Brain-Inspired Models and Systems for Distributed Computation

Jonathan Binas
BRAIN-INSPIRED MODELS AND SYSTEMS FOR DISTRIBUTED COMPUTATION

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH Zurich

(Dr. sc. ETH Zurich)

presented by

Jonathan J. M. Binas

MSc. ETH Zurich

born on Oct. 15, 1986

citizen of Germany

accepted on the recommendation of

Prof. Dr. Giacomo Indiveri
Prof. Dr. Hans-Andrea Loeliger
Dr. Matthew Cook

2017
Acknowledgments

I wish to thank my supervisors, Giacomo Indiveri and Michael Pfeiffer, for their continuous support and the freedom I was given throughout my PhD, and Hans-Andrea Loeliger and Matthew Cook for reviewing this thesis as members of the examination committee. I am grateful for all the discussions I had at the Institute of Neuroinformatics and the time I shared both with colleagues and friends. I wish to thank all the inspiring people I had the pleasure to work with, including members of the NCS group, members of the CO2 group, Shih-Chii Liu, Tobi Delbruck, Danny Neil, and Ning Qiao.

I thank my parents for always supporting me. I thank Anna for everything.
Disclaimer

## Contents

1 Introduction  

I Structural constraints in neural systems  

2 Learning in locally connected artificial neural networks  
   2.1 Model and training details  
   2.2 Improved classification accuracy in sparse networks  
   2.3 Discussion  

3 Learning optimized automata in recurrent neural networks  
   3.1 Model definition  
   3.2 Theoretical considerations  
   3.3 Simulation results  
   3.4 Optimized solutions due to local structure  
   3.5 Learning to solve behavioral tasks  
   3.6 Discussion  

II Spiking neurons for efficient inference and learning  

4 A simple neuron model for event-based neural networks  
   4.1 Multiplication-free networks  
   4.2 Lossless event-based implementation  
   4.3 Numerical results  
   4.4 Discussion  

5 Event-based training of multilayer counter networks  
   5.1 Hierarchical counter network descends energy functional  
   5.2 Contrastive update rule leads to event-based learning
III Computing with imprecise and noisy substrates

6 Noisy analog devices for probabilistic processing
   6.1 Solving CSPs with spiking neural networks
   6.2 Stochastic dynamics in an analog VLSI neural network
   6.3 Experimental results
   6.4 Discussion

7 Precise computation on imprecise analog electronics
   7.1 Training with heterogeneous transfer functions
   7.2 Analog circuit implementation
   7.3 Accurate inference on mismatched hardware
   7.4 Handwritten and spoken digit classification
   7.5 VLSI implementation
   7.6 Discussion
   7.7 Implementation details

Bibliography
Abstract

Not only do biological nervous systems clearly outshine modern computing technology in terms of intelligence, but also in terms of energy requirements, speed, and compactness. Such capabilities do not seem obvious given that brain cells transmit signals at average rates of as low as 1 Hz, are noisy and imprecise, and long-range connectivity is very sparse. Thus, the governing design principles of brains must be such that high performance processing is achieved despite noise and imprecision inherent to the computing elements, while power and space requirements are minimized. Today's computing technology is limited by constraints similar to those faced by biological systems, namely, restrictions in the number of transistors that can be accommodated per unit area, the resources and power required for communication between those transistors, and the increasing impact of noise as device dimensions shrink. Therefore, it is likely that modern information technology would benefit enormously from exploiting certain design principles loosely based on the ones found in biological systems.

The purpose of this thesis is to investigate key characteristics of biological nervous systems and suggest ways of incorporating these principles in artificial systems. The presented work is structured into three core categories, corresponding different aspects that are central to neural information processing. Specifically, aspects concerning the connectivity structure (topology), the style of communication and processing, and the underlying physical substrate are investigated, and are divided into parts I-III as follows:

I computational roles of structural constraints in neural and distributed processing systems,

II the event-based processing and communication style of biological neurons,

III exploiting and tolerating noise and imprecision in the underlying physical substrate implementing the computation.

