized to unit maximum norm since the hardware implementation requires weights bounded to $[-1, 1]$.

The Perceptron rule is guaranteed to converge if the data to be learned is linearly separable. For the large networks used in the software simulations, convergence is always observed. When working with restricted network sizes on the hardware, techniques are applied for dealing with oscillating solutions (section 6.5.1, p. 107).

3.4 Image Preprocessing

Convolutional networks operate directly on raw pixel data. Elaborate preprocessing methods are not necessary (only simple operations like scaling and color normalization are commonly applied). However, the threshold network used in this thesis requires binary input, so, grey scale images must be converted into a binary representation before they can be processed. Two different methods are applied:

Threshold segmentation In the experiments with hand-written digits good results are obtained by simply applying a threshold to the original grey scale images at half the maximum grey value. Pixels with grey value greater than 128 are regarded as black (binary value 1), pixels below this level are assigned white (value -1). Additional experiments with varying threshold levels are reported in section 4.3.1.

Hard edge detection For other pictures, such as the investigated photographs of traffic signs, an obvious background-object segmentation is not possible. Additionally, the photos exhibit varying illumination conditions. In this case, the images are first normalized to the full grey value range [0..255]. Then they are processed by a 5 x 5 Gaussian smoothing kernel and, subsequently, a 3 x 3 Laplacian filter. This Laplacian-of-Gaussian technique works as a contrast-based edge detector [23]. The result of the Laplace filter is further thresholded in order to obtain binary images. The same threshold value of 5 was used for all pictures in the described experiments.

3.5 Meta Parameters

In the above description of the network topology and the training methods, several values remain unspecified. These are meta parameters defining the details for a concrete network implementation. The meta parameters needed to specify one adjacent S/C layer pair are summarized in Table 3.1.

Meta parameters are adjusted by a meta training process. This usually involves performing several training runs with different meta parameter settings, and choosing the meta parameters which yield the best results on the test data. Depending on the nature of the meta parameters, a separate *validation set* might be required for optimizing the meta parameters in addition to the *test set* which is then only used to assess the classification performance of the final system. When doing the meta training directly on the test set, one can run into the problem that by the meta optimization, the system is tuned to perform well on the test set, but will fail on new, unseen, data. In the reported experiments, a validation set was used for the first benchmark problem (hand-written digits), but in the second benchmark (traffic signs), meta optimization was done on the test set due to the lack of sufficient training data.

Several systematic meta training techniques were developed. Eventually, the last method described below (which is manual adjustment) yielded the networks reported in this thesis.

K	The number of feature planes per layer (the same for both the S and C layer)
d_S	For the S-layers, the input region is a square of hyper-columns of size $d_S \times d_S$
d_C	C-neurons have a circular input region with diameter d_C
T_S	The threshold parameter defining the threshold for S-neurons (see eq. (3.2))
T_C	The threshold of C-layer neurons
α	The scaling factor for sub-sampling the hyper column grid in C-layers ($0 < \alpha \geq 1$)

Table 3.1: *Meta parameters for one hidden S/C layer pair.*

Systematic parameter sweeps. The most straight-forward and reliable method to find the optimum parameter settings is a systematic sweep. The parameter space is discretized into a grid and for each grid point, the system is trained and tested. In order to statistically judge the differences between different parameter settings, several training-and-validate runs are conducted for each grid point. This method yields a comprehensive picture of the parameter space, and is easy to implement [20].

However, for this thesis, systematic parameter sweeps are not practical: If only 3 discrete values of each of the 16 meta parameters (8 per S/C layer pair) were to be tested, and for each grid point, 3 training runs were conducted, covering all possible parameter combinations would take approximately 15,000 years of computing time (3×3^{16} hours), given that one training run takes roughly an hour.

On the other hand, for a small subset of the parameters, this technique is reasonably applicable. In Figure 4.11, results are shown for varying the number of feature panes in the two S-layers.

Evolutionary optimization. Introductions to evolutionary and genetic algorithms are given in the book [39] or the thesis [22]. For the particular parameter search needed here, the genetic approach is applied as follows: For each meta parameter $P_I, I = 1, ..., Q$, a set of possible values $\mathcal{P}_I$ is defined by heuristic reasoning and previous experiences (typically $2 \leq ||\mathcal{P}_I|| \leq 10$). Then, a randomly initialized population of 10 individuals is evolved using single gene mutation, no crossover, and 1-individual elitism. The fitness of one individual is given by the average validation error of three independent training-and-validate runs. This approach is consistently observed to converge at similar parameter sets after a reasonable number (50-100) of generations. However, since the evaluation of the networks in software takes a long time (the MNIST data set consists of 70,000 images), a few days of computing time are necessary for meta training.

Cyclic optimization along parameter axes. This approach is based on a simple iterative method found in [46]. Like in the evolutionary approach, a set of possible values is defined for each meta parameter. Then, cyclically, every parameter is optimized in turn:

```
1) For $J = 1$ to $Q$
    • initialize $P_J$ with a random element from $\mathcal{P}_J$.
2) For $J = 1$ to $Q$
    • Run weight training and validation for all $P_J \in \mathcal{P}_J$ while keeping the other $P_I,\ I \neq J$ fixed.
    • Replace $P_J$ by that value $\in \mathcal{P}_J$ which yiels the best validation result.
3) Go to 2)
```

Manual adjustments layer by layer. In this approach, the two layer pairs S1/C1 and S2/C2 are optimized separately by hand. The idea is to adjust each layer pair such that, after weight training according to section 3.3.1, it transforms the input data into a representation more easy to classify than the input data itself. The error of a simple linear classifier was used as a measure of how easy data is classifiable.

First, a temporary 10-class linear classifier directly connected to the input layer is trained, yielding a (rather large) classification error E_0. Then, the same linear classifier is connected to the layer C1 and the meta parameters in layers S1 and C1 are manually adjusted such that, after weight training of these layers, the classifier yields a good performance (classification error E_1). Thereby, comparing the classification errors E_1 and E_0 is a convenient hint for judging whether layers S1 and C1 really transform the image data into a more abstract representation. After this, the same procedure is repeated with the temporary classifier connected to layer C2 and varying the meta parameters of layers S2 and C2 while keeping those of layers S1 and C1 fixed.

This manual method was in fact the one yielding the network settings reported in the experimental chapters. The reason is not that the other methods were deemed inferior, but that external factors demanded a quick solution which prohibited the use of one of the computationally time-consuming methods above. Also, for verifying the benefits of the developed training method, strictly optimal meta parameters are not necessary.

For the two considered test problems meta settings were found which decrease the linear classifier's error in the first and second S/C layer pair ($E_0 > E_1 > E_2$). In the time spent it was not possible to find settings for an additional, third, layer pair (layers S3 and C3) to further simplify the data. All tested settings yielded $E_3 \geq E_2$. For this reason it was obstained from using networks with more than four hidden layers.

Chapter 4

Results With Ideal Neurons

4.1 Two Benchmark Problems

The methods developed in chapter 3 are verified on two test recognition problems. One is based on a publicly available data base of hand-written digits, the other involves classifying photographs of traffic signs taken by the author. The images are available on request.

All the experiments in the two following chapters are based on software-simulations. This way, the properties of the developed algorithms can be tested in a controlled setup, without the additional particularities of the hardware system.

4.1.1 Hand-Written Digits

The recognition of hand-written digits is one of the most-reported applications of convolutional networks [11, 41, 31, 37]. In order to provide a means of comparing the algorithms developed in this thesis with prior art, the publicly available MNIST data base of hand-written digits [40] is used as the main benchmark. It contains 70,000 28 x 28 pixel grey-value images of the hand-written digits “0” through “9” from hundreds of different writers. Samples are shown in Figure 4.1. There exists a standard partition into training and test sets, provisioning 60,000 images for training and 10,000 for testing.

Classification performances of various machine learning approaches are known for this data set. Figure 4.2 lists some of the reported test errors. The test error specifies the percentage of test images wrongly classified after the system was trained with the training set. More “high scores” can be found on the web site `http://yann.lecun.com/exdb/mnist` [40]. Unfortunately, most publications do not provide information about the uncertainty in the measured recognition rates. E.g., it is unclear whether the authors report average recognition rates or just the result of only one single training run. A comment in the review paper [32] suggests that the uncertainty of the values in Figure 4.2 is about 0.1 percent points.

For use in this thesis, the images are converted from grey scale to binary by applying a threshold at half the maximum pixel value (cf., section 3.4). For finding the optimal meta parameters of the used network, the 60,000 training

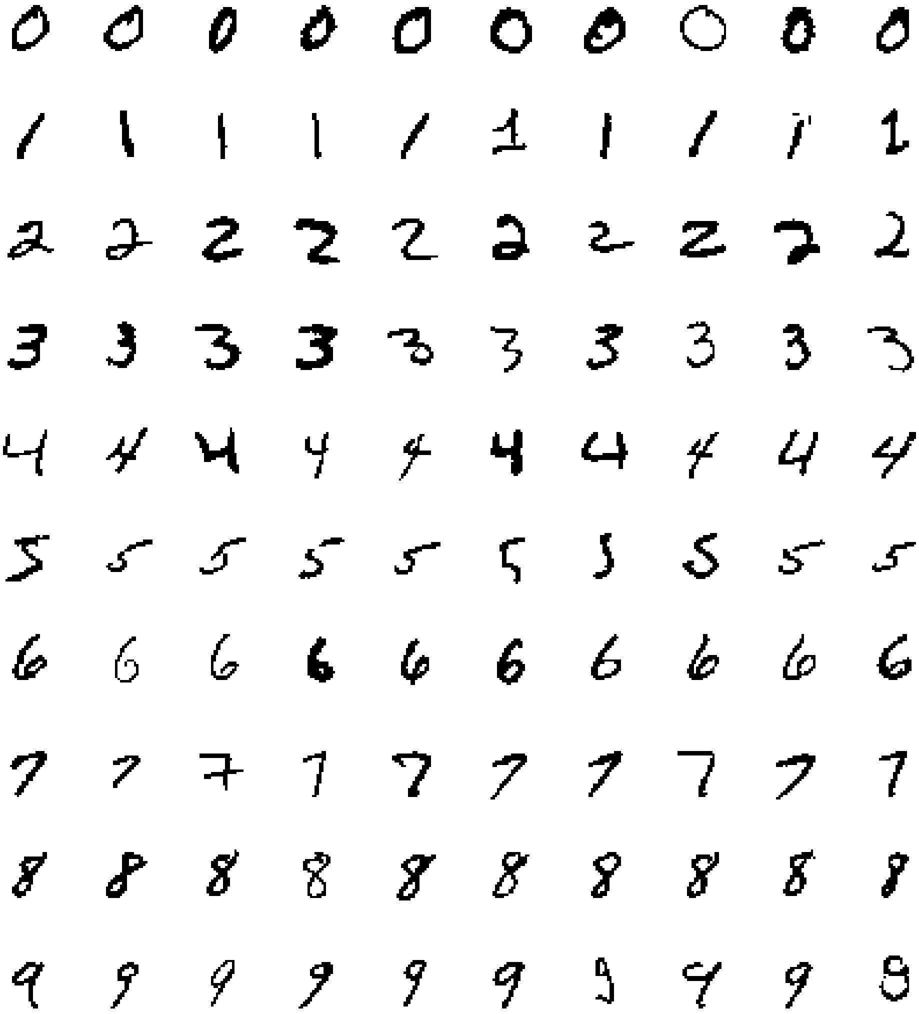


Figure 4.1: *Sample images from the MNIST data base of hand-written digits (threshold-binarized)*

images are split further into a training set (50,000) and a validation set (10,000), each with equal distribution of digit classes. With the meta parameters found to perform best on the validation set (Table 4.1), 100 networks are trained with the patterns of the training and validation set combined. The output layer consists of 45 neurons, corresponding to a 10-class pairwise linear classifier described in section 3.3.2.

Due to the size of the MNIST data set and to save computing time, only a subset of the training images (200 per class) is considered for training the hidden layers. Taking more training samples does not improve the results (cf., section 4.2.3). For training the output layer, however, all the 60,000 training samples are used.

The average error rate obtained on the test set is 1.74%±0.10% over the 100 training runs (best network: 1.49%, worst network: 1.97%). Here, the error is

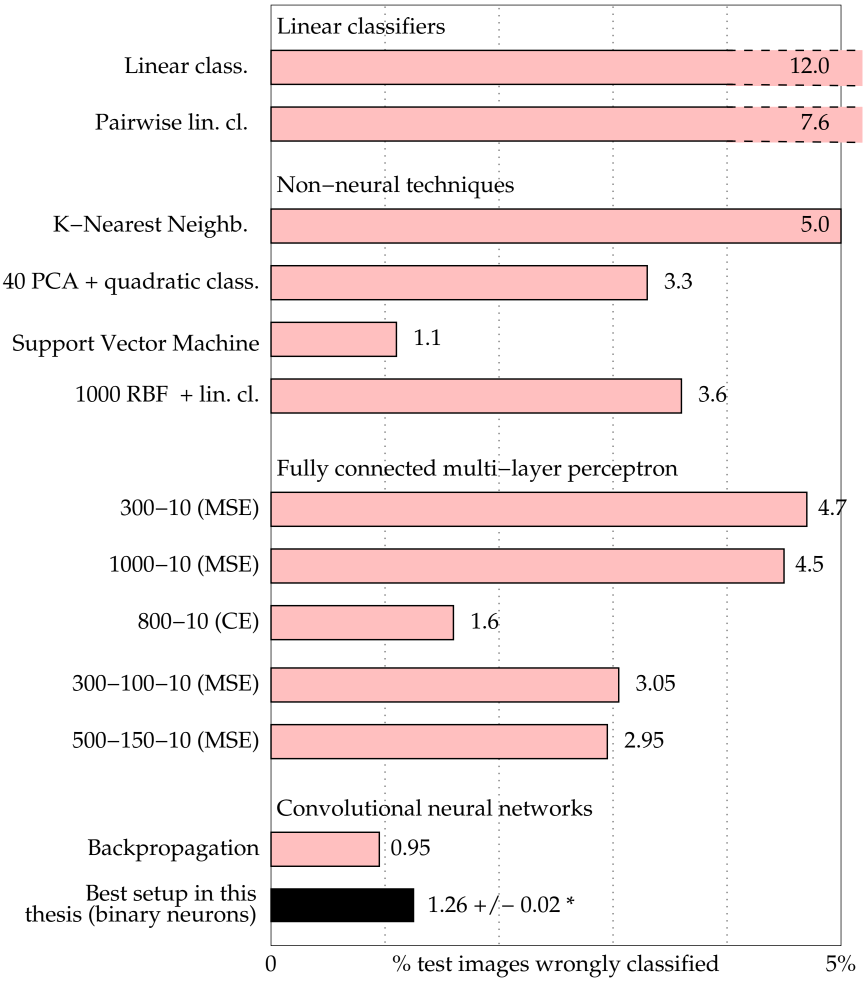


Figure 4.2: *Reported test errors for the MNIST data base. Only methods are considered which do not include sophisticated image preprocessing and which do not use artificial data set expansion. RBF=Radial basis function network, PCA=Principal component analysis, MSE=square error cost function, CE=cross-entropy cost function. Values (except the last) are taken from a review publication [32].* * *see Section 4.3.3*

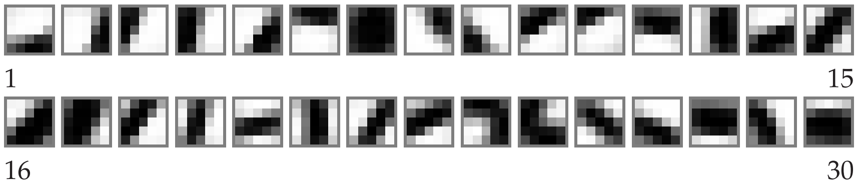


Figure 4.3: *Synaptic weights of the 30 neurons in a typical S1 layer after self-organization (MNIST experiment). The 25 weights of each neuron are shown in a 5 x 5 grid according to their spatial arrangement. Grey-scale coding: black=1, white=-1.*

given as the standard deviation within the ensemble.

In order to give an impression of how the applied training method yields a hierarchical set of feature detectors, examples of detected features are shown in Figure 4.3. The spatial distributions of the synaptic weights in the first network layer (S1) of an arbitrary network are presented (other networks look similar). The 5 x 5 weights of the 30 neurons are visualized in a grey-scale coded scheme, (white=negative, black=positive). Obviously, oriented edges and lines are the preferred stimuli in this network layer.

In the second recognition stage (layer S2), the plain weight values are inconvenient to interpret. Instead, effective receptive field stimuli are shown for selected neurons, i.e., image patches which cause strong activation. For this, the internal activation ($\mathbf{w} \cdot \mathbf{I}$ in equation 3.1) of each neuron is recorded for 1,000 images. In Figure 4.4, for 30 selected neurons the five receptive field stimuli are shown which evoke the strongest activations throughout the considered data set. The network is the same as in Figure 4.3. Apparently, features characteristic for digits (line-endings, loops, and other specific stroke segments) are recognized in layer S2. Note that until this point, no explicit knowledge about the classification task has entered the system. The investigations suggest that the proposed method of feature extraction in the intermediate layers can be very differentiated, such that—in the ideal case—the output layer merely has to choose from the features presented to it.

In Figure 4.5, for one selected training run all misclassified images from the test set are shown.

	K	D_S	T_S	D_C	T_C	α
MNIST						
Layers 1-2	30	5	0.5	7	1	0.5
Layers 3-4	150	3	0.4	7	0	0.5
Traffic signs						
Layers 1-2	25	5	0.7	11	1	0.33
Layers 3-4	100	3	0.6	11	3	0.33

Table 4.1: *Meta parameter settings used in the experiments (cf., Table 3.1)*

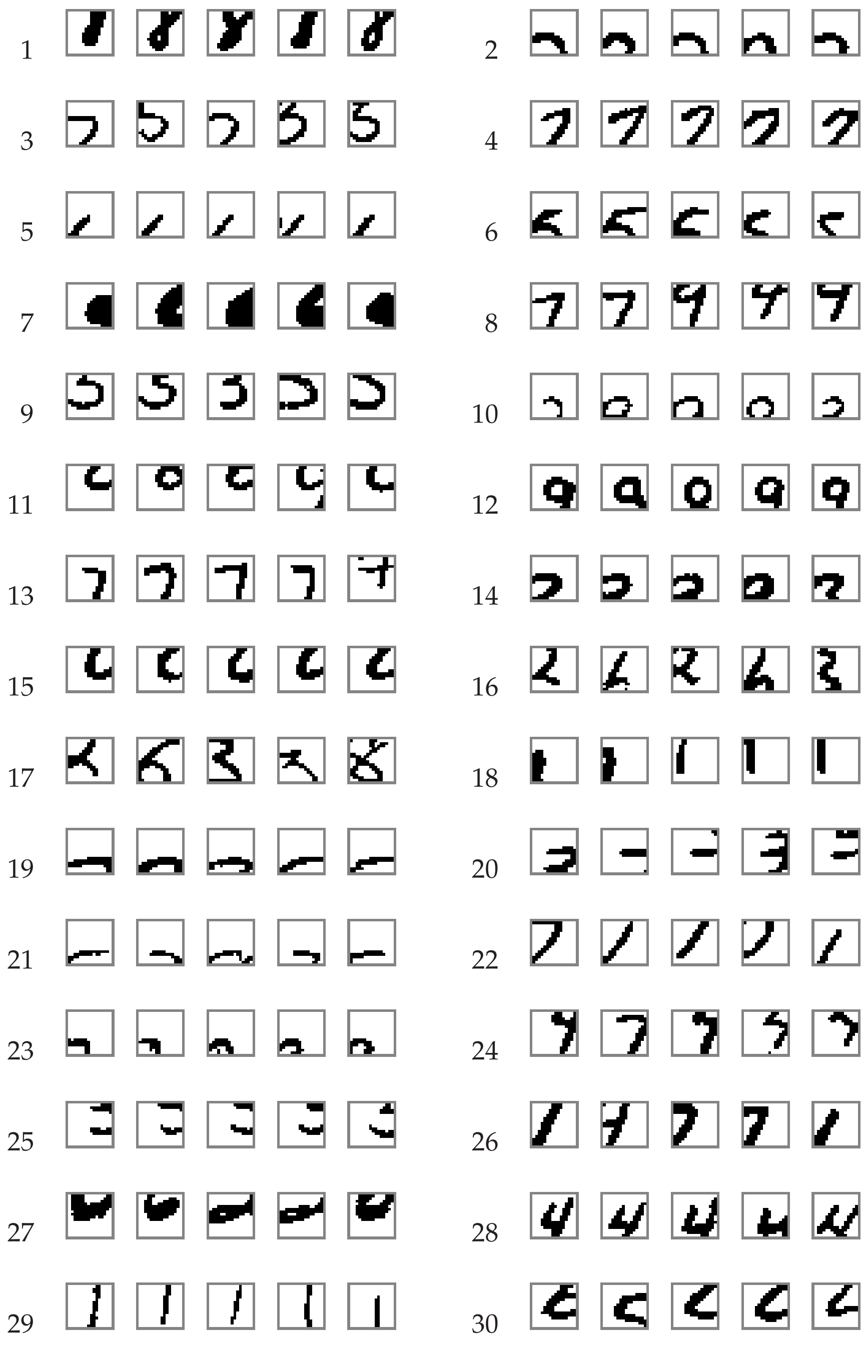


Figure 4.4: *Detected features in layer S2 after self-organization (MNIST experiment) for 30 selected neurons (of 150 total), five strongly activating receptive field stimuli are shown. Neurons taken from the same network as in Figure 4.3*

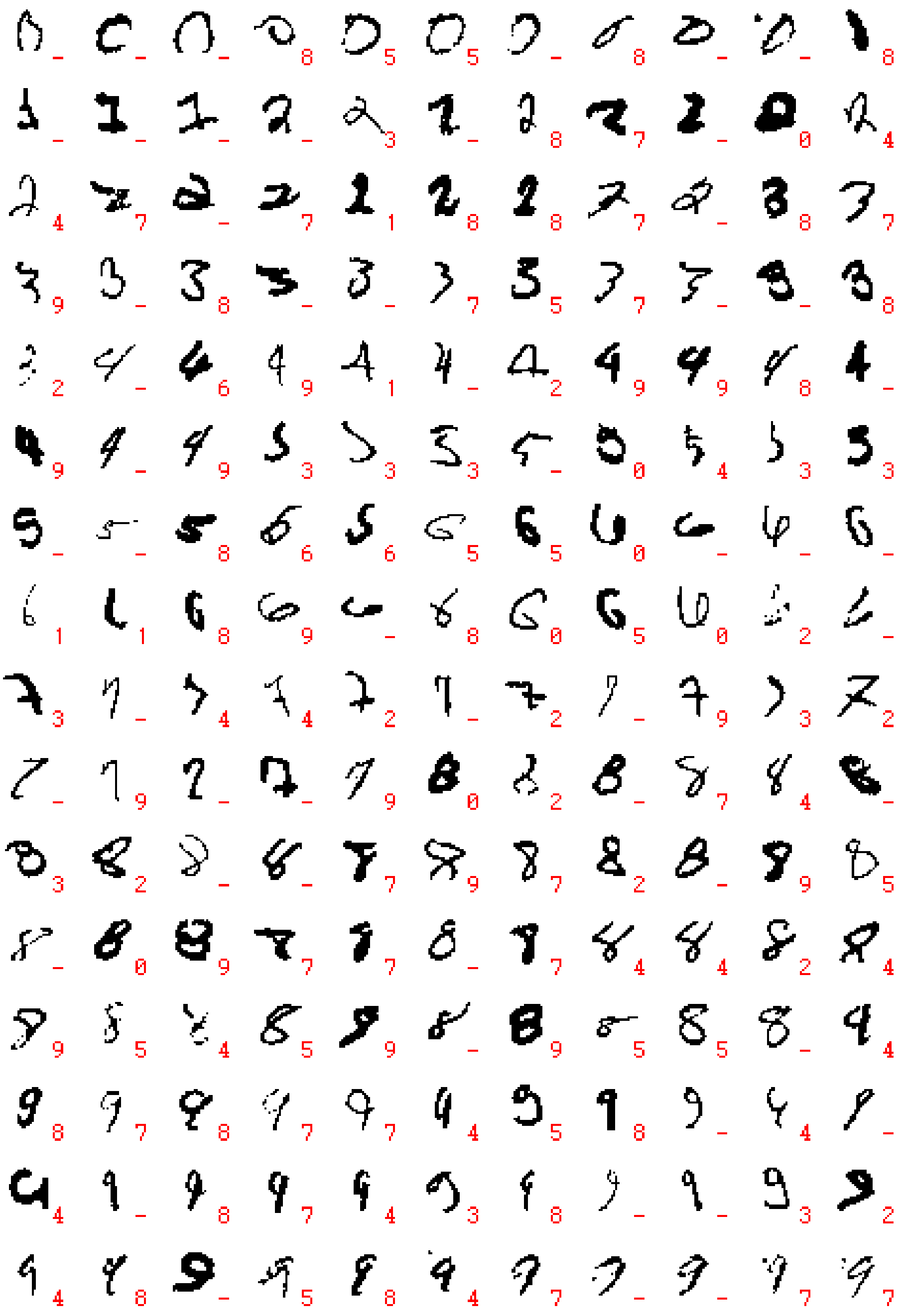


Figure 4.5: *All 176 (out of 10,000) misclassified test images for one selected training run. Small numbers show the class predicted by the neural network. A dash (-) indicates that two or more classes received an equal voting count.*

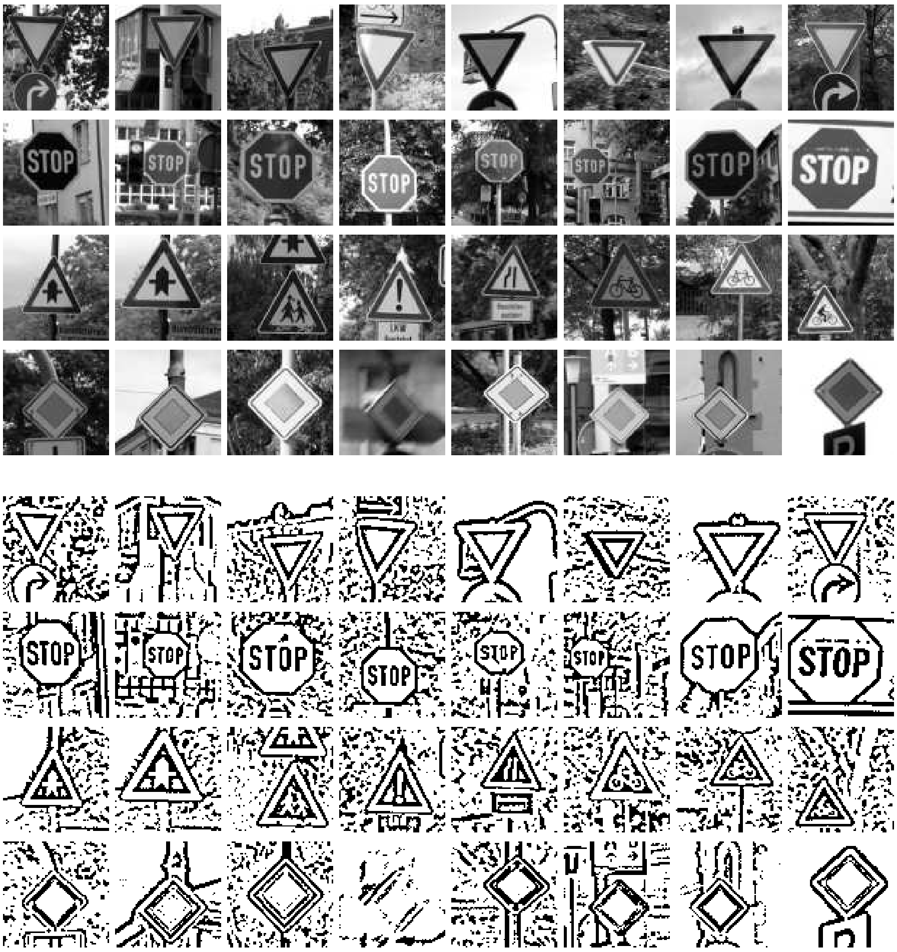


Figure 4.6: *Sample images from the data base of traffic sign photographs. Top: original grey scale images. Bottom: after pre-processing (see section 3.4).*

4.1.2 Traffic Signs

The hand-written digits exhibit a high degree of variations in shape and stroke width, but the digits are all roughly the same size and are centered in the image. Also, a clear discrimination between background and object is given. In order to illustrate that the training method works invariant of position and scale, and also in the presence of noise, the convolutional network approach is applied to photographs of traffic signs. 400 pictures of 4 classes of traffic signs (100 per class) were taken and cropped to 75 x 75 pixels dimension, while making sure that the images still show variances in shift and scale.

The images are pre-processed by an edge detector as described in section 3.4. The preprocessed images are 73 x 73 pixels in size. Figure 4.6 shows examples of the original and preprocessed images. The data set is split into two parts (300/100) serving as training and test data sets, respectively. Splitting was done randomly with the constraint that both sets contain equal class distributions. The network meta parameters are shown in Table 4.1, bottom two

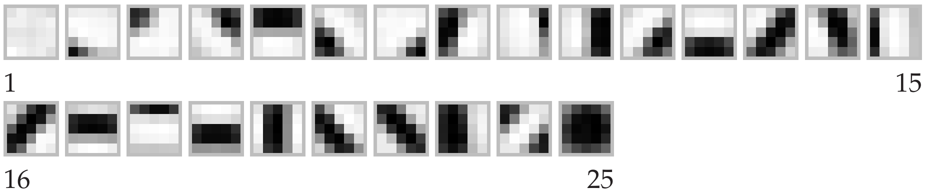


Figure 4.7: *Synaptic weights of the 25 neurons in a typical S1 layer after self-organization (traffic sign experiment). The 25 weights of each neuron are shown in a 5 x 5 grid according to their spatial arrangement. Grey-scale coding: black=1, white=-1.*

rows. Parameter optimization was done on the test set. Although it is understood that this is not 100% clean, introducing an extra validation set would have decreased the already small training set even further.

Here, the pairwise linear classifier used for the MNIST images does not produce optimal results, probably due to the smaller number of image classes. Instead, a simple ensemble voting is employed where each output unit is trained to separate one given class from all the others (cf. section 3.3.2). Per class, ten units are trained, making an overall number of $4 \times 10 = 40$ output units. On the test set, voting among these units determines the answer of the network. If two or more classes receive an equal number of votes, the image is supposed to be misclassified. Training and evaluation of the network is conducted 20 times. The average, best, and worst achieved classification errors are 2.2%, 0.0%, and 4.0%, respectively. The uncertainty in a single measurement is 1.0%.

The features detected in the S-layers as a result of the unsupervised learning procedure are visualized in Figures 4.7 and 4.8 in the same manner as for the MNIST experiment. The same network, arbitrarily selected from the 20 conducted runs, is shown in both figures. Notable is the obvious similarity between Figure 4.7 and Figure 4.3, since they result from different training input. Lines and edges of various orientations seem to be universal low-level image features.

Figure 4.9 displays the images which were misclassified in at least one of the 20 training runs.

	MNIST Digits	**Traffic Signs**
# Training images	60,000	300
# Test images	10,000	100
# Training Runs	100	20
Avg. Test Error [%]	**1.74±0.10**	**2.2±1.0**
Best Network	1.49	0
Worst Network	1.97	4.0

Table 4.2: *Summary of benchmark tests. Shown results for MNIST are not the best ones obtained. They merely serve as a basis of reference for further evaluations. Given deviations of avg. test errors correspond to the uncertainty of a single measurement.*

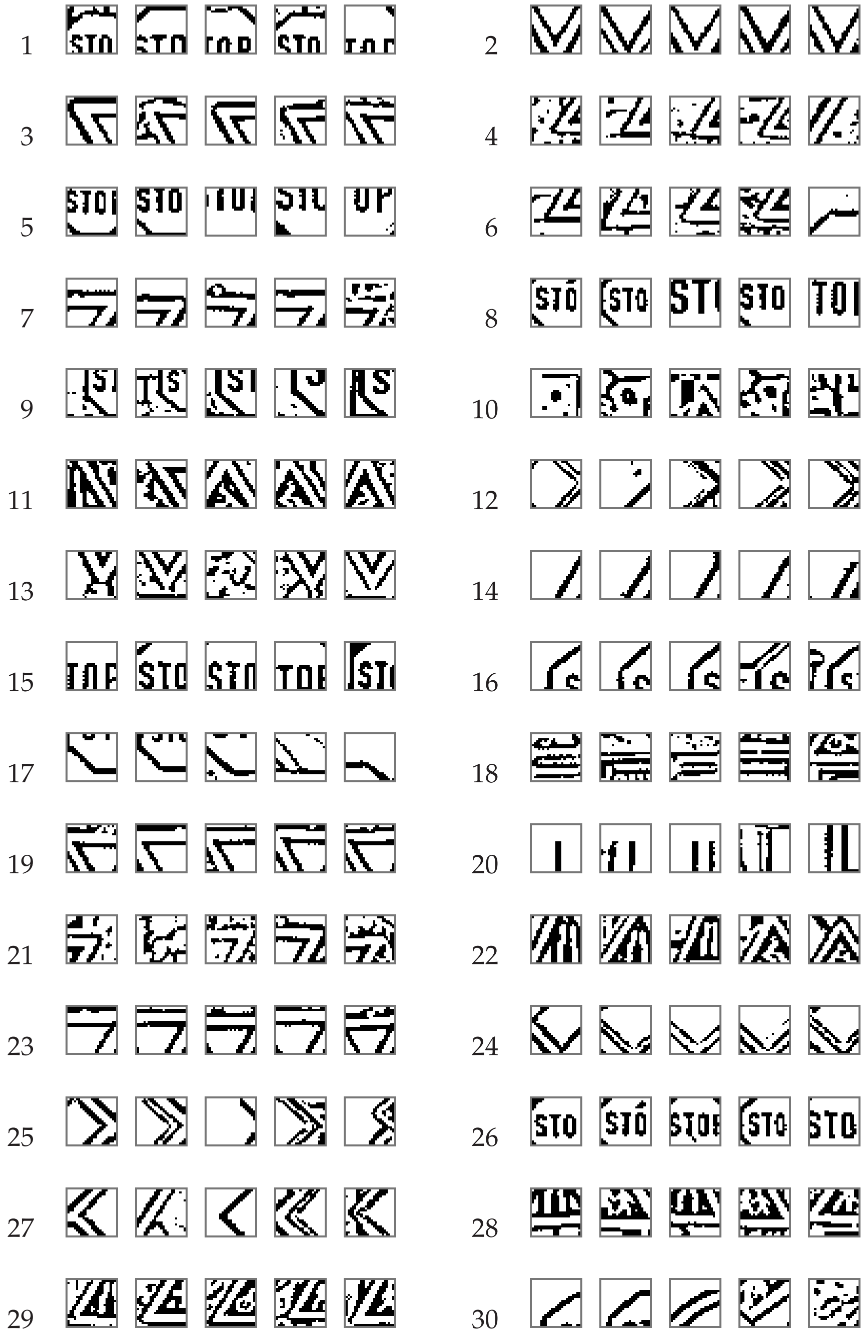


Figure 4.8: *Detected features in layer S2 after self-organization (traffic sign experiment). For 30 selected neurons (of 100 total), five strongly activating receptive field stimuli are shown. Same network as in Figure 4.7*

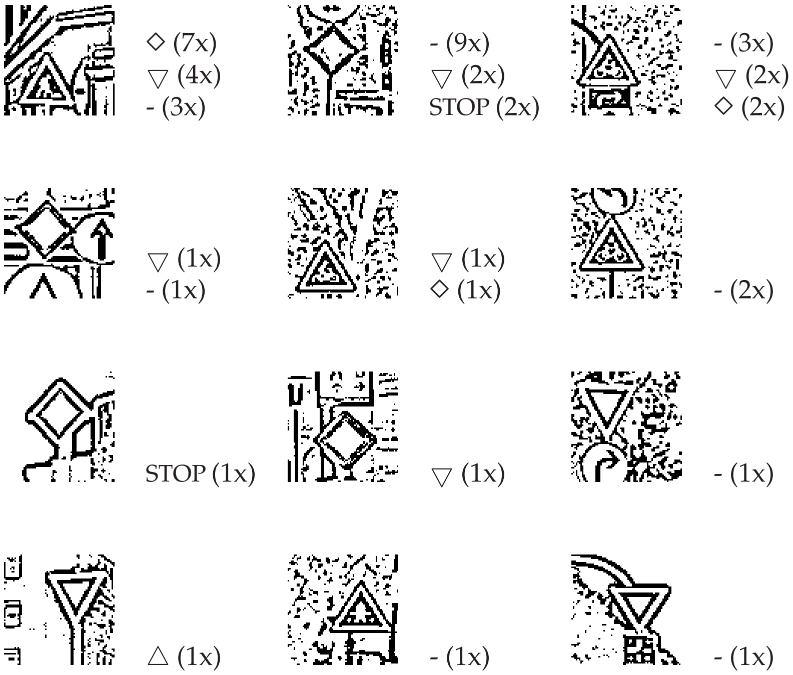


Figure 4.9: *The 12 (out of 100) test images which were misclassified in at least one of the 20 training runs. Symbols to the right of each picture indicate the classes the image was wrongly assigned to, and the number of times the misclassifications occurred. A class of "-" means two or more classes received an equal number of votes.*

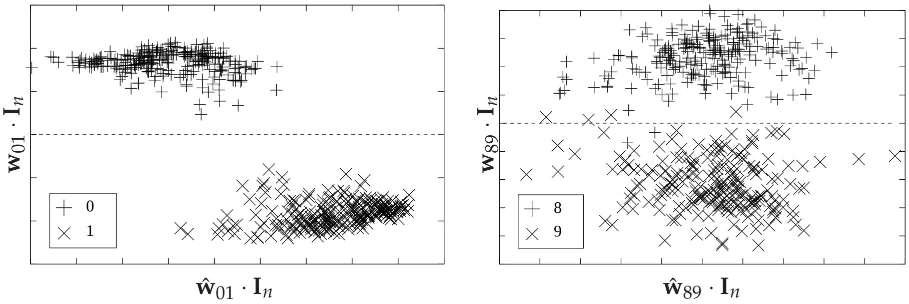


Figure 4.10: *Visualization of the high-dimensional feature space produced by the hidden network layers after self-organization. Each point represents a feature vector corresponding to one input image from the MNIST test set. Left: Feature vectors $\mathbf{I}_n$ of images showing "0" or "1", Right: Feature vectors $\mathbf{I}_n$ of images showing "8" or "9". In the feature space, the image classes are separable by a linear decision border (dashed line).*

4.2 Properties of the Training Method

The developed training method—self-organization in the hidden layers, and Perceptron learning in the output layer—was shown in the previous sections to produce satisfying results. The primary objective in developing this method was its ability to train large networks of threshold neurons, as provided by the present hardware system. However, the algorithm turns out to have more interesting properties, among which are its robustness to computation errors and its ability to be operated in a chip-in-the-loop fashion which will be examined in detail in the next chapter, p. 81. Some other properties will be be highlighted in this section. Only the MNIST data set is used for demonstrations. The traffic sign problem is a poor basis for thorough evaluations due to the low size of its training and test data sets. In the experiments, the setup was as in Table 4.1, except when otherwise stated.

4.2.1 Self-Organization Produces Linear Separability

The hidden layers can be viewed as a feature extraction stage which transforms the input images into a high-dimensional feature space. Ideally, the images are transformed into a representation in which the different image classes are linearly separable from each other. This basic principle is similar to the one underlying support vector machines or liquid computing approaches [36, 59], where, likewise, a pre-computing stage is used to transform the input data into a linearly separable form.

From the good classification performance measured in the presented networks it can be inferred that an appropriate feature representation is found by the self-organization process. Nevertheless, it is interesting to directly visualize the transformed image data in the feature space. For this, the rather high-dimensional feature vectors being produced by the last hidden layer must be projected onto a 2-dimensional surface. In order to get the optimum impression of the linear separation, the projection is computed using the weight vectors of the (trained) output layer because those represent the directions of maximum separation. A given neuron in the output layer of the digit-recognizing network is responsible for discriminating only between two of the 10 image classes (see section 3.3.2). The weight vector of the neuron trained for the digits p and q shall here be denoted by $\mathbf{w}_{pq}$. In Figure 4.10, left side, the feature vectors of images showing “0” and “1”, produced by an arbitrarily selected trained network, are projected to the surface spanned by $\mathbf{w}_{01}$ and $\hat{\mathbf{w}}_{01}$, where $\hat{\mathbf{w}}_{pq}$ is the component-wise absolute value of $\mathbf{w}_{pq}$ ($\hat{w}^i_{pq} = |w^i_{pq}|$). The the second axis of projection is actually arbitrary. The particular choice of $\hat{\mathbf{w}}_{pq}$ yields visually nice plots. Figure 4.10, right side, shows the same plot for image classes “8” and “9”. In both plots, the decision border defined by the respective output neuron is depicted by a dashed line. Each plot shows only a random sub-set (25%) of all test images available for the respective pair of digits for the purpose of clearity.

The plots demonstrate nicely the linear separability of the image data in the feature space. Note that separation between “0” and “1” is very pronounced, while a decision between “8” and “9” does not seem 100% clear. This observation is in accordance with the intuitive impression that the digits “1” and “0” are quite different in their appearance, but the digits “8” and “9” have many shape features in common. These two examples have been selected because they represent the upper and lower end of of the degree of linear separability found in the network.

4.2.2 Network Size: The Bigger the Better

In neural networks, the number of free parameters can be controlled by the the number of neurons in the network. The number of neurons must be adjusted such that the network is on the one hand complex enough to capture the abstract patterns hidden in the training data, but on the other hand it must be simple enough to avoid fitting against noise (so-called overfitting). Think of fitting a straight line, represented by 100 noisy (x, y) points, with a polynomial of degree 100. The resulting curve will fit perfectly, but abstract knowledge about the data is not discovered.

These considerations are often a critical factor in machine learning [17]. In the presented system, overfitting is not supposed to be a major issue: The output layer is a linear classifier, which is already the most simple classifier known. The hidden layers are trained by self-organization, so strictly speaking, over-*fitting* cannot occur. However, the number of feature planes defines the granularity of the input space clustering: Too few planes won’t develop discrimination power, too many planes will produce features which are not abstract enough: In the extreme case, when the number of feature planes approaches the number of distinct training patterns, each pattern will be “grouped” into its own separate feature class, and no generalization will take place at all.

Experiments are conducted to measure the influence of the number of feature planes in the hidden layers. The other meta parameters are kept as in Table 4.1. Figure 4.11 shows the results. Data points are the average of 10 training runs, except for the three rightmost points, where only 5 runs were conducted due to the large network size. The error bars correspond to the uncertaincies of the average values. In general, more feature planes produce equal or better generalization performance. The turning point, where more

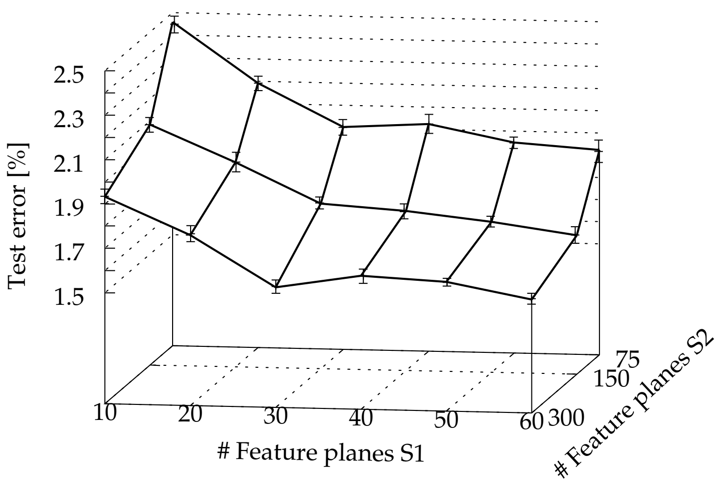


Figure 4.11: *Test error vs. number of feature planes in the hidden layers S1 and S2*

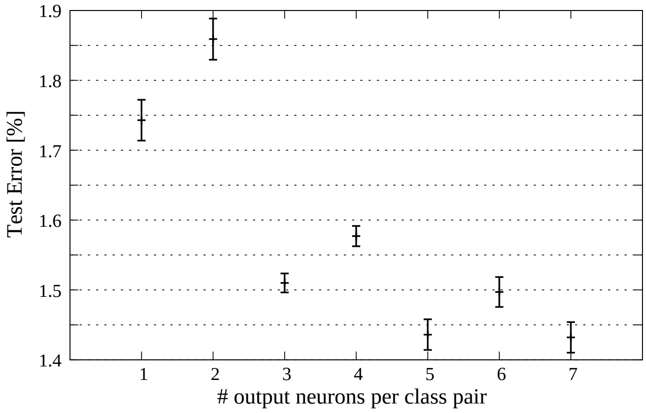


Figure 4.12: *Test error vs. the number of output neurons per class pair.*

feature planes start to impair generalization, is not reached in the tested cases. Hereby it should be noted that largest networks tested (60, resp. 300, feature planes in layers S1, resp. S2) are already huge (also compared to convolutional networks reported in literature). They nearly approached the capacities of the software training system.