It is shown that by taking into account these principles, or by actively enforcing them, improvements can be achieved in artificial systems in terms of accuracy, speed, power, or space requirements. In part I, it is demonstrated how a non-trivial network topology, featuring predominantly local connectivity, can
be beneficial in tasks involving learning in neural networks. Part II proposes simple models for event-based processing, which can lead to speed-ups and increased efficiency in artificial neural network implementations. Finally, part III discusses implications of noise and mismatch in modern transistor technology and how precise computation can be achieved regardless of such effects or how they can even be helpful in implementing stochastic processing systems. Models and example architectures are proposed based on the discussed principles, and tested experimentally.
Zusammenfassung

Biologische Nervensysteme sind modernen elektronischen Datenverarbeitungssystemen klar überlegen, nicht nur hinsichtlich ihrer Intelligenz, sondern vor allem auch, was den Energieverbrauch, die Geschwindigkeit und die Grösse betrifft. In Anbetracht der Tatsache, dass Gehirnzellen Signale mit vergleichsweise niedrigen Raten von ungefähr 1 Hz übertragen und dazu von Rauschen und anatomischen Ungenauigkeiten stark beeinträchtigt sind, wirken die kognitiven Leistungen solcher Systeme geradezu erstaunlich. Offenbar sind Gehirne so konstruiert, dass eine zuverlässige Datenverarbeitung trotz Störungen, Rauschen und Ungenauigkeiten gewährleistet ist und gleichzeitig die Leistungsaufnahme, sowie die räumliche Ausdehnung minimiert werden. Tatsächlich sind moderne Datenverarbeitungssysteme von ähnlichen Einschränkungen betroffen wie Gehirne, da die mögliche Anzahl von Transistoren sowie die zugeführte Energie pro Fläche beschränkt ist und die Störanfälligkeit mit verringriger Bauteilgröße zunimmt. Es ist daher anzunehmen, dass moderne Datenverarbeitungstechnologien von bestimmten biologisch inspirierten Gestaltungsprinzipien profitieren könnten, wenn diese auf die richtige Art und Weise angewandt werden.

Der Zweck dieser Arbeit ist, eine Reihe grundlegender Eigenschaften biologischer Nervensysteme zu untersuchen und Möglichkeiten zur Anwendung dieser Eigenschaften zur Konstruktion künstlicher Systeme aufzuzeigen. Die Arbeit ist in drei Teile gegliedert, wobei sich jeder mit einem anderen fundamentalen Aspekt neuronaler Informationsverarbeitung befasst. Insbesondere werden die Struktur, die Art der Kommunikation und Datenverarbeitung, sowie das zugrundeliegende physische Substrat im Rahmen der Teile I-III betrachtet:

I die Rolle struktureller Einschränkungen in neuronalen Systemen,

II die ereignisbasierte Informationsübertragung und -verarbeitung in biologischen Neuronen,

III Auswirkungen von Störungen im physischen Substrat und daraus folgende Konsequenzen für die Implementierung eines robusten Systems.

Im Detail wird gezeigt, wie die Berücksichtigung solcher Prinzipien in künstlichen Systemen zu erheblichen Vorteilen bezüglich deren Genauigkeit, Geschwindigkeit, Energieverbrauch und räumlicher Ausdehnung führen kann. In
Chapter 1

Introduction

The development of the human brain is perhaps the most remarkable outcome of evolution, having led to a state where this outcome, to a very significant extent, has gained control over the ongoing evolutionary process it emerged from. Having evolved only very recently on timescales of terrestrial life, the human brain can be considered the most advanced, or “state-of-the-art” biological information processing system. However, neither does this mean that it is perfect, nor that it is unsurpassed in every aspect: there are countless examples of non-human nervous systems which clearly are superior to their anthropoid counterpart in terms of speed and accuracy in specific disciplines.