In the output layer, the number of neurons is varied by simply replicating the entire layer. Since the final output is obtained by voting, the votes of the additional layer(s) can just be thrown in. During the stochastic training (random initialization + random pattern order), the neurons will develop sightly differently in each layer, and errors will cancel out statistically by virtue of the voting process.[1] The sizes of the hidden layers, and all other meta-parameters, are kept constant at the values in Table 4.1. For each setting, 10 training runs

[1] This is in fact a combination of the two voting schemes described in section 3.3.2

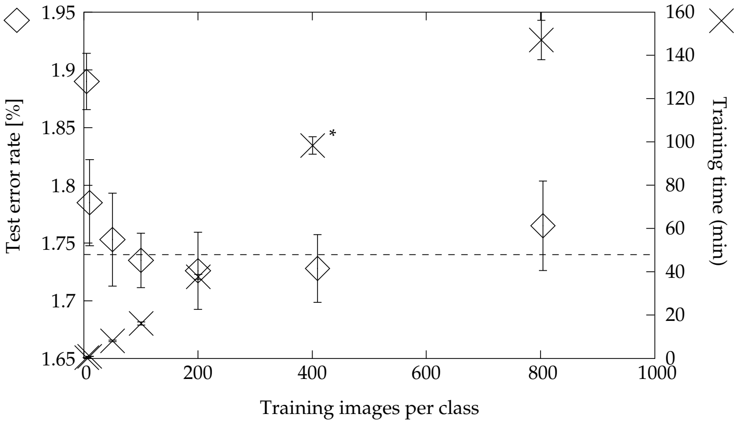


Figure 4.13: *Influence of the number training patterns in the hidden layers. ◇: Error rate on the test set (MNIST digits), ×: real training time needed for the hidden layers. Measured x-axis values are 5, 10, 50, 100, 200, 400, and 800, in this order. *These runs were executed on a different, slightly slower, computer.*

are conducted. The average test errors, and their uncertaincies, are given in Figure 4.12. The results are very interesting: Even numbers of output layers consistently produce worse results than odd numbers of similar size. This phenomenon is not completely understood. It can probably be ascribed to the fact that a stale situation can occur with an even number of voters while with an odd number a decision is always enforced. A software bug as the reason is very unlikely since two independent implementations yield the same result. The general trend however, more output neurons producing smaller test errors, is obvious.

The simple rule "bigger is better" matches well the paradigm of massively parallel analog computing, where additional network size imposes only few space and power cost, and no time cost at all. As long as the network training is done in software, a limit is given by the computing resources available.

4.2.3 Size of the Training Data Set

For the supervised training in the output layer, more training examples generally lead to a better test performance (cf., section 4.3.2). On the other hand, the self-organization process in the hidden layers turns out to require only very few training samples in order to produce a powerful feature extraction stage. The diagram in Figure 4.13 shows the dependence of the test error of the digits network to a varying number of training samples. The hidden layers are trained using the number of training images indicated on the x-axis. Then, the entire training set (60,000 images) is propagated through the already trained hidden layers, yielding the training data for the output layer. Each data point is the average of 10 training runs. Error bars correspond to the uncertainties of the mean values.

The test error reported in section 4.1.1 (1.74%, indicated by dashed line) is achieved at 200 training images per class. Adding more training data does not improve the performance as the achievable classification error seems to saturate quite early. In fact, a significant deviation from the assumed optimum at 1.74% can only be observed when decreasing the number of training samples to 10 per class (as few as 100 training images altogether). This result suggests that, when a new type of images is to be processed, the feature extraction stages implemented by the hidden network layers can be re-trained with only a few patterns, and thus very quickly, to adapt to the new task.

4.2.4 Scalability

The proposed training method has nice scaling properties compared to the frequently used back-propagation algorithm. With back-propagation, the training data must be passed forward, and the errors must be passed back, through the entire network *many times*, in a highly iterative manner. The number of iterations needed until convergence depends on the number of layers and feature planes in the network. In contrast, the method described in this thesis requires only *a single* forward pass, and no backward pass at all.

Moreover, for the hidden layers, the cost for the actual training process (clustering) is independent of the layer dimensions (i.e., the size of the input images), since only one representative hyper-column is trained. With back-propagation, on the other hand, the complexity of the training is at least proportional[2] to the number of image pixels.

4.3 Starting Points for Performance Improvement

The error rate achieved on the MNIST digits data set does not quite reach the best rates reported for convolutional networks trained by back-propagation. The best value found in the literature is 0.95% ([40], and see Figure 4.2), only considering purely neural solutions not including eleborate data preprocessing, e.g., slant normalization.

Tuning the system for a minimum error rate is not the primary goal of this thesis. Rather, the focus is on the general properties of the training methods and the applicablility to an analog hardware implementation. Nevertheless, three example approaches of enhancing the network performance are tested experimentally. The obtained results suggest that there is room for further improvements, given the need and the time.

4.3.1 Using Multi-Valued Inputs

The original MNIST data base consist of grey-scale images with continuous pixel values in the range from 0 to 255. The threshold network presented in this thesis however expects strictly back and white pixels. Thus, the images are binarized before feeding into the network which implies a loss of information. For comparison, a continuous-valued linear classifier trained using MSE (mean square error function) was found to achieve about 14.5% error rate on the binarized images in contrast to reported 12% for the unprocessed grey-value images in the literature ([40], and see Figure 4.2). The drawback

[2] Super-linear dependence can arise due to the higher number of iterations required until convergence.

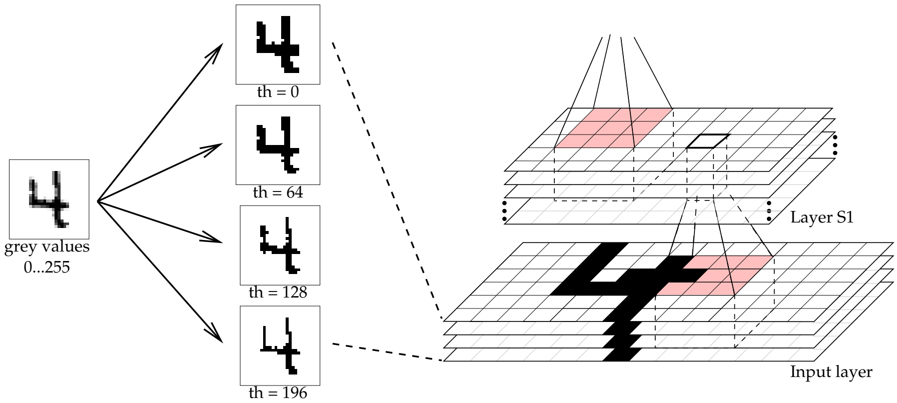


Figure 4.14: *Preserving grey-value information. The original image is binarized with four different thresholds. All four binary images are presented to the network.*

in performance compared to the best back-propagation networks (which use continuous-valued neurons), may hence be partly a result of the binarizing step.

Therefore an attempt has been undertaken to preserve at least some of the information present in the grey value images. The idea is that, principally, the input layer is not restricted to one single plane. By using n planes instead of one in the input layer, n bits of information can be provided at each pixel position. An efficient method to feed binary coded integer values into a hard-threshold network is described in [59], but for the proof of principle we use a simple thermometer code here. The setup, including four input planes, is shown in Figure 4.14. Each of the input planes represents a binarization of the original image for a different threshold. Effectively, the grey scale information is preserved at a 5-level resolution (0, 1, 2, 3, or 4 bits per pixel can be on).

Ten training runs are conducted with the rest of the setup kept as in Table 4.1. The measured mean error rate of 1.67%±0.03% (Figure 4.16, middle bar) is slightly better than for the binary only input (1.74%±0.01%). However, the difference is only about 2 standard deviations, and thus a significant improvement cannot be strictly proven.

4.3.2 Expanding the Training Set

It has been stated that the MNIST training set may be too small to infer generalization properly, which gives rise to expand the data set by applying spatial deformations to the original training images [32, 60]. In the current experiment, a deformation based on Gaussian displacement kernels is chosen. In particular, each image from the MNIST training set is subject to the following transformation: For each pixel position $\mathbf{r}' = (x', y')$ of the transformed image, the corresponding position in the source image $\mathbf{r}$ is computed according to

$$\mathbf{r} = \mathbf{r}' + \sum_{g=1}^{G} \mathbf{d}_g \exp\left(-||\mathbf{r}' - \mathbf{r}_g||^2/\sigma^2\right), \tag{4.1}$$

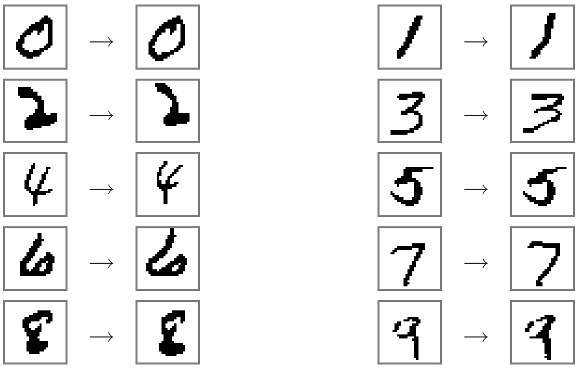


Figure 4.15: *Expanding the training data set by elastic transformations. Shown are examples of the original and transformed images (binarized versions).*

where $\mathbf{r}_g$ and $\mathbf{d}_g$ are the positions and maximum displacements of each of the G Gaussian kernels, and σ is the (uniform) kernel width. The pixel $\mathbf{r}'$ in the transformed image is assigned the grey-value of the original image at position $\mathbf{r}$. Grey-values at non-integer pixel positions are inferred by bi-linear interpolation. The constants $\mathbf{d}_g$ and $\mathbf{r}_g$ are drawn from a uniform distribution, separately for each image to be transformed. Every image from the training set is transformed, doubling the training set from 60,000 to 120,000. The parameters used are $d_g^{x,y} \in [-3, 3]$ (3 pixels maximum displacement), $r_g^{x,y} \in [0, 28]$ (every position on a 28x28 image is a possible kernel center), $\sigma = 8$, and $G = 3$ (3 Gaussian kernels used). These values were chosen *ad-hoc* by visual judgment of the transformation results, and have not been subject to optimization. Deformation is applied to the original grey-value MNIST images before binarization. Examples of the transformed images are shown in Figure 4.15. When training the output units using the expanded training set, the average classification error on the test set goes significantly down to 1.48% (Figure 4.16).

4.3.3 Using Larger Networks

The systematic variation of the network size in section 4.2.2 has shown that, as a general rule, larger networks produce better results. A limit is given at present by the capacity of the computing equipment. The largest networks evaluated during this thesis have 60 feature planes in layers S1/C1, and 300 feature planes in layers S2/C2. A 7-fold pairwise classifier was used as the output layer. All the other meta-parameters are kept as in Table 4.1. This network topology, consisting of altogether 132,615 (partly identical) neurons, has 82,500 free parameters in the hidden layers and 4,630,815 free parameters in the output layer. In 10 independent training runs a mean classification error on the test set of 1.26% ± 0.02% is measured (Figure 4.16), which is also the best value for the MNIST digits data set produced in this thesis.

4.3.4 Suggestions for Further Optimization

As stated above, tuning the absolute recognition rate is not within the scope of this thesis. Nevertheless, three approaches for performance improvement

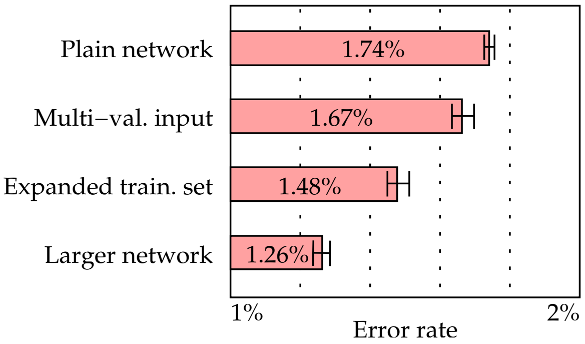


Figure 4.16: *Improving classification performance. Top bar: No improvement applied. Remaining bars: Performance improvement as described in sections 4.3.1–4.3.3.*

were presented on the previous pages. In case better performance is desired in future work, this final section provides some more ideas of where to start with optimizations.

A further enlargement of the training set appears promising. The technique of adding artificial training patterns was frequently used in successful methods before (see e.g., [60]). In section 4.3.2, the size of the training set was only doubled. Other authors reported enlargements by a much larger factor [27, 28].

A search for better meta-parameters is probably worthwile. The actual parameters used in this thesis (4.1) are not guaranteed to be optimal (cf., section 3.5).

Another action worth trying is adding more hidden layers (i.e., a third or fourth S/C layer pair). In this thesis it was not possible to find meta-parameters for a possible third layer pair which would result in a better classification performance than when using just two layer pairs (see section 3.5).

Chapter 5

Robustness Against Computation Faults

5.1 Error Compensation With Chip-in-the-Loop Training

Before operating a neural network on the analog hardware, the pre-computed weights are loaded onto the chip.[1] Due to substrate imperfections and inherent device variations the network response generally differs from the one expected from exact computation. Some of the variations can be compensated for by dedicated calibration circuitry on the hardware (see section 6.2.3). However, the focus of this chapter is to evaluate the effects of assumed hardware faults on the application, and to show ways to cope with them *from the application side*.

For the following investigations it will be assumed that variations in the weight values constitute the dominating distortion effect in the considered hardware architecture. More specifically, we assume that the actual *effective weights* on the chip differ from the explicitely *programmed weights*, according to a distortion model to be specified. In this chapter, two *chip-in-the-loop* training techniques are presented which take these weight perturbations into account during training. Chip-in-the-loop training approaches are particularly suited for compensating perturbations that do not vary over time, so called fixed-pattern errors. Temporal noise, i.e., computing errors which occur and vanish in an unpredictable manner, cannot explicitly be adapted to. However, a chip-in-the-loop method may learn to simply ignore signals exhibiting an especially large amount of noise.

The proposed chip-in-the-loop methods do not require quantification of the hardware errors nor any other explicit calibration routine. Although the suggested methods were developed for the specific convolutional network introduced in the previous chapters, the ideas should be re-usable in other situations without much modification. This is especially assumed for the chip-in-the-loop version of the Perceptron learning algorithm discussed in section 5.1.2, which cares for the inaccuracies in the output layer. The other method

[1] A dedicated on-chip learning circuitry is not present.

(section 5.1.1) deals with computation errors in the hidden network layers (S1 through C2).

5.1.1 Hidden Layers

As detailed in section 3.3, network layers are trained sequentially. Weight training (i.e., clustering) is based on the respective output of the already trained previous layers. In this constellation, it is straight forward to incorporate the hardware into the training, namely by using the hardware (in contrast to software calculations) for passing the training data through the already trained layers: After training a given layer, the trained weights are transfered onto the hardware, and the chip's output is used as training input for the consecutive layer. This way, succeeding layers can adapt to the systematic imperfections of the hardware; or, more precisely: higher-level features are extracted from the actual, distorted, output rather than from the ideal one.

Since the hardware is included in the training loop, we refer to the described procedure as the ”chip-in-the-loop” training method. In contrast, loading a completely software-trained network onto the chip is called “precomputed weights” of “pure software training” during the rest of this thesis. The chip-in-the-loop training is expected to be more robust against computation errors of the hardware.

Please note that, unlike some other chip-in-the-loop techniques, the method just described is not highly iterative. The hardware is only used once per layer, after (software-)training has completed, for passing the training data forward to the next layer. Here is the entire procedure step by step:

1. Train layer S1 (in software); 2. Evaluate layer S1 and C1 in hardware[2]; 3. Train layer S2 (in software) using the hardware output of layer C1; 4. Evaluate layer S2 and C2 in hardware; 5. Train the output layer (see below) using the hardware output of layer C2.

5.1.2 Output Layer

The straight-forward method just described does not work for the output layer because no further layer exists which could adapt to, and thus compensate for, possible errors. Thus, the output layer must be configured such that the *effective* weights on the hardware (in contrast to the *programmed* weights) are optimal. To achieve this, the Perceptron learning algorithm is applied in a highly iterative chip-in-the-loop fashion: Let $\hat{\mathbf{v}}$ be the effective weight vector after the hardware has been configured with the programmed weights $\mathbf{v}$. Then, the update rule (3.3) is applied, with the difference that the actual output O is now computed on the hardware:

$$O = O(\hat{\mathbf{v}}(\mathbf{v})). \tag{5.1}$$

If the algorithm converges, $\hat{\mathbf{v}}$ will be optimal. Note that no explicit knowledge about $\hat{\mathbf{v}}$ is necessary, i.e., there is no need for quantitative error analysis.

Note that in contrast to Perceptron learning in pure software, convergence of the algorithm it is not guaranteed, even if the data are linearly separable.

[2] Remember, C-layers do not require training.

This is a result of the distorted decision hyper-plane: It is generally not perpendicular to the weight vector any more; in fact the decision border cannot even be assumed to be a straight plane at all. For example, a malfunctioning synapse could cause the corresponding programmed weight to grow infinitely if one ore more patterns keep being classified incorrectly due to this synapse failure. However, no ill behavior is observed in the presented experiments if the data are linearly separable. For non-separable cases, the algorithm seems to behave similar to the software version where $\hat{\mathbf{v}}(\mathbf{v}) = \mathbf{v}$ (see experiments section 6.5.1). In the experiments with the artificially degraded hardware system, this algorithm produces unexpected results (section 6.5.2).

5.2 Results

The influence of hardware errors and their compensation by the chip-in-the-loop training methods are tested in a computer simulation. Simulations allow to evaluate the effects of different error modes in isolation, and results are unaffected by the particularities of a concrete hardware substrate. Experiments on the real hardware are presented in chapter 6.

Like in section 4.2, only the MNIST data set is used for the measurements. The traffic sign problem is poor a basis for thorough evaluations due to the low size of its training and test data sets.

Three different modes of synaptic errors are artificially applied to the programmed weights:

"Noise" All effective synaptic weights are subject to adding normally distributed random offsets to the programmed weights.

"Delete" A given fraction of all weights is randomly selected and set to 0, corresponding to disabled synapses.

"Clamp" A given fraction of all weights is randomly selected and set to the extreme positive or negative value (-1 or 1, each 50% chance), corresponding to clamped synapses, e.g., as caused by electric shortcuts.

These three error types are investigated separately. Each error type is applied in turn to the hidden layers only, the output layer only, and, in a third setting, to the entire network. In a first series of experiments, all these settings are evaluated when applying the synapse errors to a completely trained network. This corresponds to loading a complete software-trained network onto the chip, and is referred to as *pre-computed weights*. In a second series of experiments, the chip-in-the-loop training method is employed, as detailed in section 5.1. In particular, the synaptic errors are applied to a network layer immediately after having trained this layer (hidden layers), respectively after each Perceptron learning epoch (output layer). Except for distorting the weights, all network settings are the same as in Table 4.1.

In the real hardware implementation (section 6.2.2), the hidden C-layers are not evaluated on the analog neuro chip. Because of their special connectivity they are not as dedicated for parallel implementation as the S-layers, plus, the simplicity of calculation needed (only 1-bit synaptic resolution) makes digital (hardware-) implementations the appropriate choice. Therefore, when talking about the “hidden layers” in this section, really only the hidden S-layers are referred to. In particular, the C-layers are not subject to distortions.

Before applying the weight errors, the weight vectors are scaled such that the strongest weight has an absolute value of 1, and the thresholds are realized by one or many additional constant inputs with weights of absolute value not larger than 1, in order to match the real hardware setup (cf., section 6.2.2).

The results are shown in Figures 5.1–5.3. Each data point represents the average classification error of ten independently trained networks. For clarity, all diagrams use the the same y-axis scale. As a result, some points with a very high classification error are not displayed. Additional tables including all measurements are given on page 123.

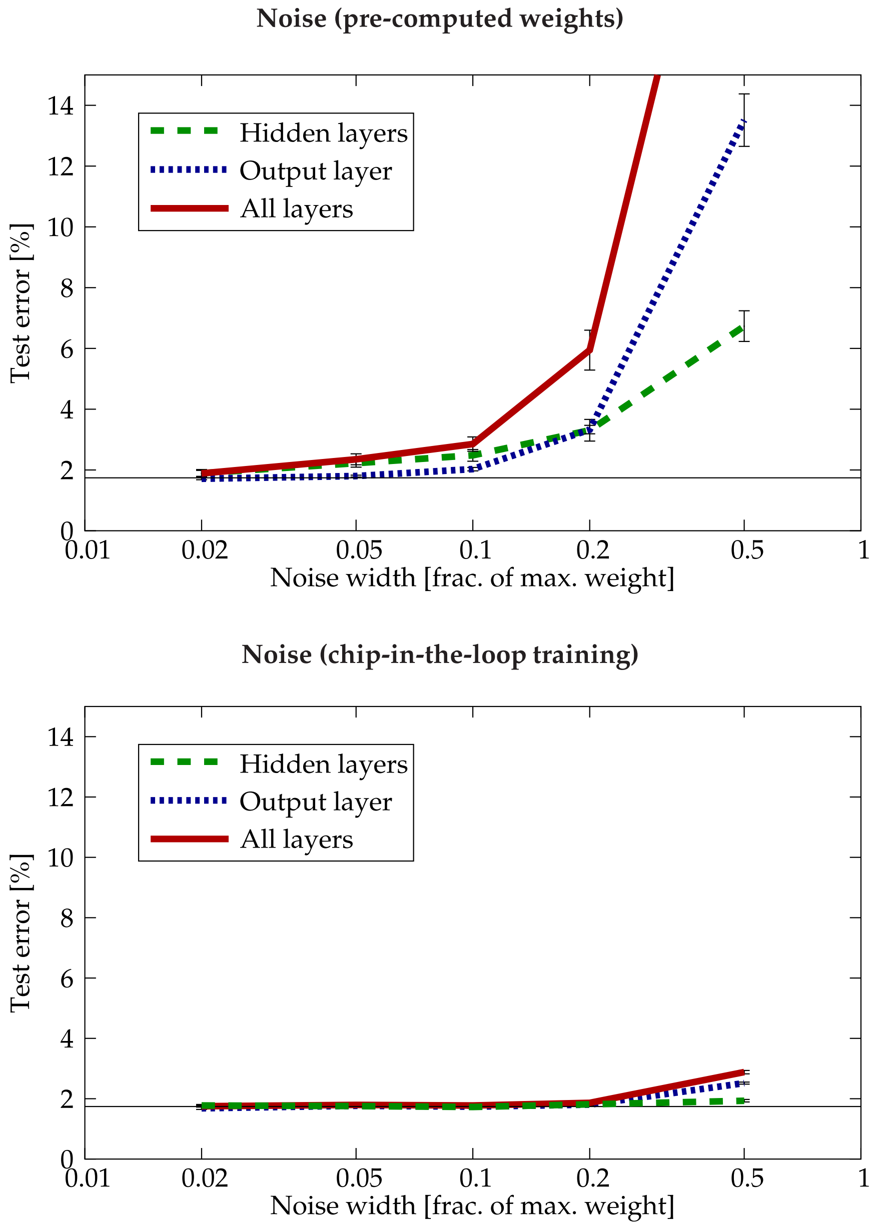
5.3 Discussion

As a general result, the chip-in-the-loop training methods seem to compensate quite well for all the tested types of synaptic errors. However, it should be noted that even without chip-in-the-loop training, which corresponds to spontaneous synaptic errors not seen during training, the performance degrades gracefully (Figures 5.1–5.3, top diagrams). For example, even with 10% randomly deleted synapses in all layers, still over 90% of all images are correctly classified.

Among all error types, “clamp” seems to have the most serious effect on the network. This is plausible since, unlike the other error types, clamping has a high probability to turn a synapse from maximum exhibitory to maximum inhibitory (or vice versa), thus reversing its sense.

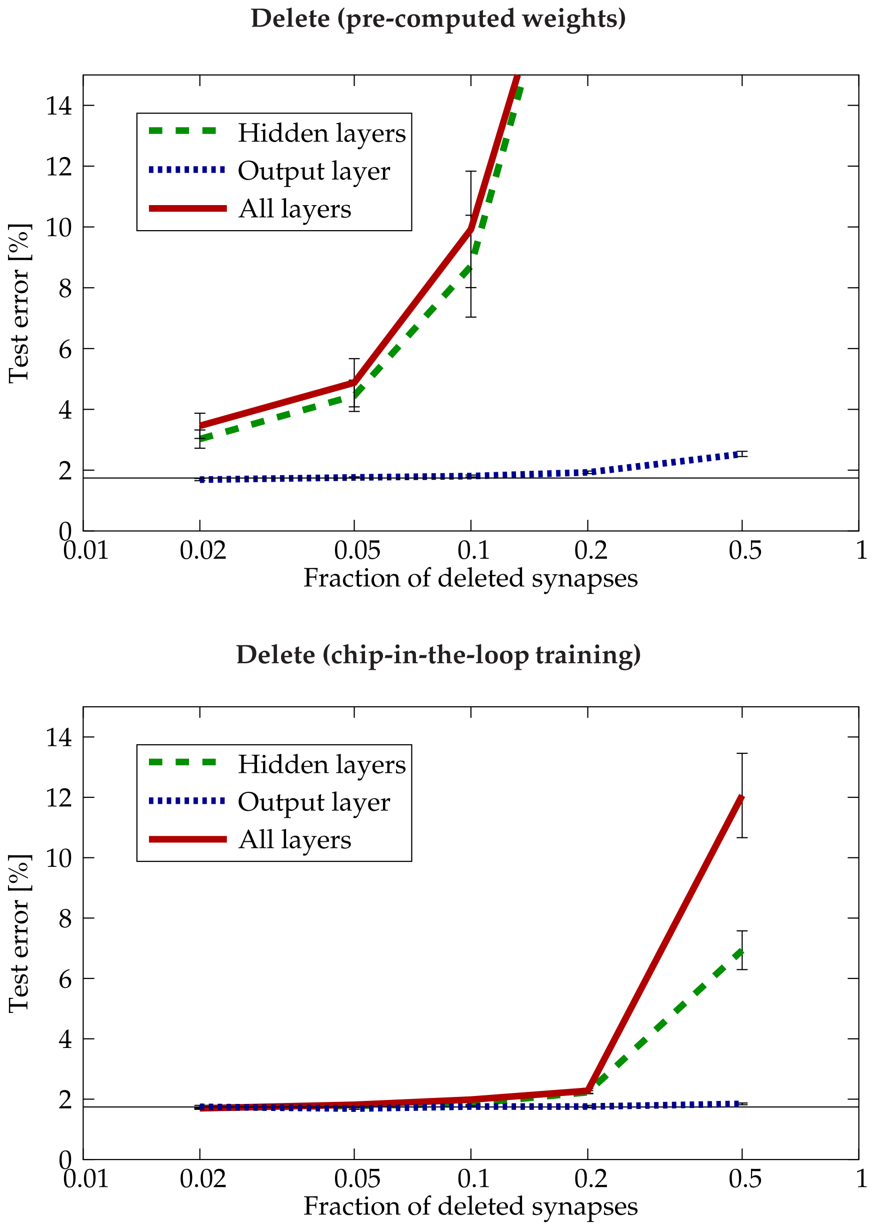
It is interesting to observe that the hidden layers show relatively high sensitivity to the deletion of synapses, but they can cope quite well with large amounts of noise, while the output layer behaves the opposite way. This fact can be understood from the different training strategies applied: The hidden layers are trained by correlation-based learning, which is known to tend to produce extreme synaptic weights [34]. Figure 5.4, left hand side, shows a typical weight distribution in the hidden layers as observed in the trained networks. In such a bimodal weight distribution, adding noise will not easily destroy the overall behavior of a neuron, while setting synapses to zero is in contrast very likely to strongly affect a neuron’s behavior. On the right hand side of Figure 5.4, a typical weight distribution in the output layer is depicted. Here, most weights are close to zero, so deleting synapses will with a high probability affect synapses which do not contribute much to the classification decision. On the other hand, adding random offsets will likely alter a synapse’s strength by a large relative factor given the width of the added noise exceeds the width of the very narrow weight distribution.

The low sensitivity of the output layer to synapse deletion might also be promoted by the large number of partly redundant connections to neuron: Each output neuron receives several thousand inputs that are highly correlated, because by virtue of the blurring C-layers neighboring pixels on a feature plane tend to be in an equal state.


Noise (pre-computed weights)

Noise (chip-in-the-loop training)

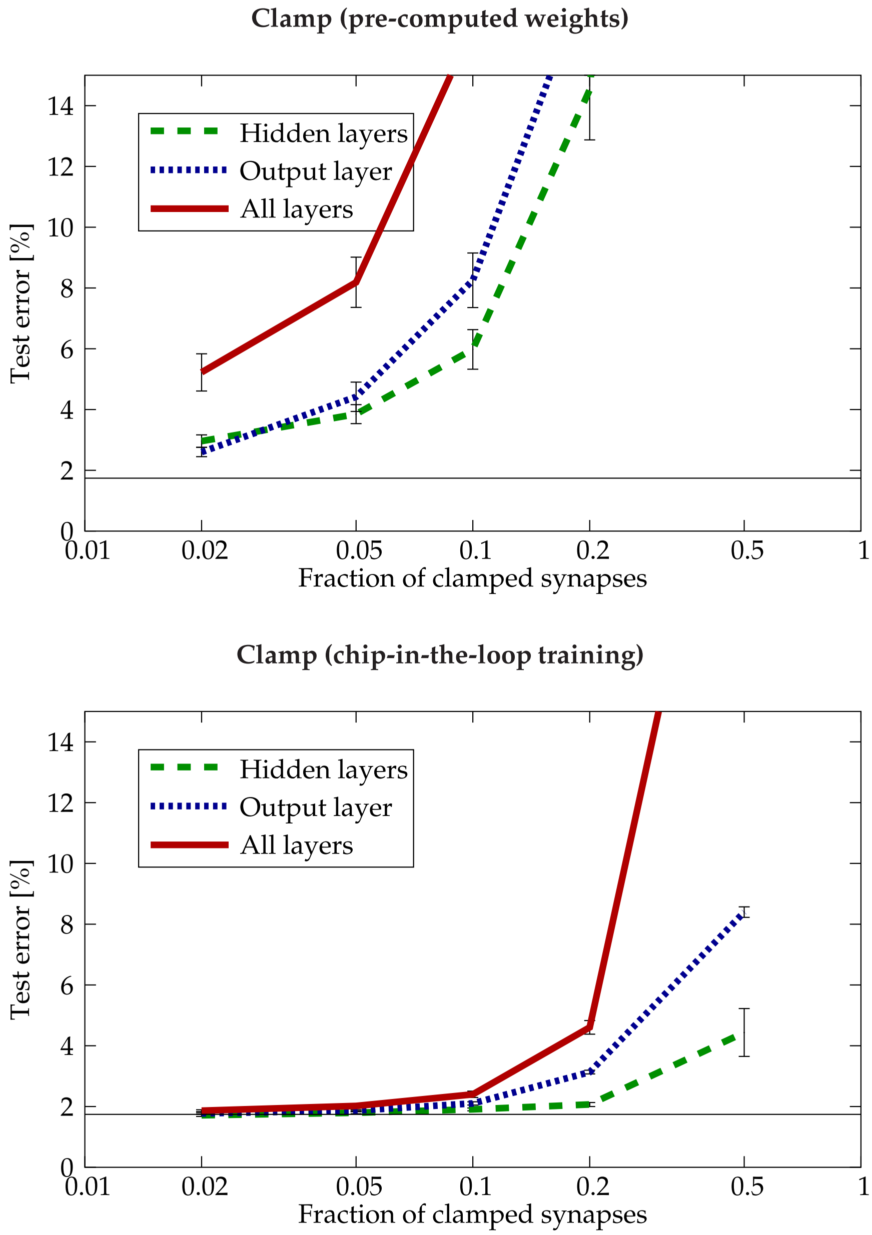
Figure 5.1: *Classification error with noisy synapses. Top diagram: Synaptic errors are applied after training. Bottom diagram: Synaptic errors are incorporated in the training (chip-in-the-loop approach). The horizonal line corresponds to network performance with ideal synapses (1.74% ± 0.01%). Error bars correspond to the uncertainty of the mean. Data points are connected merely for clarity.*



Delete (pre-computed weights)

Delete (chip-in-the-loop training)

Figure 5.2: *Classification error with deleted synapses. Top diagram: Synaptic errors are applied after training. Bottom diagram: Synaptic errors are incorporated in the training (chip-in-the-loop approach). The horizonal line corresponds to network performance with ideal synapses (1.74% ± 0.01%). Error bars correspond to the uncertainty of the mean. Data points are connected merely for clarity.*


Clamp (pre-computed weights)

Clamp (chip-in-the-loop training)

Figure 5.3: *Classification error with clamped synapses. Top diagram: Synaptic errors are applied after training. Bottom diagram: Synaptic errors are incorporated in the training (chip-in-the-loop approach). The horizonal line corresponds to network performance with ideal synapses* (1.74% ± 0.01%)*. Error bars correspond to the uncertainty of the mean. Data points are connected merely for clarity.*

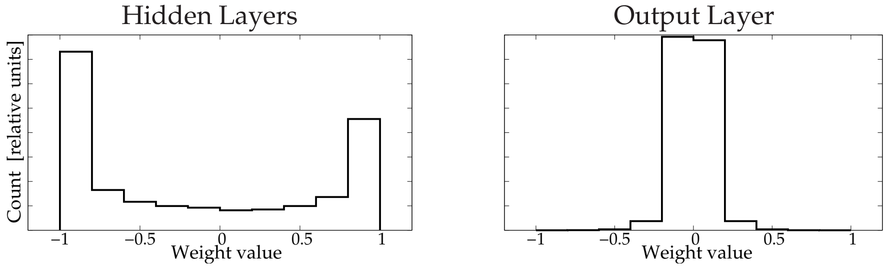


Figure 5.4: *Distribution of synaptic weights typically observed in the hidden layers and the output layer. Weights are forced to the interval [-1,1] by scaling each neuron's weight vector to unit maximum norm after training. Histograms include all weights of one arbitrarily selected trained network.*

5.4 Additional Result: Computing Without Algebra

The particular weight distribution observed in the hidden layers (Figure 5.4, left) in conjunction with the strong robustness against weight perturbations raises the question whether discrete weights that can only take extreme values (-1, 0, +1) might be sufficient for efficient feature extraction.

Such rigorous weight quantization reduces a neuron's task to calculate the Hamming distance between the weight vector and the input vector which in turn requires only most basic computing operations. A synapse with continuous weight values in a continuous-valued network performs a multiply-and-accumulate operation. In a binary-valued network with continuous weight values, only the accumulate operation remains. Multiplication is replaced by a hard condition: *if* input is active *then* accumulate. If further the weights lose their continuous nature and are restricted to the discrete values $\{-1, 0, +1\}$, the synapses do not even need to perform a general accumulation of the form $a := a \pm b$. Rather, only the special increment/decrement operation $a := a \pm 1$ must be performed, which is in fact basic counting: A neuron counts how many of its inputs are equal to their corresponding weights. In the special case of the first network layer, image pixels are pairwise compared to the pixels of a prototype template.

An additional error mode "quantization" is tested where the weights of the hidden layers are quantized using the following scheme:

$$w \leftarrow \begin{cases} +1 & \text{if } w > 0.5 \\ -1 & \text{if } w < -0.5 \\ 0 & \text{else} \end{cases} .$$

Experiment	Test Error [%]
No quantization (from sec. 4.1.1)	1.74 ± 0.10
Weight quantization	2.80 ± 1.06
Weight quantization (+ retraining)	1.80 ± 0.10

Table 5.1: *Classification error with hard weight quantization in the hidden layers. Given deviations are the uncertainties of a single measurement.*

Before quantization, the weight vectors are scaled such that the strongest weight has an absolute value of 1, and the thresholds are realized by one or more additional constant inputs with weights of absolute value not larger than 1, like done in section 5.2.

Again, the quantization is applied either to a completely trained network, or applied successively to each layer. In the latter case, consecutive layers, including the output layer, are retrained with the patterns produced by the quantized previous layer(s) (chip-in-the-loop approach). This time, only the hidden layers are subject to quantization. Apart from the quantization, all network settings are the same as in Table 4.1. Ten runs are conducted without and with chip-in-the-loop training, respectively. The results in Table 5.1 confirm that the network performance is not severely affected by the quantization. As expected, the performance loss is smaller when using the chip-in-the-loop training approach.

In conclusion, a powerful feature extraction stage corresponding to the hidden layers of the evaluated network can be built without any algebraic computing units except for simple counters. A massively parallel implementation of such a system has the potential to provide enormous capacities in terms of data throughput rate at only moderate space and power requirements.

Chapter 6

Hardware Implementation

The development of the methods described in the previous chapters was motivated by the aim of presenting an example of a feasible application for massively parallel analog computing in hardware. A particular prototype chip (see section 2.2.1) was available for this thesis together with a PC-interfaced operating environment. In the course of this work, it turned out that a powerful image recognition system requires more resources than present in the prototype chip (section 6.3.1 “Size Limitations”). Nevertheless, as a proof of principle, smaller networks were evaluated on the prototype hardware and compared to equivalent software implementations. The absolute recognition rates presented at the end of this chapter in section 6.5 are therefore not comparable to the rates reported in chapters 4 and 5.

6.1 General Approach

The considered hardware, introduced in section 2.2.1, uses a massively parallel, array-based architecture. Synapses are laid out in grids where a row of synapses contributes to one neuron. Inputs are fed into the array along the horizontal side, a column of synapses sharing the same input. A synapse array of this kind implements a fully connected single-layer Perceptron with the number of inputs equal to the array width and the number of neurons equal to the array height (see Figure 2.10 on p. 49).

Figure 6.1 illustrates how one S-layer can be computed on such a synapse array. Since all neurons in one hyper column receive the same inputs, it is a natural choice to have the synapse array implement one hyper column. The hyper-columns in one layer have identical weights, so the same synapse array can compute all the hyper columns in one layer without the need of weight reconfiguration. The used neuro chip resides on a main board with an FPGA[1]-based controller and local memory implementing data buffers for input and output data (termed “memory queues” in the Figure 6.1).

The chip features four separate synapse arrays consisting of 64 neurons each. Thus, network layers with more than 64 feature planes must be distributed on multiple blocks. One possible chip configuration is shown in Fig-

[1] Field programmable gate array; configurable hardware logic

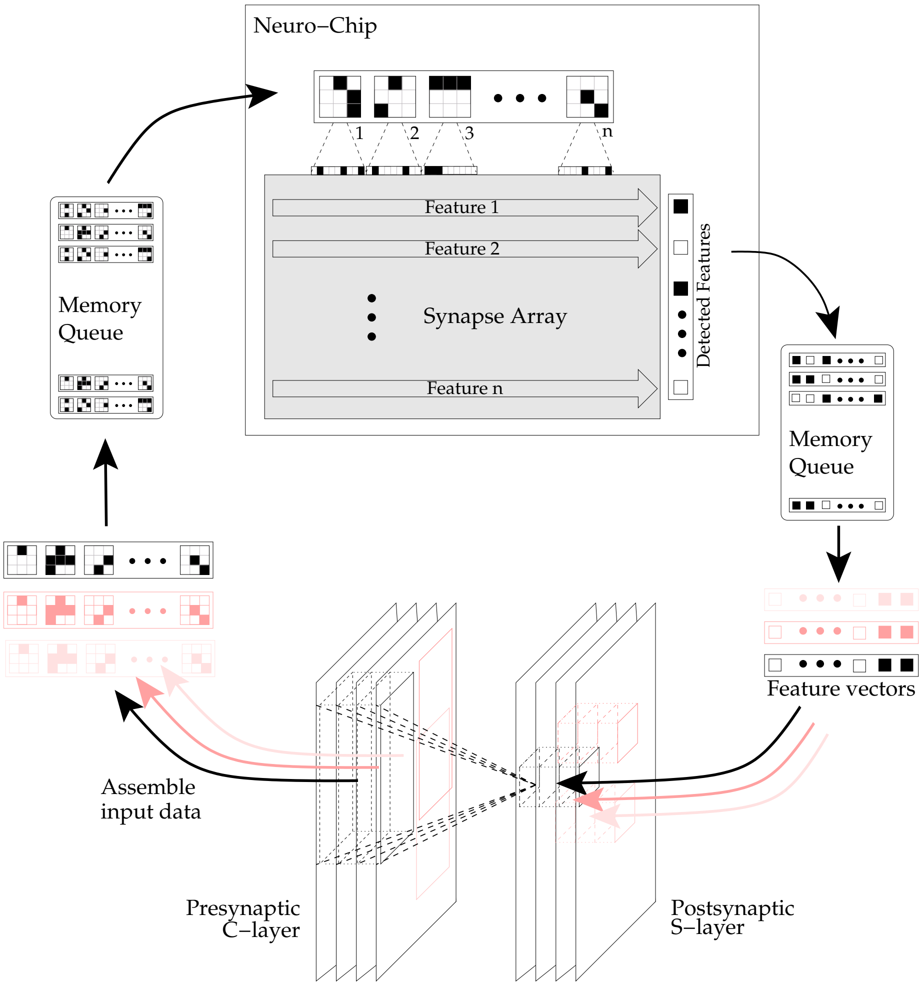


Figure 6.1: *Hardware-implementation of one S-layer. Local patches from the previous C-layer are extracted and stored in the input memory queue. One after one they are fed into the synapse array in form of linear bit patterns. All the n features of one hyper column are computed simultaneously.*

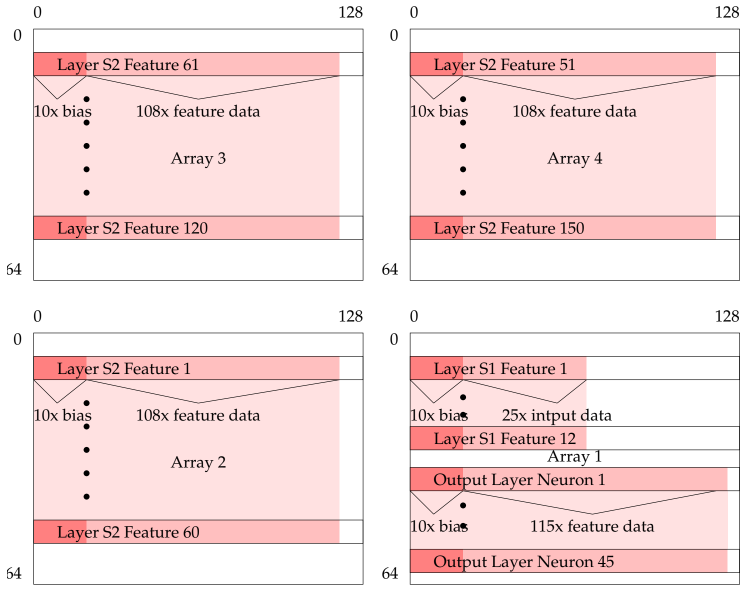


Figure 6.2: *Example usage of the four synapse arrays for the digits network (12 features in layer S1, 150 features in layer S2, 45 output neurons). In this configuration, roughly 75% of the available chip area is used. In the experiments (section6.5), the layout was slightly different.*

ure 6.2. The arrays on the chip accept up to 128 inputs, corresponding to 128 synapses per neuron. This limit imposes additional restrictions on the implementable number of feature planes per network layer (see section 6.3.1).

The preparation of the data that is fed into the synapse array, termed "assemble input data" in Figure 6.1, is done in software. In terms of time cost, this step does in fact constitute a major part of the processing chain (time measurements on page 110).

The C-layers are not considered for analog hardware implementation for two reasons. First, their sightly different connectivity (all neurons in one hyper column receive different inputs) prohibits an adequate exploitation of the parallel array-based system. Second, the simplicity of the C-layer's computation (all weights equal 1) makes them a good candidate for a fast *digital* solution, e.g. FPGA or ASIC[2] based.

Most of the remaining computations take place in the hidden layers S1 and S2: The two S-layers of the MNIST network specified in chapter 4 perform 8,526,000 multiply-accumulate operations for each input image. In comparison, the output layer only needs to do 330,750 such operations. Therefore, in the results section 6.5, the case where only the S-layers are computed in hardware and the output layer remains software-implemented, receives special attention. Nevertheless, it is shown that, in principle, a hardware implementation of the output layer is also possible.

[2] Application-specific integrated circuit; custom micro chip

6.2 Implementation Details

6.2.1 Adjusting the Neuron Model

The HAGEN chip features binary neurons with *unipolar* data signals $I, O \in \{0, 1\}$. In contrast, in the previous chapters a neuron model with *bipolar* signals $I, O \in \{-1, 1\}$ was used. This slightly different neuron model was chosen in the software simulations mainly for a simpler formulation of the training algorithms. For instance, when using bipolar neurons, both the weights and input data can be represented in the same vector space.

The unipolar and bipolar representations are equivalent and are easily transformed into each other by an adjustment of the neuron threshold. To see how this is done we will look at the same neuron in both the bi- and the unipolar representation. Let

O^u, I^u_i, w^u_i, t^u	be the unipolar output, inputs, weights and threshold,
O^b, I^b_i, w^b_i, t^b	be their bipolar equivalents,
$\theta(x)$	be the unipolar step function (1 for $x > 0$, 0 otherwise), and
$\beta(x)$	be the bipolar step function (1 for $x > 0$, -1 otherwise).

The output signal of the bipolar neuron then writes

$$O^b = \beta\left(\sum w^b_i I^b_i - t^b\right). \tag{6.1}$$

The relation between a unipolar signal X^u and its corresponding bipolar signal X^b is

$$X^b = 2X^u - 1, \tag{6.2}$$

as is easily verified. Applying this to I^b_i, (6.1) can be written as

$$\begin{aligned} O^b &= \beta\left(\sum w^b_i (2I^u_i - 1) - t^b\right) && (6.3)\\ &= \beta\left(2\sum w^b_i I^u_i - \sum w^b_i - t^b\right), && (6.4)\end{aligned}$$

and, since $\beta(x) = \beta(ax)$ for all constants a:

$$O^b = \beta\left(\sum w^b_i I^u_i - \frac{\sum w^b_i + t^b}{2}\right). \tag{6.5}$$

Using again (6.2) and $(\beta(x) + 1)/2 = \theta(x)$,

$$O^u = (O^b + 1)/2 = \theta\left(\sum w^b_i I^u_i - \frac{\sum w^b_i + t^b}{2}\right). \tag{6.6}$$

Comparing (6.6) with the standard form of the neuron equation $O^u = \theta(\sum w^u_i I^u_i - t^u)$ we arrive at the transformation formulas:

$$\begin{aligned} w^u_i &= w^b_i && (6.7)\\ t^u &= \frac{\sum w^b_i + t^b}{2} && (6.8)\end{aligned}$$

As a conclusion, before loading a trained network onto the chip, the neuron thresholds must be adjusted according to (6.8). The weights can be reused without any change.