The astounding capabilities of biological nervous systems have sparked interest in constructing artificial replicas thereof, or at least artificial systems which achieve similar performance levels in at least some very specific areas. First ideas concerning such systems were influenced by fundamental physiological and anatomical findings leading to the foundations of neuroscience, such as the fact that electricity was involved in the transmission of signals (Galvani & Aldini, 1792), or the organization of cortex into functional areas (Broca, 1863). With Ramón y Cajal’s discovery (Ramón y Cajal, 1888) that brains are composed of individual cells (later termed neurons), rather than a single, continuous piece of tissue, it became imaginable that an artificial intelligent system might be created by connecting large numbers of simple elementary units to form a larger, network-like structure. The idea of a possible artificial intelligence was fueled by Turing’s theory on computability (Turing, 1937) and the “Church-Turing thesis,” implying that a universal computing machine could be constructed based on simple mechanisms operating on symbols like ones and zeros. In fact, McCul-
loch and Pitts could show that a machine, universal in the sense of Turing, can be implemented by a network of simple processing units which, in a highly abstracted form, resemble the function of biological neurons (McCulloch & Pitts, 1943). Spurred by the early success of research along similar lines (e.g. Minsky, 1956; Newell et al., 1959; Rosenblatt, 1958; Shannon, 1950), the field of artificial intelligence took off rapidly, however, came to a halt equally quickly when promises made based on earlier results could not be met.

In the 1980s, the field experienced a temporary revival in the context of expert systems – entirely hard-coded systems, assisting experts in making decisions in very specific areas. Another low was reached when these systems hit the limits of their capabilities and technical issues related to scaling and maintenance emerged. It was not until the late 1990s that research in artificial intelligence took off a third time, to some extent due to the greater availability of computing power, and the development of purpose-built computing hardware, but also as a result of rigorous mathematical analysis of the properties and problems of previous approaches. A milestone of this period was the victory of IBM’s Deep Blue machine over the then World Chess Champion Garry Kasparov (Hsu, 1999; Newborn, 1997).

In the decade to follow, neural network-based systems progressively stood out over others, and gradually became the method of choice for most machine intelligence applications. Previous approaches had often been based on some form of propositional logic, requiring detailed manual design of the system and a lengthy configuration process. On the other hand, artificial neural networks, loosely inspired by the network structure and the elementary computing units of biological nervous systems, would allow to learn their function through the presentation of example data, rather than hard-coding it, whereby the operation of such systems could be described in a probabilistic framework. While the underlying concepts were not new at that time, with the foundations of neural network research going back to McCulloch and Pitts, it took until the 21st century to establish computing hardware powerful enough, and to realize that a substantial amount of training data was required to learn reasonably well performing models. The resulting systems soon outperformed humans in certain tasks – first in more natural ones, such as image recognition (He et al., 2015), later in more abstract ones, such as the game of Go (Silver et al., 2016).

The artificial neural networks enabling these successes, in their simplest form, consist of a number of stacked layers of neurons, with neurons of adjacent layers linked through directed, weighted connections. The neurons themselves
sum up their inputs, i.e. the weighted outputs of neurons of previous layers, and apply a non-linearity to this sum. A network is optimized to implement a certain function by altering the weight parameters. This is typically done through a stochastic gradient descent procedure, whereby chunks of input data are presented to the network, an output error is computed, and the weights are updated such that the error is lowered. The layered network structure and the differentiability of the neuron outputs with respect to individual weight parameters allow for layer-wise computation of the updates (Rumelhart et al., 1986a), making optimization of systems described by millions of parameters feasible in the first place. Similarly, recurrent neural networks, where the layered structure is absent, can be trained for tasks involving dynamic inputs or outputs. To overcome stability issues during training, these networks are typically based on some form of memory cells (Hochreiter & Schmidhuber, 1997), rather than simple neurons. Techniques for training artificial neural networks are now commonly referred to as “deep learning” (LeCun, Bengio & Hinton, 2015), where depth expresses the substantial number of stacked layers these networks typically consist of.