6.2.2 Weight and Threshold Scaling

The chip's circuitry has a technical limit for synaptic strengths which, in the used control software, corresponds to programmed weights of ±1. In order to attain maximum computation accuracy, the weights should be as large as possible while at the same time not violating this limit. Throughout this thesis, each weight vector is therefore normalized to unit maximum norm (i.e., $|w_i| \leq 1$ for all i, while $w_i = 1$ for at leat one i) before loading it onto the hardware.

The neurons considered in the previous chapters compute their activation according to $\sum w_i I_i - t$, where w_i are the weights and t is the threshold. On the network chip the threshold is fixed to $t = 0$. Neurons with $t \neq 0$ are therefore realized by an additional, constantly active input with a weight of $-t$.

When calculating the scaling factor for weight normalization, only the w_i, but not t is considered, resulting in possible values of $|t| > 1$ after normalization. The reason for not treating the threshold as a regular weight in the scaling procedure is again accuracy. The proposed training methods often produce neuron configurations where $|t|$ exceeds the modulus of the strongest regular weight. Thus, incorporating t in the calculation of the scaling factor would decrease the effective resolution of the regular weights. The problem of large t can be easily solved by reserving a few hardware synapses (typically 5-10) for bias inputs. For example, a threshold value of $t = n + a$, where $n \in \mathbb{N}$ and $a \in [0, 1)$, is then realized by setting the first n bias weights to $+1$, the $(n+1)^{\text{th}}$ weight to a and the rest to 0. A situation where $|t|$ is larger than the number of bias weights is resolved by adjusting the scaling factor for all weights until $|t|$ is equal as or smaller than the number of bias synapses.

Another limit for synaptic strength is given by the fact that non-linear behavior is observed for high total activations. The underlying reason is the limited dynamic range of the circuit comparing the summation currents I^{pos} and I^{pos} (see section 2.2.1). In test measurements, a linear behavior is observed until $I^{\text{pos}} + I^{\text{pos}} \approx 660\mu A$. In the used setup, a weight with maximum strength ($w = \pm 1$) produces a synaptic current of $\pm 22\mu A$. Thus, the sum of the absolute values of all weights must not exceed 30, or, in a formula: $\sum |w_i| I_i + |t| \leq A_{\max}$ with $A_{\max} \approx 30$. This inequality must be ensured by additional weight scaling.

In summary: Given N_b reserved bias inputs, a preliminary scaling factor γ_1 is computed as

$$\gamma_1 = \begin{cases} 1/\max_i |w_i| & \text{if } |t|/\max_i |w_i| < N_b + \epsilon \\ (N_b + \epsilon)/|t| & \text{otherwise} \end{cases} , \tag{6.9}$$

accounting for the condition $|w_i| \leq 1$ and the bias issue. Then, in order to ensure the linear regime, the final scaling factor is:

$$\gamma_2 = \begin{cases} \gamma_1 & \text{if } \gamma_1(\sum |w_i| + |t|) \leq A_{\max} \\ A_{\max}/(\sum |w_i| + |t|) & \text{otherwise} \end{cases} , \tag{6.10}$$

The weights are then scaled according to

$$w_i \leftarrow \gamma_2 w_i, \quad \text{for all } i \tag{6.11}$$
$$t \leftarrow \gamma_2 t. \tag{6.12}$$

The constant $\epsilon \approx 1$ makes sure that enough room is left for additional bias needed due to possible device variations (see next section).

6.2.3 Calibration of Fixed Offsets

The algorithms developed in this thesis are claimed to work in the presence of unspecified computing errors in the synapses. Effects of variations in the neuron body computation, which is basically the computation of the threshold function $\theta(x)$, have not been studied since offsets in the switching point (ideally at $x = 0$) can be accounted for by calibration as described in the following section.

Neuron Offsets. The switching point of a hardware neuron on the prototype chip is generally not at $\sum w_i I_i - t = 0$, where w_i are the effective synaptic weights, but rather at a value ϑ slightly less or greater than zero, which we call the *neuron offset*. As a simple approach, one might suggest to measure this displacement for each neuron and compensate it by a dedicated bias input which is constantly active and has a weight of $-\vartheta$. The scheme described in [22] does exactly this. It works well if only a few synapses per neuron are used, but it turns out to fail when evaluating networks with a large number of synapses neuron. From experiments done together with the lab member Steffen Hohmann, it became clear that the neuron offsets are not constant but they actually depend on the overall amount of synaptic weight used: $\vartheta = \vartheta(\sum I_i|w_i|)$. According to the chip's developer Johannes Schemmel, a simple linear model is in good accordance with the hardware design (cf., Figure 6.3), where

$$\vartheta = \vartheta_0 + \gamma \sum I_i|w_i|. \tag{6.13}$$

In order to understand why, it is necessary to discuss details of the hardware implementation. A synapse whose input is active (input value equals 1) draws a current proportional to the absolute value of its weight from one of two neuron-global electric lines (see also Figure 2.11, p. 50). Synapses with positive weights use one line, synapses with negative sign use the other. The positive and negative portions of the internal neuron activation are thus accumulated by the Kirchhoff current law, the sums being represented by the total currents I^{pos} and I^{neg}. Further, $\sum I_i w_i$ corresponds to $I^{\text{pos}} - I^{\text{neg}}$ and $\sum I_i|w_i|$ to $I^{\text{pos}} + I^{\text{neg}}$. The total positive and negative currents are separately converted into voltages and compared by circuitry schematically shown in Figure 6.3, right hand side. Basically, two two resistors R convert the total currents into voltages

$$\begin{aligned} U^{\text{pos}} &= R^{\text{pos}} I^{\text{pos}}, \\ U^{\text{neg}} &= R^{\text{neg}} I^{\text{neg}}, \end{aligned} \tag{6.14}$$

and a comparison of these two voltages determines the output state of the neuron. The transistors M are support circuitry ("cascodes") for decoupling the comparator from the large capacitances of the current lines with the aim of speeding up the network operation. The two equal bias currents I_b keep the whole circuit in saturation even if no synapse is active.

The constant term ϑ_0 in (6.13) corresponds mainly to device variations in the current sinks I_b^{pos} and I_b^{neg} which are slightly unequal. Differences between R^{pos} and R^{neg} contribute to the term being proportional to the total synaptic activation $\sum I_i|w_i|$.

The gain γ in (6.13) is found to be in the order of only a few per cent (given ϑ_0 and w_i are both measured in the same units, e.g. in synaptic least significant

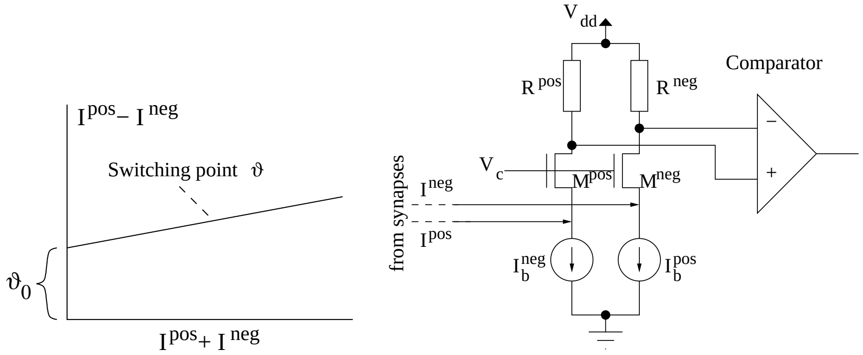


Figure 6.3: *A neuron's switching point is dependent on the total synaptic current. Left: The variation of the switching point can be approximated by a linear model. Right: Schematic of the neuron circuit (after [54])*

bits). So, if only a small number of synapses are used (implying $I^{\text{pos}} + I^{\text{neg}}$ being small), the neuron offset is dominated by ϑ_0. With growing synaptic currents, as more synapses are activated, the variable term can be well in the order of magnitude of ϑ_0.

As a conclusion, a general compensation of the switching point is not possible by just adding a constant offset through a bias input. For full compensation, a way to fine-tune the neuron circuitry would be additionally required. Unfortunately, this feature is not present in the current hardware, but it can be added in future implementations.

In this thesis, calibration is performed only using a constant offset, but care is taken that the chosen bias value is optimal for the range of activation produced by the data actually processed. Therefore, the measurement of ϑ is not done with artificially created input vectors, but with input vectors corresponding to real receptive fields from the training data. The following program shows how offset calibration is done when loading a neuron onto the chip.

0) For loading a neuron defined by its ideal weights w_i and threshold t onto the chip, do the following steps:

1) Load the synaptic weights w_i (excluding the threshold t) onto the chip while reserving space for a few (say, n) bias inputs.

2) Take the input data set $\mathcal{T}$ the weights were originally trained with.

3) Divide $\mathcal{T}$ into two subsets $\mathcal{T}^+$ and $\mathcal{T}^-$ depending on the response of the ideal neuron. Call this partition the desired classification. For any programmed bias $t_{\text{chip}} \in [-n, n]$ we define errors $E^+ =$ (number misclassified patterns from $\mathcal{T}^+)/||\mathcal{T}^+||$, and similarly E^-.

4) By the method of nested intervals find a t_{chip} such that $|E^+ - E^-|$ is minimal.

If there exists a t_{chip} for which the desired classification is reproduced, this t_{chip} will be found. Otherwise, the result is a suitable approximation of the desired classification. For an ideal neuron without computing errors, and sufficiently dense input data, one would get $t_{\text{chip}} = t$. Sometimes, the data is perfectly separable, but not dense (e.g., in the experiments with the traffic sign data on page 109). Then, $|E^+ - E^-| = 0$ within a range of $t^- < t_{\text{chip}} < t^+$. In such cases we set $t_{\text{chip}} = (t^+ + t^-)/2$.

The described calibration routine is applied for the neurons of the hidden S-layers only, and for output neurons when loading pre-computed weights onto the chip. When using the chip-in-the-loop training for the output layer, calibration is implicitly performed by training algorithm.

The input data set $\mathcal{T}$ in the above calibration routine is usually very large, since it consists of every possible input region for each input image. For example, in the first hidden layer with an input image size of 28 x 28 and 1000 training images, there are 784,000 training patterns. In order to reduce the data set to a processable number, duplicates are removed from $\mathcal{T}$, and then, from the remaining unique patterns, the 10000 patterns are kept which are most close to the neuron's desired decision border.

6.2.4 Optimizing Training Speed by Cumulative Weight Update

The output layer is trained using the Perceptron learning rule. In its stochastic form as described in sections 3.3.2 and 5.1.2, a weight update is done after each training pattern seen. The algorithm can in principle be conducted also in "batch" mode, where the weight updates are not immediately applied, but the weight increments are accumulated over an entire loop through the training set, and applied at once. If the training patterns are highly correlated[3], the stochastic method converges usually much faster than the batch version [61].

When using the stochastic method in the chip-in-the-loop version, the weights would have to be updated after every training pattern processed. In theory, all the 32,000 synapses of the network chip can be reconfigured in about a tenth of a millisecond. Still, updating the weights after each single pattern is not very efficient, due to the necessary data transfer to and from the hardware, and other overhead associated with one network run. In practice, the maximum training speed can be achieved by evaluating a number of training patterns at once (but less than the whole training set, as in the classic batch mode) and then updating the weights with the accumulated modification: Let $\mathbf{v}$ be the weight vector of the neuron to be trained. In each training epoch, U training patterns $\{\mathbf{J}_i, \quad i = 1 \dots U\}$ are selected randomly from the training set, and are evaluated. Then, the weight vector $\mathbf{v}$ is updated according to (cf., equation (3.3)):

$$\mathbf{v} \quad \leftarrow \quad \mathbf{v} + \sum_{i=1}^{U} \Delta_i, \quad \text{where} \tag{6.15}$$

$$\Delta_i \quad = \quad \begin{cases} -O_i \mathbf{J}_i, & \text{if } O_i \text{ is incorrect} \\ 0, & \text{if } O_i \text{ is correct} \end{cases} . \tag{6.16}$$

Here, O_i, the outputs for the patterns $\mathbf{J}_i$, are all computed using the same (old) weight vector. As a result, a weight update is necessary only every U^{th} pattern. Therefore we call U the *update interval*.

[3] This is the case in the tested benchmarks. E.g., two instances of the digit "3" look very similar.

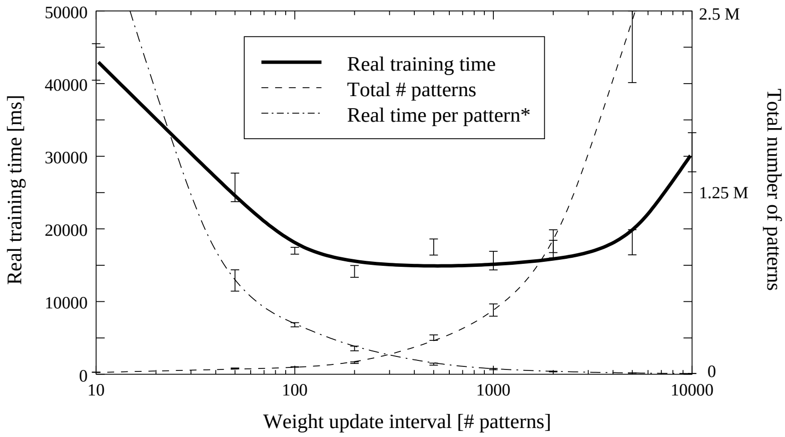


Figure 6.4: *Chip-in-the-loop Perceptron learning. The fat line shows the total training time on the used hardware setup for one output neuron until convergence. A minimum exists for weight update after every 100-1000 patterns. Dashed curve: Total number of training patterns seen until convergence. Dotted-dashed curve: training time per pattern (quotient of the two other curves). Spline interpolations are merely for a clearer visualization. *For this curve, divide the figures on the left y-axis by 12,000.*

The effects of the size of the update interval were tested for several output neurons, all giving similar results. Figure 6.4 shows the real time until convergence for one of the tested output neurons of the MNIST experiments. Also shown (dashed lines) are the total number of patterns processed until convergence, and the mean time spent per training pattern (computed as the quotient of the two other curves). Apparently, more patterns are necessary when larger update intervals are used, being in accordance with the assumption that the stochastic update policy converges faster. On the other hand, with larger update intervals, more patterns can be processed per second because the fixed operation costs of the hardware carry less weight.

For the particular software/hardware setup used the total training time seems to be minimal for update intervals between 100 and 1000 patterns. Throughout the hardware experiments an update interval of 1000 is used.

6.3 Limitations of the Prototype System

The entire hardware system used in this thesis does not consist only of the neuro chip HAGEN. It is a complex aggregate composed of a control PC, a custom-made PCI board including a configurable logic chip as a controller, specifically developed communication protocols, and a software programming interface. The HAGEN chip itself is mounted on another daughter board connected to the PCI board. The various hardware and software components in their present state are the result of a research program conducted by many scientists during the course of several years. This implies that many features of the system are still in a preliminary state and that the usage is not as care-free as one would expect from a commercially available, well documented, apparatus.

One of the shortcomings of the HAGEN chip—a missing calibration function for the neuron gain—has been already discussed in section 6.2.3. Here, more aspects of the system are described, as far as they are related to the reported experiments. Thereby it is understood that these issues can be resolved by investing sufficient amounts of time and money. Since the aim of this thesis is not to deliver a final, marketable product, but rather to evaluate the general applicability of analog computing, the reported experiments were conducted using the hardware system in its current state.

6.3.1 Size Limitations of the Chip

The neurons featured by the prototype chip can receive up to 128 input connections (section 2.2.1). This is still not enough for the large-scale networks used in this thesis. Although, in principle, the used VLSI architecture allows chip implementations with in the order of 10^3 inputs per neuron, in the course of this thesis experiments with smaller networks were done on the existent prototype chip. A similar, but larger, network chip might be manufactured in the future. In this section it is described which layers are affected by the size limitations and how the problem is addressed in the experiments.

Hidden Layers

The number of inputs to a neuron in layer S2 is defined by the size of its input region and the number of feature planes in the preceding layer C1 (see Figure 3.1). Defining the minimum reasonable input region size to be 3 x 3 hyper columns, the number of feature planes in layer C1 (and thus also in layer S1) is limited to 14, corresponding to 3 x 3 x 14 = 126 inputs to a layer S2 neuron, where 128 is the maximum supported by the chip. In fact, some spare inputs are required for the bias (section 6.2.2) and calibration (section 6.2.3) which further decreases the number of effectively usable feature planes to only 12.

The number of hyper planes in layer S2 is, similarly, limited by the number of inputs to the output neurons. However, this limitation is solved in the output layer (see below). The next upper limit is given by the total number of neurons present on the chip, which is in theory 256 (64 neurons in each of the four arrays). However, the two left arrays tend to produce unpredictable output. The reason could not be determined in the course of this thesis. It is not even clear whether the error is rooted in the software interface, in the support system, or in the neural network chip itself. So, in practice, only two of the four synapse arrays are reliably usable in the current setup. Avoiding the first and last rows of each array due to undesired border effects, the maximum number of feature planes for layer S2 is 120.

Output Layer

Each neuron in the output layer possesses several thousand input connections. Even a hypothetical larger implementation of the VLSI architecture might be technically limited to 1,000-2,000 inputs per neuron [56], so the problem of reducing the number of output connections is a general one, not only present for the prototype chip. Instead of just reducing the number of feature planes in layers S2 and C2, similar to section 6.3.1, we keep the all feature planes but ask for statistical methods to prune away unimportant connections from layer C2 to the output layer.

How can we reduce the number of data dimensions while keeping as much information as possible? At the first glance, PCA (principal component analysis) seems the appropriate out-of-the-shelf solution. PCA projects N-dimensional data onto an M-dimensional subspace ($M < N$) spanned by the M first eigenvectors of the auto-correlation matrix of the data set. Since the eigenvectors are generally rotated with respect to the original Cartesian axes, the components of the new M-dimensional data vectors are likely to be linear combinations of the components of the original N-dimensional vectors. However, when working with the hardware, input components are expected to be strictly binary $\in \{0, 1\}$, so any method involving arbitrary linear combinations is not applicable.

Instead, we keep the original coordinate axes and throw away the $N - M$ of them which contain the least information. The following task must be solved: Given J N-dimensional labeled training data

$$\mathcal{T} = \{(\mathbf{I}_1, c_1), ..., (\mathbf{I}_J, c_J)\}, \quad (\mathbf{I}_j, c_j) \in \{0, 1\}^N \times \mathbb{N}, \tag{6.17}$$

where $c_j \in \mathbb{N}$ are the class labels, find the M components of $\{0, 1\}^N$ which are most useful for recovering the c_j if only $\mathbf{I}_j$ were given. Two methods have been tested.

Simple ad-hoc solution. Intuitively, a component of the training data which has a constant value (0 or 1) throughout the data set does not contain any useful information. On the other hand, a component which assumes both binary values may be more adequate, especially if it changes its value simultaneously with c_j. Based on these thoughts, the following ad hoc method of selecting components was perceived:

1. Bring the sequence of training patterns $\mathcal{T}$ into a random order.

2. Traverse $\mathcal{T}$ while counting for each component of $\mathcal{I}_j$ how often it changes its value. In other words, for $n = 1...N$,

$$\text{count}_n := \sum_{j=1}^{J-1} \left| (I_j)_n - (I_{j+1})_n \right|, \tag{6.18}$$

   where $(I_j)_n \in \{0, 1\}$ denotes the n^{th} component of $\mathbf{I}_j$.

3. Select the M components with the highest counts.

The random order of $\mathcal{T}$ ensures that consecutive patterns $\mathbf{I}_j$ and $\mathbf{I}_{j+1}$ belong with a high probability to different classes (assuming the number of classes is much greater than 1). If $\mathcal{T}$ would be ordered by classes, a component fluctuating randomly between 0 and 1 would produce a much higher count than a component perfectly correlated with the class labels.

Mutual information. The mutual information (see e.g., [17]), between two discrete random variables X and Y is a measure for the certainty in Y after

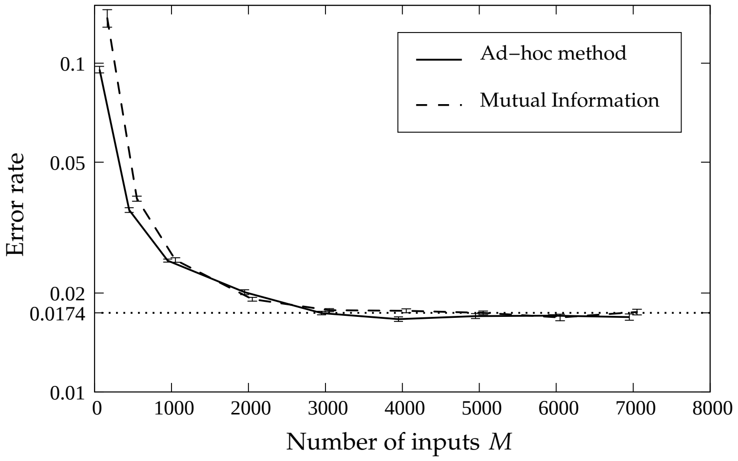


Figure 6.5: *Decreasing the number of input connections to the output layer using two different pruning methods (software simulations). In the original network the number of inputs is 7350 per neuron, yielding a test error of 1.74% (dotted line). The MNIST digits problem was the basis for the test.*

having observed X. Having X assume possible values x from an alphabet $\mathcal{X}$, and similarly Y assume values $y \in \mathcal{Y}$, the mutual information between X and Y is computed as

$$I(X,Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}, \tag{6.19}$$

where $p(x,y)$ is the probability distribution of the joint variables X and Y, and $p(x)$, $p(y)$ are the distributions of each single variable. From the formula we see that for two completely independent variables the mutual information vanishes (because then $p(x,y) = p(x)p(y)$ and $\log 1 = 0$). According to the suggested method, those M components of $\mathbf{I}_j$ are kept which show the highest mutual information with the class labels throughout the training set.

Both methods were applied to the MNIST digits network, which has, in its original size, 7530 inputs for the output neurons (150 planes x (7 x 7) hypercolumns in layer c(4)). From those inputs, sub-sets of varying size were selected and the output layer was retrained. Figure 6.5 shows the measured test errors. The network's performance is expected to decrease with the number of inputs remaining. Apparently, more than half of the inputs can be removed without affecting the result. At 115 inputs (the value is regarded to be appropriate for the HAGEN chip), still roughly 90% of all test images are correctly classified (10% error).

Remarkably, the simple ad-hoc method performs as well as (for number of inputs < 1000 even better than) the computationally more expensive mutual information approach. All results reported in section 6.5 have therefore been produced using the ad-hoc method.

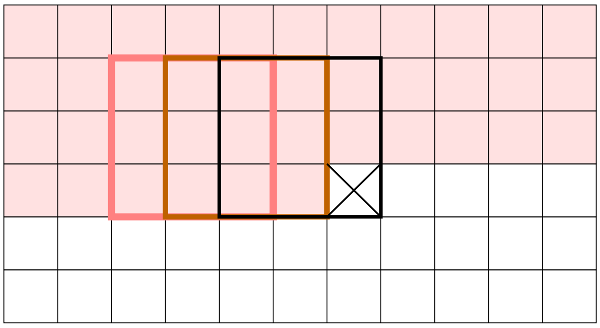


Figure 6.6: *A 3 x 3 window moving across the image line by line. At each window position, only one image pixel (marked with a cross) contains new information. The remaining 8 pixels were already seen by the system before (shaded).*

6.3.2 Data Handling and Transfer

The hardware system was developed with a different application in mind, namely evolutionary training of the synaptic weights. For this, the weights must be iteratively reconfigured at high rates. Dedicated logic elements are present to manipulate the weights locally on the hardware system without any interaction with the control PC [57]. Thus, the data traffic between the PC and the hardware, which is a critical in terms of time consumption, is kept low. The processed data is transfered once to the hardware system and stays constant throughout one experiment. In contrast, in the present application, weights are programmed only once per experiment and stay fixed during the data processing (except in case of the chip-in-the-loop Perceptron algorithm, section5.1.2), so the weight manipulation system is of no use. On the other hand, a high data throughput rate is desired, which is complicated by the following fact: The input data for the neurons consists of local, overlapping regions of hyper columns in the previous layer. In order to be processed by the hardware array, the moving input window is extracted and presented to the weight array as a linear bit string. Since adjacent input fields are largely overlapping (see Figure 6.6), a lot of redundant data is sent to the hardware system. The time measurements in section 6.5.1 confirm that the time needed for data transfer is in the same order of magnitude as the time spent on the actual processing.

If high data throughput should be of importance in the future, an appropriate data processor can be included in the configurable logic chip present in the system, producing shifted variants of the input data. With an input region window of size 5 x 5, as present in the network layer S1, the data traffic from the PC to the hardware could be reduced by a factor 1/25.

6.4 Actual Array Layout

Figure 6.7 illustrates how the convolutional network structure is mapped onto the chip's synapse arrays in the actual experiments reported in section 6.5. Shown are all the weights of one of the digit recognition networks. The network dimensions correspond to Table 6.1, rightmost column. Although the layout of the figure suggests that all four computing arrays of the chip are used, in fact the implementation is realized with only two physical arrays. The other two could not be reliably interfaced in the current hardware setup as stated above. The networks are evaluated in a time-multiplexed way, where at one point in time only one layer (S1, S2, or the output layer) is realized on the chip. Before executing the next layer, the chip is reconfigured.

Interestingly, the properties of the weight histograms in Figure 5.4 can be recognized in Figure 6.7: Ignoring the bias weights at the left border of each array, we observe that the output layer exploits the entire range of possible synapse values, where, however, only few synapses show a very strong absolute value. This corresponds to the narrow, zero-centered distribution of Figure 5.4, right hand side. On the other hand, the hidden layers use basically only two distinct weight values, one positive and one negative one. This corresponds to the bi-modal histogram in Figure 5.4, left hand side. The generally lower contrast seen in the arrays for layer S2 is a result of down-scaling the weights necessary to keep the chip in the linear domain (cf., section 6.2.2).

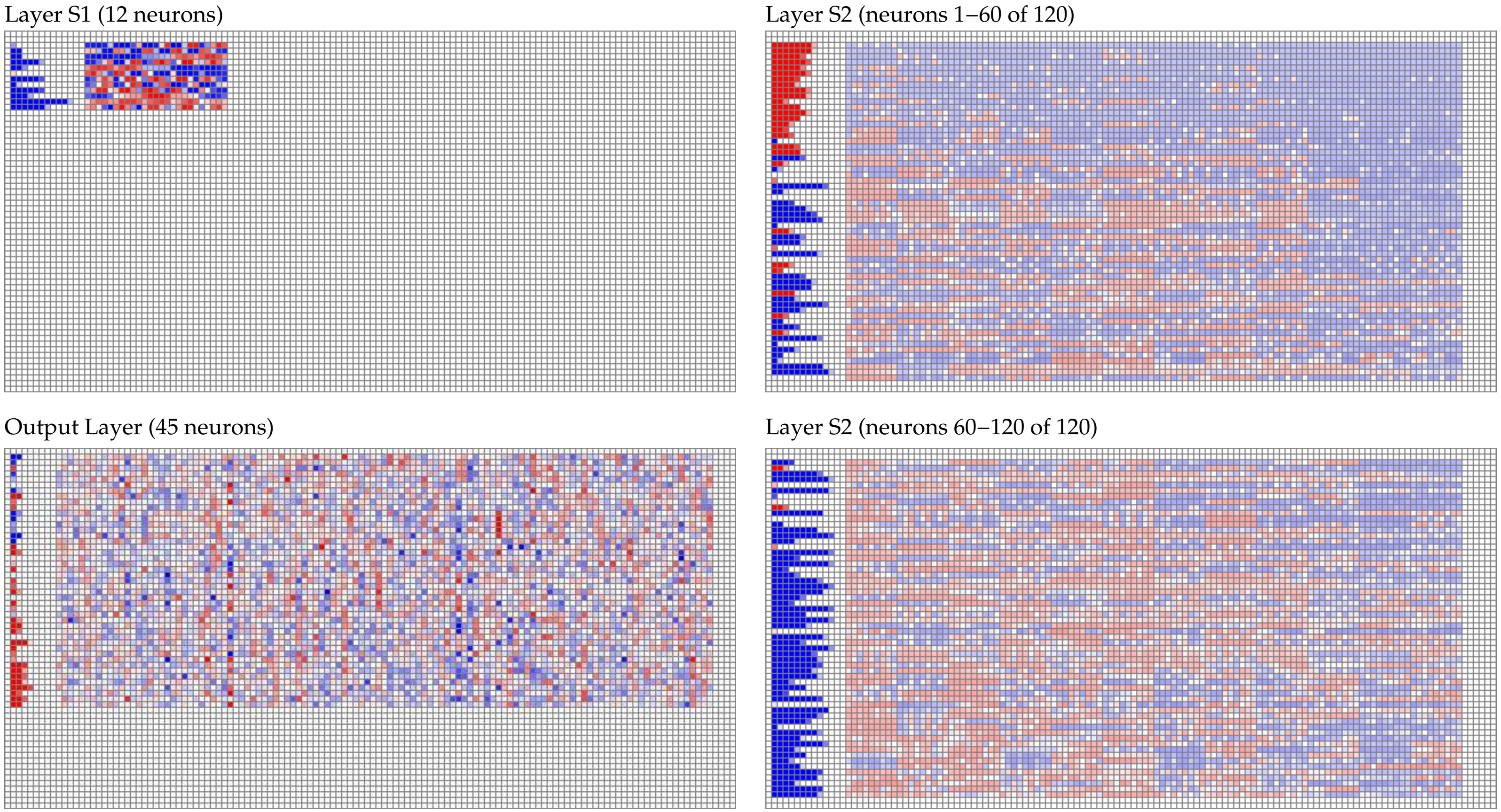


Figure 6.7: *Example views weight arrays configured for the different layers. Layer S2 is distributed over 2 arrays. Synaptic values are color-coded:* ●*=-1,* ●*=1. Each synapse row constitutes one neuron. At the left side of each array, the bias synapses can be seen. More details: see text.*

6.5 Results

The two benchmark problems from chapter 4 are tested on the real hardware setup. Due to size limitations of the prototype chip, the experiments have to be conducted with smaller networks (details in section 6.3.1). Actual network sizes are shown in Table 6.1.

As stated in section 6.1, the vast majority of the computation load is accomplished by the hidden network layers. On the other hand, it is the output layer that must be pruned most severely in order to fit on the synapse arrays provided by the hardware. Therefore, it is a sensible choice to implement only the hidden layers on the analog chip and fall back to digital techniques (software or dedicated hardware) for the output layer. This strategy, which had been also adapted by other authors before [52], is given special emphasis in the presented experiments. Nevertheless, it is also shown that a full network, including the output layer, can be implemented on the chip.

Section 6.5.1 shows results for the case that the chip is operated as intended. In section 6.5.2, the chip's accuracy is artificially degraded and the resulting effects are investigated.

6.5.1 Optimal Hardware Operation

Hidden Layers

In a first experiment, only the hidden layers S1 and S2 are implemented on the prototype chip. The output layer remains to be evaluated in software. Three different training variants are evaluated:

Pre-computed weights. The weights obtained from software training are transfered to the hardware (including offset calibration after section 6.2.3).

Chip-in-the-loop training. The network layers, including the output layer, are trained in a chip-in-the-loop fashion (method in section 5.1.1).

Retrain output layer only. This setup could be called as well "chip-in-the-loop *light*". It is motivated by the fact that each exemplar of the chip shows

	Full SW (chapter 4)	Hidden L. in HW, output L. in SW	Full HW
Digits			
# feature planes L. S1/C1	30	* 12	* 12
# feature planes L. S2/C2	150	* 120	* 120
# conn. to output L.	7350	* 5880	* 115
* Traffic signs			
# feature planes L. S1/C1	25	* 12	* 12
# feature planes L. S2/C2	100	100	100
# conn. to output L.	3600	3600	* 115

Table 6.1: *Network sizes in the hardware experiments. Values affected by the chip's size limitations are marked with stars. Abbreviations: #=number of, SW=software, HW=hardware, L=layer, conn=connections*

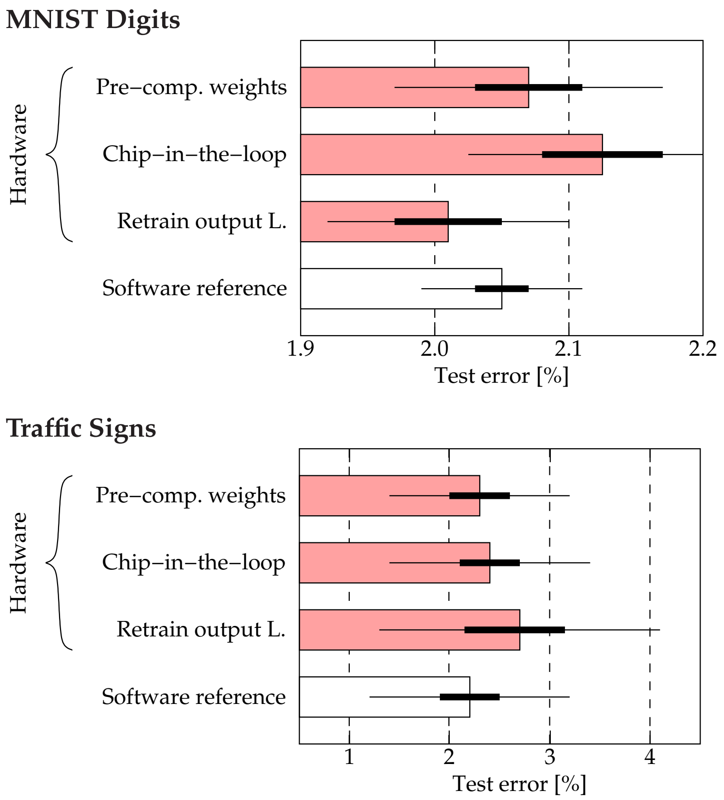


Figure 6.8: *Hidden layers (S1 and S2) are evaluated in hardware. Shown are the results for three training strategies and the software reference. Error bars depict both the uncertainty of the average over all training runs (fat) and the uncertainty in a single measurement (thin).*

> different device characteristics, implying that, in practice, the chip-in-the-loop training must be repeated for every chip produced. In this setup, instead of retraining both layers S1 and S2 using the hardware output, only the output layer is retrained (as done also before in [52]).

As a reference setup, all layers are trained and evaluated in software, with floating-point computing precision, i.e. without incorporating any model of analog distortion.

The results are shown in Figure 6.8 for the two benchmark data sets. All three training strategies produce classification errors not significantly different from the software reference.

Output Layer

A second experiment is dedicated to the hardware implementation of the output layer. The chip's size limitation implies a maximum number of 115 input connections to each neuron in the output layer (section 6.3.1). We will start with the hand-written digits of the MNIST benchmark problem. It turns out that,

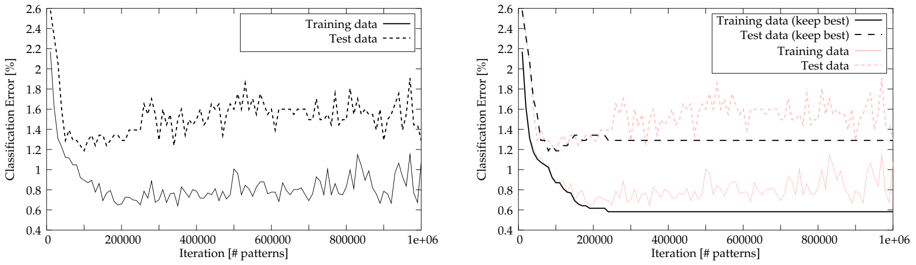


Figure 6.9: *Perceptron learning with inseparable data. Left: The algorithm does not converge, but remains in an oscillating state. Right: Same training run, but the weight configuration with the best training error is kept (curves from other plot inserted as shaded lines for clarity) . Both plots show the training of the neuron discriminating digits "0" and "6" in the output layer of one arbitrarily chosen network in a pure software implementation. The update interval (see section 6.2.4) is 1,000.*

with the restricted number of inputs, most of the 45 data sets to be learned are not linearly separable any more as they were in the pure software implementations where many more input connections could be considered. As a result, the Perceptron learning algorithm does not converge, but yields an oscillating solution. In the observed training runs, the classification errors (of both the training and test set) decrease very quickly to a temporary minimum and subsequently enter a phase of strong fluctuations. The error curves of one typical training run is shown in Figure 6.9, left hand side.

There is no text-book solution for handling this situation. In the experiments presented in this thesis, a combination of two methods is used: First, the weight configuration which yields the least error on the training set *during the entire training run* is regarded as the final solution. Figure 6.9, right hand side, shows the performance of the best weights on the training and the test data. Now, the training error is a monotonously decreasing function of the iteration number, as it represents the up-to-now minimum of the oscillating solution. The independent test error also stays near the lower bound of its oscillating version. However, the test error still seems to cross a minimum. Therefore, the technique of *early stopping* [3] is additionally employed. The same fixed number of iterations is used for training all output neurons. Tests are conducted with the digits images to identify the optimal number of training iterations. For this test, a portion of the training data is reserved as a validation set, as done before in section 4.1.1. For ten networks, the output layers are trained with the remaining training data while recording the classification error on the validation set as a function of the iteration number i.[4] Due to software restrictions, the validation error is not tracked continuously, but is measured only for a few discrete values of i. The validation error for network n after i iterations is denoted by C_i^n. In order to detect a general trend, the C_i^n are normalized over i and then averaged over the ten networks:

$$\tilde{C}_i = \frac{1}{10} \sum_{n=1}^{10} \frac{C_i^n - \langle C_{i'}^n \rangle_{i'}}{\max_{i'} C_{i'}^n - \min_{i'} C_{i'}^n} \tag{6.20}$$

[4] The neurons in one output layer are not trained one after another, but simultaneously, such that at all times, every neuron has experienced the same number of training iterations

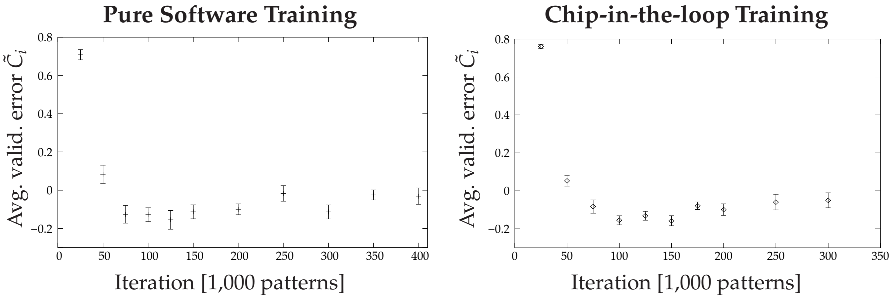


Figure 6.10: *Determining the best point for early-stopping the Perceptron learning. In the average of all tested networks, there is a minimum of the validation error at stopping after* $\approx 125,000$ *training patterns. Software (left) and hardware (right) results look similar.*

where $\langle\cdot\rangle_x$ denotes the mean over all x. The average function $\tilde{C}_i$, together with its statistical uncertainty, is plotted in Figure 6.10. Apparently, at approximately 125,000 seen training patterns, the validation error is quite reliably near its minimal value. Similar behavior is observed for both the pure software training and the chip-in-the-loop training. The training set for one neuron consists of only two classes of the total number of training images, so, 125,000 patterns correspond to approximately ten passes through the training set. The update interval (section 6.2.4) was 1,000 in both the software and hardware runs.

In the final experiments, the training of the output layer is conducted with the entire training set (60,000 patterns) and the network's performance is tested with the test set (10,000 patterns). Training is stopped after 125,000 pattern presentations per neuron. The output layer is evaluated in hardware, both with pre-computed weights and chip-in-the-loop training. In the pre-computed weights setting, the output layer is trained in software and the weights are transfered to the hardware (including calibration according to section 6.2.3). The chip-in-the-loop method is described in section 5.1.2. In a control setting, both training and evaluation of the output layer is done in software. The hidden layers of the network are always trained in software and evaluated in hardware (pre-computed weights). An update interval of 1,000 patterns (see section 6.2.4) and a defined pattern presentation scheme (repeated passes through the training set with a different random order each time) are used for all settings in order to make the results comparable.

The same experiment is also conducted for the traffic sign problem (300 training images, 100 test images). Probably due to the low number of training examples, the problem remains linearly separable in the output layer, even with the restricted number of only 115 input connections. Therefore, early stopping is not applied. The update interval is ad hoc set to 100.

The results are displayed in Figure 6.11. Each bar shows the average classification error of ten independent training runs. The generally very high classification error around 7% is the result of the severe pruning necessary for fitting the output layer on the chip (Table 6.1). The performances of the hardware implementations do not differ significantly from the software reference. As observed also for the hidden network layers, the classification errors of the

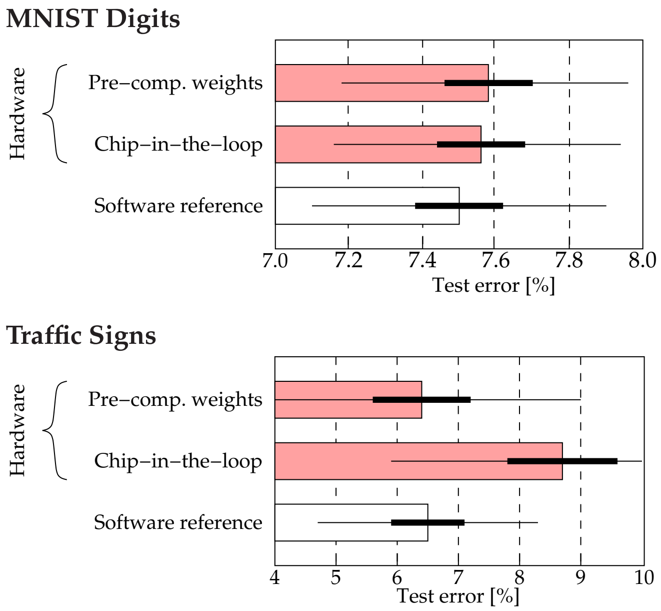


Figure 6.11: *The output layer and the hidden layers (S1 and S2) are evaluated in hardware. Shown are the results for two training strategies and the software reference. The hidden layers are always run with pre-computed weights. Error bars depict both the uncertainty of the average over all training runs (fat) and the uncertainty in a single measurement (thin).*

pre-computed weights and the chip-in-the-loop training are comparable.

The chip-in-the-loop training in case of the traffic sign problem might perform a little worse. It must be said here that the errors on the training data is zero in all the observed training runs, for pre-computed weights, chip-in-the-loop, and the software reference. In such a case, it is plausible that pre-computed weights can perform better than the chip-in-the-loop training: The calibration routine (section 6.2.3) ensures that the decision border maximizes the distances to the two closest training data points, which is often regarded as the optimal strategy. In the chip-in-the-loop training, where no calibration is performed, the exact position of the decision border is determined by chance. However, this consideration cannot explain a possible difference in classification error between the chip-in-the-loop method and the software reference.

Time Measurements

Figure 6.12 show measured processing times spent on average for evaluating one of the digit images (28 x 28 pixels). The total time is broken down into the times spent in the first, respectively second, S/C layer pair, and within each layer pair, into several subtasks. The computing time for the output layer is 0.11ms in software and 0.12ms in hardware, including all overheads. Since this is neglectable compared to the hidden layers, the output layer is not treated in detail. Tasks labeled SW (software) and HW (hardware) are only executed in

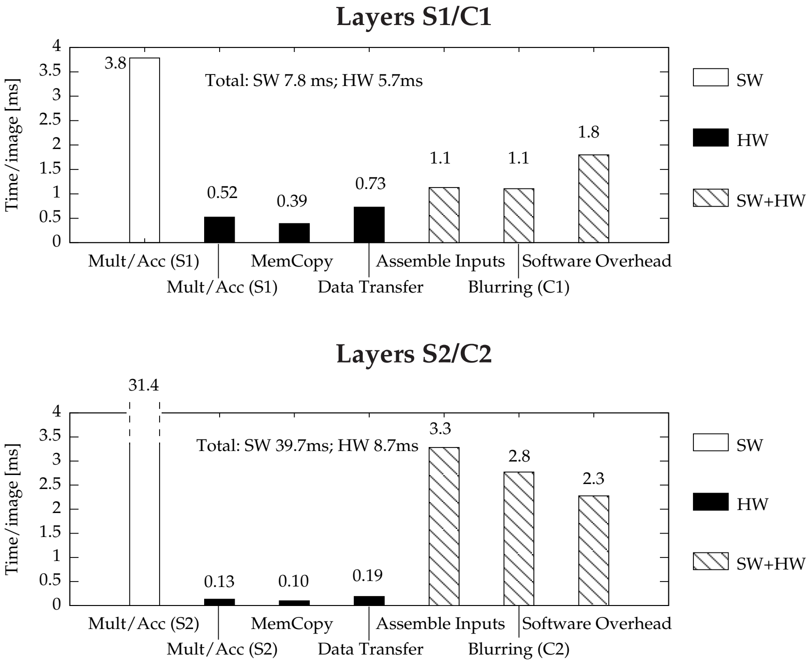


Figure 6.12: *Measured times for processing one digit image (28 x 28 pixels), layers S1 through C2, broken down into separate sub-tasks. The three right-most tasks, labeled SW+HW, are always performed in software, so they are the same for the hardware and the software implementations. The computing time for the output layer is neglectable and is thus not measured in detail. The uncertainties in the numbers are not larger than than a change in the least significant digit.*

the respective setup. The tasks that are labeled HW+SW are performed in both the hardware and the software implementation.