Despite impressive results achieved with state-of-the-art deep learning methods, biological nervous systems clearly outshine their artificial counterparts in many aspects, such as flexibility and learning speed. A striking difference is in the efficiency of both systems, with the power consumption of biological brains being orders of magnitude lower than that of current artificial neural networks. At the same time, efficiency is one of the main limiting factors of modern deep learning, a reason for the success of which was the clever employment of graphics processing units (GPUs) to parallelize the necessary computation to some extent (Scherer et al., 2010). In fact, the contemporary von Neumann computing architecture, characterized by a central processing unit and a separate block of memory, seems rather incompatible with the massively parallel and distributed nature of nervous systems, where processing and memory share the same physical substrate.

It is imaginable that by developing both models and hardware which better reflect the operating principles of biological nervous systems, it will be possible to create even more powerful artificial information processing systems. The purpose of this thesis is to do so by investigating key principles found in biological nervous systems and their computational role, and applying them in artificial systems. In particular, three key aspects are investigated in parts I-III of this thesis, namely the potential computational role of structural constraints in the
network topology (part I), the event-based communication and processing style (part II), and the effects of mismatch and noise in the physical substrate (part III). The presented topics are meant to contribute to the development of novel distributed, asynchronous computing architectures, potentially based on novel physical substrates to accommodate massively parallel networks of processing units.

Individual chapters are self-contained, each describing a different phenomenon, model, or implementation, and include introductions covering the necessary basics. As an overview, the main ideas of each part are briefly outlined below.

**Part I: Structural constraints.** How biological neurons can be connected is crucially constrained by the strictly limited space and power resources available to brains. In fact, biological nervous systems show a high degree of short-range connections, with long-range connections occurring only sparsely. This is in stark contrast to most computer models, where any connection is equally possible. What first seems like a limitation, might be advantageous during learning in the sense that an imposed structure reduces the dimensionality of the parameter space, and even might serve as a prior that initializes the system closer to the target state in the first place. In this part, it is shown that networks featuring predominantly local interactions indeed lead to improved results compared to fully connected ones.

**Part II: Processing style.** The style of processing of biological neural networks is fundamentally different from that of their digital counterparts. Neurons communicate through digital-like pulses, or “spikes”, and might code non-binary quantities in terms of rates, counts, or temporally, in terms of delays, for example. In this part, it is shown how modern digital implementations can benefit from such an asynchronous, event-based processing style, and simple models are proposed for efficient event-based learning and inference.

**Part III: Physical substrate.** Unlike common digital electronics, biological tissue is a noisy and imprecise computing substrate. Nevertheless, brains are able to make precise decisions. Progressively shrinking length scales of microelectronics, and novel physical substrates for implementing them, inevitably lead to similar effects. It is therefore desirable to establish models of computation based on such imperfect hardware. In this part, it is shown how noisy analog
electronics can be used for probabilistic processing and how precise processing can be achieved with mismatched analog circuits.
PART I

Structural constraints in neural systems
Introduction

Brains and modern microelectronics alike, are faced with similar restrictions in terms of energy and space. With a substantial fraction of the volume or area, respectively, being taken up by connections between cells or circuit elements, both systems crucially rely on efficient connectivity schemes. While restricting the interactions between elements of a distributed system makes the system less general, some preimposed structure might also be beneficial in certain scenarios, and it is likely that brains make use of representations and algorithms which are not harmed by, or even benefit from such constraints. Artificial systems might profit from similar principles, and the objective of this part is to point out some scenarios where this is the case.