All software parts are executed on a Pentium IV 2.4GHz machine with 512kB second-level cache. The software has been compiled without any system-specific compiler flags. The subtasks itemized in Figure 6.12 are now discussed in detail:

Multiply / Accumulate The multiply-accumulate operations in the S-layers are the parts of the calculation which, in the hardware implementation, are actually performed on the analog chip. The pure software implementation shows that these computations constitute the largest part (more than 70%) of the entire work load. When executed in hardware, the S-layer operations become almost neglectable compared to the other parts of the algorithm. One interesting observation is that, when computed in hardware, layer S2 consumes only roughly a quarter of the time of layer S1, although the number of performed multiply and accumulate operations is more than 12 times as high. The explanation for this is that the parallel chip resources are much better exploited in layer S2. In Figure 6.7 it can be seen that in layer S1 most of the computing array remains unused.

Memcopy and Data Transfer These two items are only relevant for the hardware implementation and account for getting the data to and from the chip. In the present system, the input data must be re-ordered into a form suited for the neuro chip, and the output data coming from the chip must be treated reversely. This process is termed Memcopy in the diagram. The time labeled Data Transfer is the time necessary to transfer the data between the PC's RAM and the hardware system via the PCI interface.

Assemble Inputs The feature planes are stored in the PC's RAM in a bitwise manner, each byte encoding 8 pixels. This saves memory, but makes data access laborious and slow. Therefore, before processing a layer, the output data of the previous layer is copied into a temporary field of floating point numbers where four bytes encode one pixel. In the observed experiments, this field does not exceed 30kB in size and thus fits easily into the processor's second-level cache. The input vector for a neuron consists of local, overlapping patches of the feature planes in the previous layer. Since all the 2-D feature planes are stored sequentially in the linearly addressed field, assembling a neuron's input involves a considerable amount of housekeeping of indices. These two tasks, copying the feature planes and assembling the actual input vectors, are summarized in the task labeled Assemble Inputs.

Blurring The C-layers (also called blurring layers in this thesis) are not considered for analog hardware implementation. As stated in section 6.1, they are not well suited for an implementation on a parallel computing array.

Software Overhead This item is just the difference of the total measured computing time and the sum of all separate parts. It is not related to the actual algorithm. It accounts for the allocation and deletion of data objects, event handling, writing to log files, and the like.

In the hardware implementation, the sub-tasks evaluated in software (assembling the input data and blurring) make up the largest part of the processing time. However, these operations are not very complex (mainly re-ordering and counting is required), so a fast digital hardware implementation, e.g., with an FPGA[5], seems straight-forward. The realization was however not pursued during this thesis, since it was not deemed to yield additional insight.

Discussion

The experiments demonstrate that the computing precision of the investigated hardware architecture, if operated in an optimal setup and appropriately calibrated, is sufficient for evaluating the the developed neural networks using pre-computed weights. However, the calibration routine as used in the current setup (section 6.2.3) depends on the actual training data, so it must be re-run for each specific problem. This shortcoming is rooted in a special type of analog device mismatch (details in section 6.2.3) which can in principle be compensated by additional calibration circuitry not present in the current prototype.

[5] configurable logic chip

Such calibration can be easily included in future implementations of the architecture. Then, the chip calibration will most probably be data-independent and networks will be transferable between different chip exemplars without any problem-specific adjustments.

In a real-life application, probably only the hidden layers should be computed on the analog chip. Although, in principle, a hardware implementation of the output layer is in possible, it does not seem very efficient: First, the output layer must be pruned severely in order to fit on the chip: The number of inputs to each neuron is limited to 128, compared to several thousand in the software implementation. Even with similar chips 10 times larger as the current prototype, the optimal number of inputs cannot be realized. This pruning leads to a strong increase in the classification error in the observed experiments. Second, the output layer is computationally very cheap (see Time Measurements above) so a digital implementation will not destroy the speed gain from evaluating the hidden layers on the parallel hardware.

6.5.2 Artificially Degraded Hardware

In order to give an impression of the application's capability to cope with an imprecise, or even impaired, hardware system, the network chip is operated under non-optimal conditions. In particular, the magnitude of the synaptic currents drawn from the summation lines are systematically decreased (the chip's operation principle is explained on page 49). Smaller currents are realized by a combination of two methods: first, by physically lowering the reference current I_{ref} used for programming the analog synaptic weights (cf., [54]), second by scaling down the weight values before loading them on the chip[6]. The current I_{ref} directly translates to the current drawn by a synapse with a weight of 1: $I_{\text{syn}} = I_{\text{ref}}/2$ (Formula 3.3 in [59]). Denoting the optimal value of I_{syn} by I^0_{syn}, the total relative decrease of the synaptic current is thus calculated as $1 - (\text{weightscale} \cdot I_{\text{syn}}/I^0_{\text{syn}})$. The actually used values of I_{syn} and weight scaling factors, together with the resulting fraction of the optimal synaptic currents, are listed in Table 6.2.

[6] This combined approach, instead of simply lowering the physical reference current, was motivated by the chip developer Johannes Schemmel who advised against an operation with very low reference currents

I_{syn}	Weight Scale	Percentage of Optimal Current
$22.08\mu A*$	1	100%
$7.78\mu A*$	0.25	8.8%
$7.78\mu A*$	0.125	4.4%
$7.78\ \mu A*$	0.0625	2.2%
$4.03\ \mu A*$	0.0625	1.1%

Table 6.2: *Hardware degradation. Synaptic currents are artificially decreased from their optimal strength by lowering a physical reference current on the chip (I_{ref}) and applying a scale factor to the programmed weights. *Inferred from measured I_{ref} after formula 3.3 in [59]; uncertainty of I_{syn}: $\pm 0.05\mu A$*

Generally, lower synaptic currents increase the influence of noise (both temporal and fixed-pattern) relative to the signal. Additionally, since the weights are transfered to the chip digitally, quantization effects become apparent when scaling the weights down in software.

With each of the settings in Table 6.2 (all other settings are the same as in section 6.5.1), the MNIST digits problem is trained and evaluated. This time, the output layer remains trained and evaluated in software, so only computing errors made in the hidden layers are assessed. The hardware weights of the hidden layers are obtained using the three methods: pre-computed weights, chip-in-the-loop, and retrain output layer (see section 6.5.1 for details). Figure 6.13 shows the results. Each point is the average of 10 training runs. Error bars correspond to uncertainty of the mean. As expected, the chip-in-the-loop training methods are able to compensate for errors present with the pre-computed weights. However, no significant difference can be observed between re-training the output layer, and the full chip-in-the-loop approach.

It is worth noting that the synaptic currents are the main factor contributing to the chip's power dissipation [54]. Thus, decreasing these currents results in a drastic reduction of the power consumed.

Fixed-Pattern Versus Temporal Noise

Chip-in-the-loop training can compensate for so-called fixed-pattern errors, i.e. errors which are caused by systematic device variations and thus do not vary over time. With temporal noise, the chip-in-the-loop techniques are not supposed to work as well, since the system can only adapt to errors which are consistently reproduced for repeated computations. Still, a basic ability for compensating for temporal noise can be expected: Since randomly fluctuating signals do not contain reliable information, the training algorithms might just learn to ignore them while giving more significance to signals which are not as noisy. The effects of temporal noise on the convolutional network application were not tested in the computer simulations (chapter 5) and are deferred to future work.

However, additional measurements shall give an idea of how much both fixed-pattern and temporal noise play a role in the considered hardware architecture. For this goal, first, the deviations between software-implemented and hardware-implemented networks are measured: For a number of software-implemented networks, the four hidden layers, S1 through C2, are evaluated with the test data set, and all neural responses in layer C2 are recorded for later reference. Then, the network weights are transfered to the hardware, including calibration after section 6.2.3, and again the data set is evaluated, while counting how often a neuron response differs from the recorded software reference. The same experiment is repeated for all the degradation settings in Table 6.2. In Figure 6.14, the data points labeled *A* show the average percentage of responses that disagree with the recorded responses. The measured differences include both fixed-pattern and temporal variations.

The same experiment was conducted again, but now, another hardware run of each network is taken as the respective reference. This way, the temporal hardware fluctuations are measured in isolation (Figure 6.14, data points *B*).

One conclusion that can be drawn from Figure 6.14 is that, when loading pre-computed weights onto the hardware, the software reference is reproduced

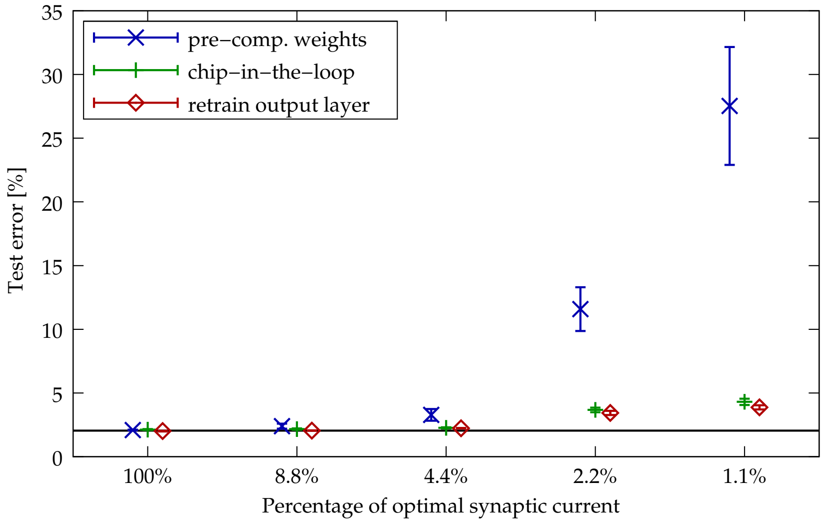


Figure 6.13: *Hardware degradation of the hidden layers by lowered synaptic currents (cf., Table 6.2). The output layer is evaluated in software and is thus not subject to degradation. Shown is the test error of the MNIST problem for three training methods (pre-computed weights, chip-in-the-loop, and retrain output layer). The synaptic currents constitute the major part of the chip's power consumption. Black line = software reference from Figure 6.8.*

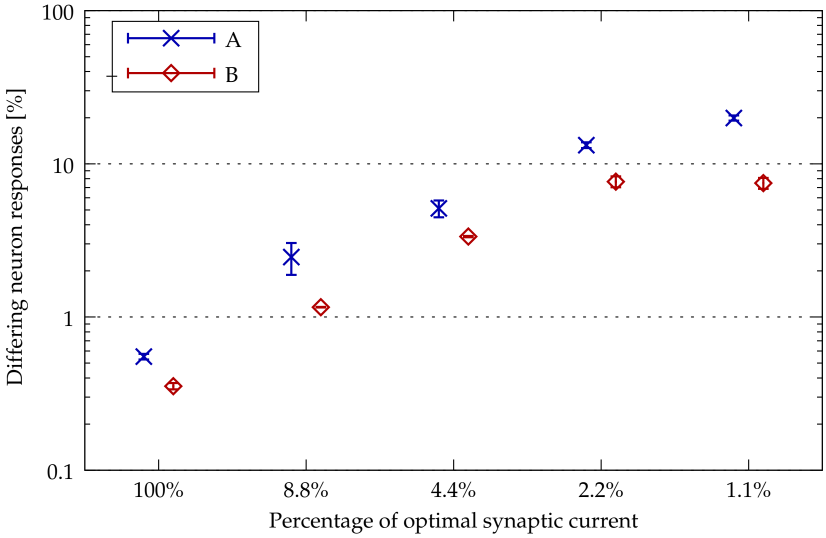


Figure 6.14: *Network noise with lowered synaptic currents. A: Percentage of neuron responses in layer C2 differing between software and hardware implementation (pre-computed weights). B: Percentage of neuron responses in layer C2 differing between two equal hardware runs (temporal noise).*

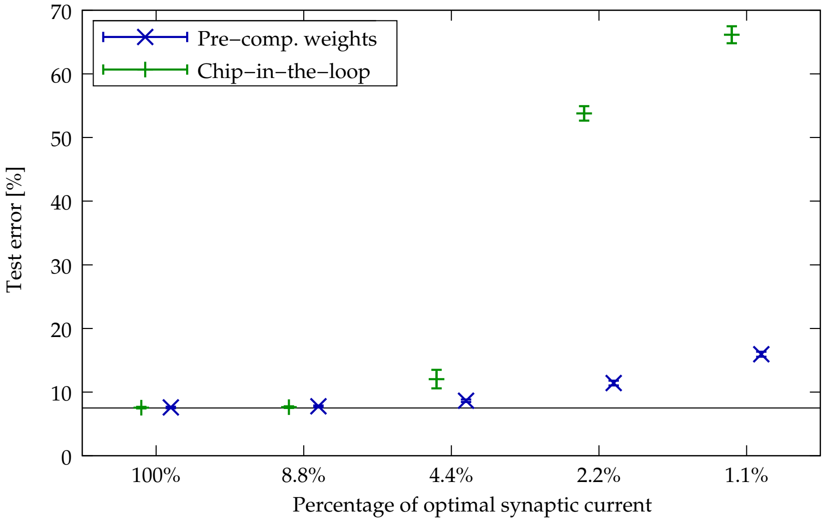


Figure 6.15: *Hardware degradation by lowered synaptic currents (cf., Table 6.2). Only the output layer is subject to degradation. The hidden layers are evaluated on the optimal hardware system. Shown is the test error of the MNIST problem for two training methods (pre-computed weights, chip-in-the-loop). Black line = software reference from Figure 6.11.*

quite well[7]. In optimal conditions (100% synaptic current), only about 0.3% of all outputs in layer C2 differ from the software calculation. This is particularly worth mentioning because this includes the accumulated errors of two subsequently computed hardware layers (S1 and S2).

Comparing the points labeled A with the points B, the fixed pattern noise is 1-2 times as strong as the temporal noise (corresponding to $A/B \approx 2.5 \pm 0.5$). From the fact that a significant fraction of the hardware errors is time-invariant, it could have been predicted that the chip-in-the-loop training would be able to compensate for at least part of the errors. This is in accordance with Figure 6.13.

Another, probably less relevant, conclusion is that a decrease of the synaptic current I_{syn} on does not seem to result in as much noise as decreasing the weights in software: The degradation levels 8.8% through 2.2% which differ only in weight scale produce monotonously increasing network noise. The degradation levels 2.2% and 1.1% which differ only by the setting of I_{syn}, produce approximately the same amount of temporal noise and are only little different in fixed-pattern noise. In the measured domain, the noise introduced by digital quantization seems to dominate.

Output Layer

In the above experiments, the degraded hardware was applied only to the hidden layers. Similar experiments are also conducted for the output layer. Now, the hidden layers are evaluated with the optimal hardware configuration using pre-computed weights. The output layers are evaluated with the hardware

[7] With the limitations stated under Discussion on page 112

degradation levels listed in Table 6.2, both with pre-computed weights (including chip calibration according to section 6.2.3) and with weights trained by the chip-in-the-loop method (section 5.1.2).

In Figure 6.15 we observe that both training methods yield classification performances equal to the software reference until decreasing the synaptic current to 8.8%. For continued degradation the results are unexpected: The chip-in-the-loop Perceptron learning rule, which was especially developed for coping with computation errors, performs in fact worse than loading pre-computed weights onto the chip. Looking at the performance of the single neurons in the output layer (no plot shown) reveals that with lower synaptic currents, some neurons still work almost perfectly while others fire only sparsely or do not fire at all, even if the programmed weight values indicate an active state. The reason for this behavior remains ultimately unclear. The measured data for the pre-computed weights shows that synapse configurations with much better performance do actually exist. They are just not found by the chip-in-the-loop algorithm. One possible explanation includes a detail of the chip's design (the so-called *dac offset*; see [54, 22]): Small programmed weights (positive or negative) can have a much larger absolute value on the chip. In particular, weight values close to zero cannot be written. Normally, this undesired behavior is prevented by a compensation routine, which might however not work correctly with too small reference currents. The function translating the programmed weight into the effective synaptic weight would then have a non-continuous step near zero which might impair the functioning of the training algorithm. Also, the effect of the mentioned malfunction would be similar to the clamp errors investigated in the software simulations, where weight values are randomly changed to a large positive or negative value. Such errors were found to have a comparably strong influence on the network operation (Figure 5.3).

Summary and Conclusions

This thesis demonstrates by means of an example application that massively parallel computing with analog hardware can constitute a feasible alternative to standard digital computers. In the particular test case studied, namely object classification using large-scale neural networks, the considered parallel hardware architecture has in fact advantages over the implementation on sequential, von Neumann-type machines: The parallel structure of the computing problem is reflected in a *physically existing* substrate. This is fundamentally different from *simulating* parallel processing in software. On the available prototype chip, more than 32,000 synaptic operations are performed truly simultaneously. All computing units are integrated in one small micro chip, consuming approximately two orders of magnitude less power than a single conventional PC.

The application is based on a large-scale convolutional neural network featuring a feed-forward multi-layer topology with a connectivity based on local receptive fields (Figure 3.1). Effectively, each network layer computes a set of convolutions on the previous layer with subsequent non-linear scaling. Thus, a hierarchical feature extraction pyramid is implemented which provides the basis of the recognition process (Figure 1.8).

The neuron model implemented by the considered hardware system features a threshold activation function, prohibiting the straight-forward application of gradient-based training methods which are commonly used to train convolutional networks. A gradient-free training method was developed, suited for both precomputing the weights in software and hardware-in-the-loop training. The network layers are trained separately, one after another. The training of the hidden layers is based on self-organization. Only linear classifiers in the output layer are trained in a supervised way. The developed training method was tested on two recognition problems (chapter 4). The classification rates obtained with the publicly available MNIST data set of hand-written digits are comparable to other state-of-the-art methods (Figure 4.2). The separation of the training into independent phases, one for each network layer, results in smaller search spaces within each sub-problem compared to training the entire network as a whole. Thus, the proposed method scales better with the network size than global training algorithms as for example the back-propagation algorithm (section 4.2.4). Scalability of the training method is an important issue when using parallel hardware that allows the realization of very large networks. The nature of the training approach (self-organization combined with linear classifiers) also prevents overfitting:

In the experiments, using larger networks does never decrease the generalization ability (section 4.2.2).

Analog computing does not offer the degree of precision provided by digital computers. It is a major concern of this thesis to assess how much this affects the usability of analog computers in practice. The studied neural network application was shown in software simulations to tolerate a wide range of computing errors, especially when the inaccuracies are incorporated in the training according to the developed chip-in-the-loop techniques (chapter 5, Figures 5.1–5.2). The exceptional robustness observed in the hidden network layers when adding random offsets to the synaptic weights is promoted by the chosen training approach (competitive learning) which tends to produce synaptic weights with extreme (positive or negative) values (Figure 5.4). Such networks are very stable under weight perturbations.

A prototype implementation of an analog array-based neural network architecture was available for experiments. The micro chip and the supporting hardware system are the result of previous research. They were not particularly designed for the presented application and, thus, certain restrictions are imposed on the experiments that could be performed (section 6.3).

However, it was proven that the used analog computing architecture actually provides more precision than needed for an optimal operation of the application (section 6.5.1). In fact, it was possible to operate the chip in a sub-optimal regime, implying a drastic reduction in power consumption, without sacrificing much of the recognition performance (Figure 6.13). This observation supports the claim that analog computing can be a robust alternative to digital technology. Furthermore, it can be concluded that in fact a very simple device featuring only very low computing resolution would be sufficient for the tested application (section 5.4). In a conceivable dedicated appliance, the low precision requirement could be traded off for a gain in synapse density, higher processing speed, or for further power economy. Small size and low power consumption are crucial prerequisites for mobile applications, as for example tools for personal assistance or intelligent systems in the automotive domain.

Not all the aspects of the investigated application have been implemented on the hardware system. The computing of the blurring layers (see Figure 3.1) and much of the data handling remain to be done in software (Figure 6.12). The network's output layer can in principle be evaluated on the analog hardware, but it must be severely pruned in order to fit on the chip (section 6.5.1). This, together with its low computation cost, makes a digital implementation of the output layer seem more efficient. It was not in the scope of this thesis to optimize the hardware system, but rather to show the general applicability of the underlying analog computing architecture. Given sufficient engineering effort, it seems to be straight forward to develop a hardware integration of all the algorithm parts, possibly not even requiring the overhead of a separate control PC. One idea for such a solution is given by the recently developed distributed system where each network chip is integrated with digital logic and memory in small independent modules (section 2.2.2). Moreover, for a real-life application, some of the known, but uncritical, limitations of the current prototype chip would have to be resolved, e.g., the size restriction (section 6.3.1) and the shortcomings in the neuron calibration (section 6.2.3).

A chip-in-the-loop version of the Perceptron learning rule, not requiring any model of the hardware variations, was proposed as part of the training method (section 5.1.2). It shows to produce good results while being able to implicitly compensate for computing errors. Complications were only observed when operating the hardware system under conditions far away from the optimal regime (Figure 6.15). This training method is not restricted to the special case of convolutional neural networks. Therefore, when training linear classifiers in general on the hardware system, the chip-in-the-loop Perceptron algorithm can provide a simple and fast alternative for other approaches which have been used previously for this purpose, e.g., genetic algorithms [22].

The developed methods include the training of a hierarchical feature extraction stage by self-organization, in particular, by applying a form of competitive learning (section 3.3.1). The weight update rule is quite local: only a hyper column of neurons within the same layer is inter-dependent. This eases a possible hardware integration. The good scaling property of the current hardware architecture would not be impaired while analog computing errors would only accumulate within a confined local domain. A Hebbian-style learning rule (without competition) was already successfully implemented on a different neuro chip developed in the work group [55]. Neural network chips including competitive learning could serve as general feature extraction processors, adapting themselves to changing input data in real-time.

Appendix

Complete measurements from section 5.2 (Figures 5.1–5.3).