Biological nervous systems heavily feature local connections, whereas long-range ones occur rather sparsely (Binzegger et al., 2004), a situation which seems inevitable given that longer connections require both more space and more energy (Attwell & Laughlin, 2001). This is due to the fact that signal transmission in neurons relies on active amplification along the ‘cable’, the axonal and dendritic processes of the cell. In vertebrates, the energy dissipation due to signal transmission is counteracted, to some extent, through myelination: insulating layers of a fatty substance are wrapped around axonal processes. However, thereby the thickness, and thus the physical dimensions of the substrate are increased, trading one scarce resource for another.

The detailed connectivity of neurons in the cortex can be investigated through electrophysiological experiments, for example, to obtain certain statistical properties. As an example, fig. 1.1 shows the spatial distribution of sites in the neural tissue that lead to responses in layer 2/3 neurons of the ferret visual cortex. Up to a certain distance, this distribution is well fit by an exponential function, suggesting that, on a statistical level, some of the connectivity can partially be described by an exponentially decaying connection probability (Budd & Kisvárday, 2001; Roerig & Chen, 2002). While efficient use of resources presumably is one aspect leading to such connectivity patterns, the intelligence of brains is likely not merely due to certain statistical properties. Since the function of a neural circuit is ultimately defined by its structure, development and learning play an important role in forming the specific connectivity. On the other hand, despite of excessive anatomical studies, countless theories and
Figure 1.1 Connectedness of ferret cortical neurons. The data shown correspond to measurements of responses of layer 2/3 neurons to stimulation of different sites in the neural tissue. The density of sites that led to responses is plotted as a function of the distance of the location of stimulation to the cell body. The line corresponds to a least-squares fit of an exponential function to the data (points up to 900µm were considered). Data adapted from Roerig & Chen, 2002.

models (e.g. Douglas & Martin, 2004; Marcus et al., 2014; Mountcastle, 1997), it remains an open question how the detailed structure of neural systems relates to their function (Douglas & Martin, 2011).

As in biological ones, in electronic computing systems a substantial fraction, in some cases the majority, of both space and power requirements can be attributed to wires, not transistors (Deutsch et al., 2001). It therefore seems reasonable to think about efficient wiring schemes for distributed processing systems, and to co-design computing architectures and algorithms that permit or even benefit from predominantly local connectivity (Dally & Towles, 2001). In fact, what is a limitation in general, can be advantageous in certain types of distributed systems, as is shown in the following two chapters. It turns out that structural constraints promoting or enforcing local connectivity can speed up convergence and lead to better results in certain scenarios where the system structure is to be learned through an optimization process. In particular, the effects of localized connectivity are illustrated in the context of feed-forward neural networks trained for a classification task, as well as in a bio-inspired recurrent network model for deterministic sequence processing.
Chapter 2

Learning in locally connected artificial neural networks

The enormous success of neural network-based machine learning methods, now commonly referred to as deep learning, has led to a wide range of applications in areas such as computer vision, object recognition, or speech processing (LeCun, Bengio & Hinton, 2015). Much of the success of these approaches can be attributed to the growing abundance of training data, as well as to an increased availability of computing resources, whereas the actual underlying neural network models and training methods (Rumelhart et al., 1986a; Werbos, 1974) have changed uninspiringly little over the past decades.

With both the amount of available data and the computing capacity being the main driving factors behind recent successes of deep learning systems, they are also very much their limiting factors in the sense that a model can only be trained as well (in reasonable time) as the available resources allow. One way of overcoming some of these limitations might be the utilization of network structures that make more efficient use of the computational resources than simple, fully-connected multilayer networks, where every unit in a layer is connected to all units of adjacent layers. To some extent, this has been achieved with convolutional neural networks, which essentially replace the dense connection matrix, containing of the order $n^2$ elements for layer sizes of order $n$, by a set of small kernels the layers are convolved with (Krizhevsky et al., 2012; LeCun, Bengio, et al., 1995). While allowing for reduced memory requirements and improved training outcomes, convolutions particularly make sense if they are processed sequentially: in an implementation on conventional computing architectures, the convolution kernel would be read from memory just once and then be applied
to segments of the input image, one by one. In a fully parallel and distributed system with local memory, however, a structure featuring local connectivity (without requiring sharing of parameters across entire layers) seems equally appropriate, if potential training advantages due to weight-sharing are ignored. In fact, such locally connected network models have produced state-of-the-art results in certain areas, such as face recognition (Taigman et al., 2014).