Figure 5.1 (Noise)

σ of noise	Avg. test Error	Std.-dev	Error of mean	# measurements
Pre-computed weights (Hidden layers):				
0.02	0.01894	0.00381	0.00121	10
0.05	0.02226	0.00415	0.00131	10
0.1	0.02481	0.00612	0.00194	10
0.2	0.03307	0.01127	0.00356	10
0.5	0.06733	0.01596	0.00505	10
Pre-computed weights (Output layer):				
0.02	0.01713	0.00116	0.00037	10
0.05	0.01798	0.00094	0.00030	10
0.1	0.02027	0.00172	0.00054	10
0.2	0.03328	0.00436	0.00138	10
0.5	0.13512	0.02724	0.00862	10
Pre-computed weights (All layers):				
0.02	0.01888	0.00315	0.00100	10
0.05	0.02353	0.00569	0.00180	10
0.1	0.02852	0.00752	0.00238	10
0.2	0.05944	0.02073	0.00656	10
0.5	0.26550	0.04515	0.01428	10
Chip-in-the-loop (Hidden layers):				
0.02	0.01777	0.00109	0.00035	10
0.05	0.01754	0.00081	0.00026	10
0.1	0.01727	0.00120	0.00038	10
0.2	0.01813	0.00081	0.00026	10
0.5	0.01933	0.00133	0.00042	10
Chip-in-the-loop (Output layer):				
0.02	0.01684	0.00122	0.00038	10
0.05	0.01764	0.00114	0.00036	10
0.1	0.01735	0.00139	0.00044	10
0.2	0.01805	0.00098	0.00031	10
0.5	0.02517	0.00111	0.00035	10
Chip-in-the-loop (All layers):				
0.02	0.01752	0.00068	0.00021	10
0.05	0.01795	0.00123	0.00039	10
0.1	0.01777	0.00094	0.00030	10
0.2	0.01865	0.00068	0.00022	10
0.5	0.02882	0.00179	0.00056	10

Figure 5.2 (Delete)

Fraction of deleted synapses	Avg. test Error	Std.-dev	Error of mean	# measure- ments
Pre-computed weights (Hidden layers):				
0.02	0.03022	0.01338	0.00299	10
0.05	0.04446	0.02307	0.00516	10
0.1	0.08711	0.05298	0.01675	10
0.2	0.22527	0.10998	0.03478	10
0.5	0.60973	0.08030	0.02539	10
Pre-computed weights (Output layer):				
0.02	0.01690	0.00118	0.00037	10
0.05	0.01761	0.00060	0.00019	10
0.1	0.01807	0.00107	0.00034	10
0.2	0.01929	0.00124	0.00039	10
0.5	0.02534	0.00270	0.00085	10
Pre-computed weights (All layers):				
0.02	0.03460	0.01310	0.00414	10
0.05	0.04876	0.02500	0.00791	10
0.1	0.09920	0.06060	0.01916	10
0.2	0.23826	0.11998	0.03794	10
0.5	0.63167	0.07003	0.02214	10
Chip-in-the-loop (Hidden layers):				
0.02	0.01682	0.00134	0.00042	10
0.05	0.01769	0.00080	0.00025	10
0.1	0.01827	0.00093	0.00029	10
0.2	0.02213	0.00214	0.00068	10
0.5	0.05944	0.01816	0.00574	10
Chip-in-the-loop (Output layer):				
0.02	0.01745	0.00161	0.00051	10
0.05	0.01689	0.00078	0.00025	10
0.1	0.01758	0.00073	0.00023	10
0.2	0.01757	0.00121	0.00038	10
0.5	0.01842	0.00093	0.00030	10
Chip-in-the-loop (All layers):				
0.02	0.01696	0.00081	0.00026	10
0.05	0.01817	0.00102	0.00032	10
0.1	0.01987	0.00157	0.00050	10
0.2	0.02276	0.00258	0.00082	10
0.5	0.12061	0.04418	0.01397	10

Figure 5.3 (Clamp)

Fraction of clamped synapses	Avg. test Error	Std.-dev	Error of mean	# measure- ments
Pre-computed weights (Hidden layers):				
0.02	0.02962	0.00647	0.00205	10
0.05	0.03848	0.00985	0.00311	10
0.1	0.05979	0.02059	0.00651	10
0.2	0.14525	0.05224	0.01652	10
0.5	0.75662	0.08861	0.02802	10
Pre-computed weights (Output layer):				
0.02	0.02602	0.00488	0.00154	10
0.05	0.04420	0.01527	0.00483	10
0.1	0.08252	0.02841	0.00898	10
0.2	0.18443	0.05155	0.01630	10
0.5	0.45756	0.07441	0.02353	10
Pre-computed weights (All layers):				
0.02	0.05222	0.01930	0.00610	10
0.05	0.08186	0.02615	0.00827	10
0.1	0.16599	0.04463	0.01411	10
0.2	0.35822	0.09062	0.02866	10
0.5	0.88936	0.03068	0.00970	10
Chip-in-the-loop (Hidden layers):				
0.02	0.01713	0.00131	0.00041	10
0.05	0.01795	0.00105	0.00033	10
0.1	0.01905	0.00146	0.00046	10
0.2	0.02067	0.00206	0.00065	10
0.5	0.04436	0.02492	0.00788	10
Chip-in-the-loop (Output layer):				
0.02	0.01777	0.00096	0.00030	10
0.05	0.01870	0.00065	0.00020	10
0.1	0.02103	0.00171	0.00054	10
0.2	0.03134	0.00186	0.00059	10
0.5	0.08398	0.00551	0.00174	10
Chip-in-the-loop (All layers):				
0.02	0.01861	0.00096	0.00030	10
0.05	0.02018	0.00096	0.00030	10
0.1	0.02402	0.00328	0.00104	10
0.2	0.04605	0.00716	0.00226	10
0.5	0.27877	0.08590	0.02716	10

Bibliography

[1] V. Beiu, J. M. Quintana, M. J. Avedillo: VLSI implementations of threshold logic—a comprehensive survey. IEEE Transactions on Neural Networks, 14(5) (2003)

[2] R.E. Bellman: Adaptive control processes. Princeton University Press, Princeton, NJ (1961)

[3] C. M. Bishop: Neural networks for pattern recognition. Oxford University Press Inc., New York (1995)

[4] B. E. Boser, E. Säckinger, J. Bromley, Y. LeCun, L. D. Jackel: An analog neural network processor with programmable topology. IEEE Journal of Solid-State Circuits, 26(12), 2017–2025 (1991)

[5] E. Cosatto, H.P. Graf: NET32K high speed image understanding system. In Proceedings of the Fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems, IEEE Computer Society Press, 413-421 (1984)

[6] P. Dayan, L.F. Abbott: Theoretical neuroscience: computational and mathematical modeling of neural systems, MIT Press (2001)

[7] J. Fieres, A. Grubl, S. Philipp, K. Meier, J. Schemmel, F. Schürmann: A platform for parallel operation of VLSI neural networks. Conference on Brain Inspired Cognitive Systems (BICS 2004), Stirling, Scotland (2004)

[8] J. Fieres, J. Schemmel, K. Meier: Training convolutional neural networks of threshold neurons suited for low-power hardware implementation. Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN 2006), 21–28, IEEE Press (2006)

[9] J. Fieres, J. Schemmel, K. Meier: A convolutional neural network tolerant of synaptic faults for low-power analog hardware. Proceedings of 2nd IAPR International Workshop on Artificial Neural Networks in Pattern Recognition (ANNPR 2006), Springer Lecture Notes in Artificial Intelligence 4087, 122–132 (2006)

[10] K. Fukushima, S. Miyake: Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. Pattern Recognition, 15(6), 455–469 (1982).

[11] K. Fukushima: Neocognitron: A hierarchical neural network capable of visual pattern recognition. Neural Networks 1, 119–130 (1988)

[12] Y. Frégnac, D. Shulz: Models of synaptic plasticity and cellular analogs of learning in the developing and adult vertebrate visual cortex. In: Advances in neural and behavioral development (Eds: V.A. Casgrande and P.G. Shinkman), 4, 148–235, Ablex Publ. Corp., Norwood, New Jersey (1994)

[13] A. Grübl: Eine FPGA-basierte Plattform für Neuronale Netze. Diploma Thesis (German), Ruprecht-Karls-University, Heidelberg (2003)

[14] J.-L. Gailly, M. Adler: The gzip compression utility. On: The gzip home page, http://www.gzip.org.

[15] E. Gamme, R. Helm, R. Johnson, J. Vlissides: Design patterns: Elements of reusable object-oriented software. Addison-Wesley, Reading, Massachusetts (1995)

[16] H.P. Graf, R. Janow, D. Henderson, R. Lee: Reconfigurable neural net chip with 32K connections. Proceedings Int. Conference of Information Processing Systems (NIPS1992), 1032–1038 (1992)

[17] S. Haykin: Neural networks: a comprehensive foundation. 2nd ed., Prentice Hall, New Jersey (1999)

[18] D. Hubel, T. Wiesel: Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. J. Physiology 160, 106–154 (1962)

[19] D. Hubel, T. Wiesel: Receptive fields and functional architecture in two non-striate visual areas (18 and 19) of the cat. J. Neurophysiol. 28, 229–289 (1965)

[20] S. G. Hohmann, J. Schemmel, F. Schürmann, K. Meier: Exploring the parameter space of a genetic algorithm for training an analog neural network. Proceedings of the Genetic and Evolutionary Computation Conference (GECKO 2002), 375–382, Morgan Kaufmann Publishers, San Francisco (2002)

[21] S. G. Hohmann, J. Fieres, K. Meier, J. Schemmel, T. Schmitz, F. Schürmann: Training fast mixed-signal neural networks for data classification. Proceedings of the 2004 International Joint Conference on Neural Networks (IJCNN 2004), 2647–2652, IEEE Press (2004)

[22] S. G. Hohmann: Stepwise Evolutionary training strategies for hardware neural networks. PhD thesis, Ruprecht-Karls-University, Heidelberg (2005),
http://www.kip.uni-heidelberg.de/vision/publications

[23] B. Jähne: Digital image processing. 6th ed., Springer Verlag Berlin, Heidelberg, New York (2005)

[24] J.-S. R. Jang, C.T. Sun, E. Mizutani: Neuro-fuzzy and soft computing. Prentice-Hall (1997)

[25] E. van Keulen, S. Colak, H. Withagen, H. Hegt: Neural network hardware performance criteria. Proceedings of the IEEE International Conference on Neural Networks 1994, 1885–1888 (1994)

[26] A. H. Kramer: Array-based analog computation. IEEE Micro 16(5), 20-29 (1996)

[27] E. Kussul, T. Baidyk, D. Wunhsch II, O. Makeyev, A. Martin: Image recognition systems based on random local descriptors. Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN 2006), 4722–4727, IEEE Press (2006)

[28] E. Kussul, Lab of Micromechanics and Mechatronics, Universidad Nacional Autónoma De México: Oral conversation.

[29] S. Lawrence, C.L. Giles, A.C. Tsoi, A.D. Back: Face recognition: a convolutional neural network approach. Transactions on Neural Networks 8(1) 98–113 (1997)

[30] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, L.D. Jackel: Backpropagation applied to handwritten zip code recognition. Neural Computation 1(4), 541–551 (1989)

[31] Y. LeCun, L. D. Jackel, B. Boser, J.S. Denker, H. P. Graf, I. Guyon, D. Henderson, R.E. Howard, W. Hubbard: Handwritten digit recognition: Applications of neural net chips and automatic learning. IEEE Communications Magazine, November 1989, 41–46 (1989)

[32] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner: Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324 (1998)

[33] T. Lehmann: Hardware learning in analogue VLSI neural networks. PhD Thesis, Techincal University of Denmark, Lyngby, Denmark (1994)

[34] R. Linsker: From basic network principles to neural architecture. (Series of 3 papers) Proc. Natl. Sci. USA 83, 7508–7512 (1983)

[35] D. R. Lovell, T. Downs, A. C. Tsoi: An evaluation of the Neocognitron. IEEE Transactions on Neural Networks 8(5), 1098–1105 (1997)

[36] W. Maass, T. Natschläger, H. Markram: Real-time computing without stable states: A framework for neural computation based on perturbation. Neural Computation 14(11) 2531–2560 (2002)

[37] P. Masa, P. Heim, E. Franzi et al.: 10 mW CMOS retina and classifier for handheld, 1000 images/s optical character recognition system. Proceedings of the IEEE International Solid-State Circuit Conference, 202– (1999), see also
`http://www.csem.ch/detailed/pdf/m_111_Handheld OCR for eBanking.pdf`

[38] M. R. J. McQuoid: Neural ensembles: Simultaneous recognition of multiple 2-D visual objects, Neural Networks 6, 907–917 (1993)

[39] T.M. Mitchell: Machine learning. International ed., McGraw-Hill Book Co, Singapore (1997)

[40] Y. LeCun: The MNIST database of handwritten digits.
`http://yann.lecun.com/exdb/mnist`

[41] C. Neubauer: Evaluation of convolutional neural networks for visual recognition. Transactions on Neural Networks 9(4), 685–696 (1998)

[42] J. von Neumann: First draft of a report on the EDVAC. manuscript, Moore School of Electrical Engeneering Library, University of Pennsylvania (1945). Transscript in: M. D. Godfrey: Introduction to "The first draft report on the EDVAC" by John von Neumann. IEEE Annals of the History of Computing 15(4), 27–75 (1993)

[43] D. Niedenzu: Aufbau eines binären Neocognitrons. Diploma Thesis (German), Ruprecht-Karls-University, Heidelberg (2003)

[44] E. Oja: Principle components, minor components, and linear neural networks. Neural Networks 5, 927–936 (1992)

[45] M.W. Oram, D.I. Perret: Modeling visual recognition from neurobiological constraints. Neural Networks 7, 945–972 (1994)

[46] W.H. Press , B.P. Flannery, S.A. Teukolsky, W.T Vetterling: Numerical recipes in C: The art of scientific computing. Cambridge University Press, Online ed. (1992)
`http://www.nr.com`

[47] Trolltech ASA, Norway: Qt - Cross-platform C++ development. http://www.trolltech.com/products/qt

[48] M. Riesenhuber, T. Poggio: Hierarchical models of object recognition in cortex. Nature Neuroscience 2, 1019–1025 (1999)

[49] R. Rojas: Theorie der Neuronalen Netze. Springer Verlag (1993)

[50] D. E. Rumelhart, D. Zipser: Feature discovery by competitive learning. Cognitive Science, 9 75–112 (1985)

[51] D. E. Rumelhart, G. E. Hinton, R. J. Williams Learning representations by back-propagating errors. Nature 323, 533–536 (1986)

[52] E. Säckinger, B. E. Boser, J. Bromley, Y. LeCun, L. Jackel: Application of the ANNA neural network chip to high-speed character recognition. IEEE Transactions on Neural Networks 3(3), 498–505 (1992)

[53] S. Satyanarayana, Y. P. Tsividis, H. P. Graf: A reconfigurable VLSI neural network. IEEE Journal of Solid State Circuits 27(1), 67–81 (1992)

[54] J. Schemmel, S. Hohmann, K. Meier, F. Schurmann: A mixed-mode analog neural network using current-steering synapses. Analog Integrated Circuits and Signal Processing 38, 233–244 (2004)

[55] J. Schemmel, A. Gruebl, K. Meier, E. Mueller: Implementing Synaptic Plasticity in a VLSI Spiking Neural Network Model. Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN 2006), 1–6 (2006)

[56] J. Schemmel: Oral communication

[57] T. Schmitz: Evolution in Hardware – Eine Experimentierplattform zum parallelen Training analoger neuronaler Netzwerke. PhD thesis (German), Ruprecht-Karls-University, Heidelberg (2005), `http://www.kip.uni-heidelberg.de/vision/publications`

[58] F. Schürmann, S.G. Hohmann, K. Meier, J. Schemmel: Interfacing binary networks to multi-valued signals. Supplementary proceedings of ICANN/ICONIP 2003, IEEE Press, 430–433 (2004)

[59] F. Schürmann: Exploring liquid computing in a hardware adaption: construction and operation of a neural network experiment. PhD thesis, Ruprecht-Karls University, Heidelberg (2005)

[60] P.Y. Simard, D. Steinkraus, J.C. Platt: Best practices for convolutional neural networks applied to visual document analysis. Intl. Conf. Document Analysis and Recognition, 958–962 (2003)

[61] P.Y. Simard, Microsoft Research: Email conversation.

[62] K. Tanaka: Inferotemporal cortex and object vision. Ann. Rev. Neuroscience 19 109–139 (1996)

[63] L. Tao, M. Shelley, D. McLaughlin, R. Shapley: An egalitarian network model for the emergence of simple and complex cells in visual cortex. PNAS 101, 366–371 (2004)

[64] J. Teichert, R. Malaka: A component association architecture for image understanding. Proceedings of the 2002 International Joint Conference on Neural Networks (IJCNN 2002), Lecture Notes in Computer Science 2415, 125–130 (2002)

[65] S. Ullmann, S. Soloviev: Computation of pattern invariance in brain-like structures. Neural Networks 12, 1021–1036 (1999)

[66] M. Valle: Analog VLSI implementations of neural network with supervised on-chip learning. Analog Integrated Circuits and Signal Processing 33, 263–287 (2002)

[67] J. Weng, N. Ahuja, T. S. Huang: Learning recognition and segmentation using the Cresceptron. International Journal of Computer Vision 25(2), 109–143 (1997)

[68] A. Zell: Simulation neuronaler Netze. 1st Ed. Addison-Wesley Germany GmbH (1994)

Acknowledgments (Danksagungen)

Thanks for funding...

This work was funded in part by the European Union, grant numbers IST-2001-34712 (SenseMaker) and IST-2004-2.3.4.2 (FACETS). The author was temporarily supported by a scholarship of the Landesgraduiertenförderung, Baden-Württemberg. Funds were also received by the state of Baden-Württemberg and, last but not least, by the author's parents. Thank is expressed to all the donors.

...for scientific support...

Ein Werk wie das Vorliegende lässt sich nur schwerlich als Einzelperson vollbringen. Den Personen, die maßgeblich oder teilweise am Gelingen dieser Arbeit beteiligt waren, sei herzlich gedankt:

- Ich danke Herrn Prof. Meier für die freundliche Aufnahme in seine Electronic Vision(s)-Gruppe. Er hat mir die Möglichkeit gegeben, europaweit an einem internationalen Projekt mitzuarbeiten, hat stets für den notwendigen Rahmen gesorgt (s.o.) und war immer erreichbar, wenn Ratschläge vonnöten waren.

- Ich bedanke mich bei Herrn Prof. Jähne, der freundlicherweise das Zweitgutachten dieser Arbeit übernommen hat.

- Großen Dank schulde ich Dr. Johannes Schemmel, einem der Mitbegünder der Arbeitsgruppe und dem Entwickler des analogen neuronalen Netzwerkchips, der die Grundlage und die Motivation für die vorligende Arbeit darstellt. Johannes fällt ein großer Teil der inhaltlichen Betreuung zu. Seine visionären Ideen in allen Bereichen der Technik haben regelmäßig zu inspririerenden Gesprächen geführt, sein erbarmungsloser Scharfsinn hat viele Aussagen in dieser Arbeit auf den eigentlichen Punkt gebracht, und seine Geduld, einem Elektronik-Laien wie mir immer wieder die ausgeklügelte Funktionsweise des Chips nahezubringen, ist bewundernswert.

- Meinem (ehemaligen) Kollegen, Schreibtischnachbarn, Freund, Frontman und Trauzeugen Dr. Steffen G. Hohmann gebührt an dieser Stelle besonderer Dank. Seine Anwesenheit in der Arbeitsgruppe war nicht zuletzt einer der Faktoren, die die entgültige Entscheidung, dieses Doktorarbeitsthema zu wählen, mitbestimmt haben. Die fruchtbare Zusammenarbeit im Handwerk des Software-Engineerings war eine angenehme und lehrreiche Erfahrung. Insbesondere möchte ich ihm für die Korrektur und Besprechung des Großteils dieser Arbeit danken.

- Bei den Kollegen aus dem Hardwarezimmer, besonders Stefan Philipp, Tillmann Schmitz und Andreas Grübl, möchte ich mich für Unterstützung in Hardwarefragen und die ständige Verbesserung der Kommunikationsschnittstellen bedanken. Einiges an Arbeit wäre ihnen ohne meine Wünsche und Probleme sicher erspart geblieben.

- Besondere Erwähnung verdienen die "Soft-Boyz", mit denen ich das Softwarezimmer teile: Daniel Brüderle, Lars Büsing und Eilif Mueller. Ihnen danke ich für ein angenehmes Betriebsklima, tägliche Exkursionen zum Unishop und für die Reisebetreuung in Kanada.

- Ich danke allen Mitgliedern der Arbeitsgruppe für eine freundliche Atmosphäre, Hilfsbereitschaft, Kaffee-Sekt-und-Kuchen zu jedem Anlass, erheiternde Gespräche zwischen Tür und Angel, und die gegenseitige Bereicherung aller möglichen Festlichkeiten durch gutgelauntes Erscheinen.

- Vielen anderen Mitarbeitern des Kirchhoff Instituts für Physik, die mich unterstützt haben, möchte ich meinen Dank aussprechen: Robert Weis für die unermüdliche Aufrechterhaltung der IT-Systeme, Markus Dorn für die Tag&Nacht-Adminstration der Fourofeight, Ralf Achenbach für die psychologische Vermittlung zwischen mir und seinem Klimaschrank. Ich möchte mich auch bei den Mitarbeitern in der Verwaltung für Hilfe im bürokratischen Alltag bedanken, insbesondere bei Claudia Brüser, Harald Jacobsen und Oscar Martin-Almendral.

- Diese Arbeit wäre unmöglich gewesen, wenn nicht unzählige Entwickler ihre Freizeit darauf verwendeten, freie und nützliche Software zu erstellen. Anstatt die mir größtenteils unbekannten Namen der Programmierer zähle hier ich die von mir meistverwendeten Werkzeuge auf: Linux, KDE, g++, bash, emacs, grep, gawk, sed, gnuplot, xfig, latex und gzip. Danke für diese exzellenten Programme.

...and for all the rest

Eine Doktorarbeit ist in erster Linie assoziiert mit der Erstellung einer wissenschaftlichen Arbeit. Darüber hinaus bezeichnet der Begriff aber, schon aufgrund der zeitlichen Ausdehung einer Dissertation, einen kompletten Abschnitt im Leben eines Menschen. Denjenigen, die mir in dieser Zeit persönlich wertvolle Weggefährten waren (und hoffentlich noch eine zeitlang bleiben werden) möchte ich hier meinen Dank aussprechen, insbesondere:

- der Band "Fake-It", deren Schlagzeuger ich sein durfte, mit allen ihren Mitgliedern: Ani, Franky, Fred, Mad, Merd und Stevo für die gemeinsame Entwicklung von einer Gruppe Musik-Laien und Halbmusikern zur rockigsten, groovigsten, swingigsten und eingeschworensten Party-Kultband in der ich je gespielt habe. Die Proben waren ein Höhepunkt der Woche, von den Auftritten ganz zu schweigen.

- dem Mensaclub und seinen (aktuellen und gegenwärtigen, regelmäßigen und sporadischen) Mitgliedern, für die mittägliche Konstante im Arbeitsalltag bei gutem wie bei schlechtem Essen, für angeregte Gespräche oder entspanntes Schweigen, für außermensarische Kontakte – und das alles über Jahre hinweg; besonders aber: Annika, Florian B., Florian F., Kristin, Kristine, Martin, Nadine, Tom, Verena und Wieland.

- den wertvollen Freunden, die mir über die Studienzeit hinweg bis jetzt treu geblieben sind, und die noch nicht anderweitig erwähnt wurden.

- meinen Eltern, die mich bis heute uneingeschränkt in allen Lebenslagen und -fragen unterstützen.

- Das Beste soll man sich für den Schluß aufsparen. Deshalb gebührt diese Stelle keiner anderen als Ana Fieres, geb. Kovatcheva, die ich während meiner Promotionszeit erst richtig schätzen gelernt habe, und die ich, als einzig sinnvolle Konsequenz, letztendlich geheiratet habe. Allein ihretwegen wäre meine Promotion eine lohnende Sache gewesen.