Figure 2.1 Non-standard connectivity profile: a neuron of layer $n$ receives inputs from a small subset of layer $n-1$ cells, where the connection positions are drawn from a probability distribution centered around the cell position.

In this chapter, simple feed-forward neural networks are considered, consisting of several layers of non-linear neurons, with neurons of each layer providing input to units of the next higher layer through weighted connections. Such networks can be optimized for a particular input-output relation by tuning the weight parameters through a stochastic gradient descent procedure, whereby examples of a training dataset are presented and the weights are updated such that the output error of the network is lowered. Previous work focusing on local interactions, i.e. convolutional networks and locally connected networks without weight-sharing, has mainly been based on rectangular kernels, where each neuron receives inputs from a rectangular patch of cells of the previous layer. Such perfectly rectangular receptive fields certainly do not occur in biological systems, and it is by no means clear that this choice is ideal or particularly good. The purpose of this chapter is to investigate the effects of non-standard connectivity profiles, as illustrated in fig. 2.1, on the network’s learning performance for a given level of sparseness. In particular, inspired by findings in biological systems, exponentially decaying connection probabilities are investigated, and are found to surpass other connection schemes in sparsely connected systems.
The results are meant to stand as a proof of concept, to show that the local distribution of connections does have an impact on the network performance, and further investigation is necessary to characterize the detailed effects of this approach.

### 2.1 Model and training details

To illustrate the effects of different connectivity patterns, small multilayer networks, consisting of an input layer of dimensions $28 \times 28$, two hidden layers of $30 \times 30$ neurons each, and an output layer of size 10, were trained on the MNIST handwritten digits dataset (LeCun, Cortes, et al., 1998). Thereby, the probability of two neurons of adjacent layers being connected was a function of the euclidean distance (distance between relative x-y coordinates of the cells within layers), defined for a cell $i$ in layer $n$ and a cell $j$ in layer $n - 1$ by

$$d_{ij} = \sqrt{\left(\frac{s_{y}^{(n-1)}}{s_{x}^{(n)}}x_{i}^{(n)} - x_{j}^{(n-1)}\right)^{2} + \left(\frac{s_{y}^{(n-1)}}{s_{y}^{(n)}}y_{i}^{(n)} - y_{j}^{(n-1)}\right)^{2}}, \quad (2.1)$$

where $s_{x}^{(k)}, s_{y}^{(k)}$ are the side-lengths of the rectangular grid representing a layer $k$.

In particular, three different cases were examined, with the connection probability defined by either of the following distributions:

1. Uniform connectivity: $p(d_{ij}) = p_0$,

2. Step function: $p(d_{ij}) = p_0$ if $d_{ij} < d_0$; 0 otherwise,

3. Exponential connectivity: $p(d_{ij}) = p_0 \exp(-\lambda d_{ij})$,

where $p_0, d_0,$ and $\lambda$ are constants, which were set to a range of different values, leading to different sparsity levels in the network. Specifically, for the spatially uniform distribution, $p_0$ was varied; for the step function, $p_0$ was fixed to 0.7 and $d_0$ was varied; and for the exponential distribution, $p_0$ was set to 0.7 and $\lambda$ was varied.

The networks were trained through stochastic gradient descent using the Adam algorithm (Kingma & Ba, 2014). The precision of the weight parameters was limited to three signed bits, such that small weight values were set to zero, increasing the sparseness of the network. To train the networks with quantized weights, dual-copy rounding techniques were employed, where a high precision copy of the weights is retained to compute the error gradients, while
in the inference phase the low precision values are used to compute the output (Courbariaux, Bengio & David, 2014; Stromatias et al., 2015). Weight noise was added during training as a regularizer. The networks were trained for 80 epochs on 50k samples from the MNIST training set, and then tested on the MNIST test set.

2.2 Improved classification accuracy in sparse networks

![Classification accuracy of networks trained with different connection schemes.](image)

Figure 2.2 Classification accuracy of networks trained with different connection schemes.

Figure 2.2 shows the classification performance of the network generated with different connection distributions $p$ and different sets of parameters, leading to different sparsity levels. The results shown correspond to the classification error on the MNIST test set. Significant differences for the different connectivity patterns can be observed: while the locally connected networks clearly outperform uniformly connected ones in all cases, the exponential connection probability seems to be beneficial in very sparse networks (less than 1% of connections retained), whereas the step function-based networks’ performance peaks at connection probabilities of around 10%.

2.3 Discussion

The presented results indicate that the way in which the connectivity between adjacent layers in a multilayer feed-forward network is constrained can have a substantial influence on the network’s learning performance. Specifically, less
trivial but more biologically realistic network structures seem to surpass simpler ones, and in particular so, in cases where the properties of the network are more similar to biological ones (high sparsity levels). Reducing the number of non-zero weights in neural networks, as well as model compression techniques, are active areas of research (Han et al., 2015; Wen et al., 2016) and a priori constraints as the ones presented here could serve as a starting point for such efforts. The results shown here are meant to serve as an example or motivation of what might be achieved by including constraints related to the physical substrate the models run on, in the models themselves. Further research is necessary to characterize the effects of particular connectivity patterns in more detail.
Chapter 3

Learning optimized automata in recurrent neural networks

Finite automata, or finite-state machines (FSMs), present one of the simplest frameworks to model deterministic sequential behavior, and are essential building blocks in theoretical computer science (Hopcroft et al., 2007). FSMs can model many aspects of high level deterministic behavior, such as production of movement sequences, navigation, state-dependent decision making, logical reasoning, or understanding and production of language. Although many neural processes can be better modeled by probabilistic graphical models, taking into account the inherent environmental and neural stochasticity, almost deterministic sequences of neural activation have been observed in brains of various species and during diverse activities. Examples include synfire chains (Abeles & Gat, 2001), sequences during song production in birds (Hahnloser et al., 2002), or location-dependent patterns during navigation in rats (Harvey et al., 2012).

It has been shown previously how FSM-like dynamics can be realized in networks of interconnected neural populations (Neftci, Binas, et al., 2013; Rutishauser & Douglas, 2009; Rutishauser, Slotine, et al., 2012). In these implementations, states are represented as point attractors in a multistable attractor neural network. Special gating units implement a mechanism for switching between different states conditional on the current state and the external input stimulus. This framework allows bottom-up engineering of given finite automata by setting the relevant connections in the network, i.e. the transition table of a given automaton can directly be translated to connections between populations. The described model is particularly appealing as it is based solely on interconnected winner-take-all networks, small circuits that closely match
connectivity patterns found in neuroanatomy (Douglas & Martin, 2004).

While those previous approaches have addressed the synthesis of state machines with known dynamics, this chapter presents a biologically plausible neural framework for learning deterministically behaving systems in the form of neural FSMs. The work initiated by Rutishauser & Douglas, 2009 is extended through a reinforcement learning rule, such that an agent equipped with the proposed mechanism can learn and adapt its deterministic behavior through sparse, external reward signals. Specifically, the network produces an output after receiving a sequence of inputs and is provided feedback in terms of a reward signal that indicates whether the output corresponds to the target behavior. The network then adapts its internal model using a reward-modulated Hebbian-type learning rule (Friedrich et al., 2011; Pfeiffer et al., 2010) to maximize the correspondence of its behavior with the target behavior.

For any neural system, the number of states required to model the target behavior cannot be known in advance. However, it is crucial for such a system to make efficient use of the available neural resources. It is found that within the proposed framework simple, local modulation of the network structure leads to greatly improved solutions in terms of the optimality of the solutions that are learned, as well as in the search time required to find a valid solution. Note that the term optimal refers to the minimal automaton that implements the desired dynamics, here. The structural assumptions made nicely translate to real biological systems, where nearby populations of neurons are more likely to be connected than far away ones.

In the following, the model is first introduced formally, and then mapped to a network of neural populations. Simulations show how local structure in the network connectivity leads to solutions that are globally optimized in terms of the number of state populations used to implement a target behavior, while at the same time improving the learning performance.

3.1 Model definition

An FSM is defined by a set of states $S = \{1, 2, 3, \ldots\}$, an alphabet of input symbols $A = \{a, b, c, \ldots\}$, and a transition table $T$ that contains an entry $T_{ik} = j$ for every state $i$ and input symbol $k$, specifying a transition to a target state $j$ for that particular state-input combination. Furthermore, there is an initial state $s_0$, and a set of one or multiple accept states $S_{\text{acc}} \subseteq S$. The system computes by switching between its internal states while processing a finite string of input
symbols. The outcome is binary: the system either ends up in one of the accept states in $S_{\text{acc}}$ or not. This mechanism can be used to parse and classify given input strings, i.e. to make a decision on the basis of a sequence of input stimuli. An illustration of a simple FSM is shown in fig. 3.1a. The system is initialized in state 1 and ends up in the accept state 3 if a number of blue inputs is provided, followed by a number of red inputs. Note that self-transitions are not shown, but are assumed implicitly for all inputs for which no transition is specified.

### 3.1.1 Neural populations as computing elements

The implementation of finite automata in attractor neural networks is illustrated by means of an abstract population model, which is used to illustrate the learning rule and various aspects of the plastic system.

![Figure 3.1](image)

**Figure 3.1** Building state machines from neural populations. **a** Abstract directed graph representation of an example state machine, consisting of three states (1, 2, 3) and two input symbols (red, blue). Arrows indicate state transitions, double circles indicate accepting states. **b** Implementation of the state machine from **a** in a neural network of coupled WTA circuits (dashed boxes). The units $X_i$ in the top WTA represent the states, while the units $y_{jk}$ of the lower WTA trigger transitions between them. Each unit in the lower WTA corresponds to one particular input-state combination, indicated by color and state index, and can activate its target state through a strong excitatory projection.

The neural implementation is based on two interconnected Winner-Take-All (WTA) networks (fig. 3.1b), one of which serves as memory of the current state while the other one functions as a gating mechanism, activating exactly one transition per state-input combination. This formalism was introduced by Rutishauser & Douglas, 2009. A WTA network contains a number of neural populations that compete through mutual inhibition, such that only one of them can be active at a time. If all populations are configured identically, the active population of a WTA is the one receiving the greatest input. If local self-excitation
Chapter 3  Learning optimized automata in recurrent neural networks

is added to the circuit, i.e. each population recurrently excites itself, the populations can retain their activity, and therefore a memory, even if no external input is provided. However, due to the mutual inhibition only one population is active at a time, such that one can switch from one attractor state to another by providing a strong input to the other population. A WTA network with recurrent excitation is used to represent the internal states, that is, activation of population $x_i$ indicates that the system is in state $i$ and this state is retained even if no input is provided.

A second WTA network without recurrent excitation implements the transitions between states. This second WTA contains exactly one population for every state-input symbol combination, whereby each population receives input from one input symbol and one of the state populations. Due to the competition mechanism, only the population $y^{j,k}$ corresponding to the current state $j$ and the current input symbol $k$ will become active. An activation threshold makes sure that the populations become active only if they receive both input from a state population and an input symbol and are silent otherwise. In order to implement a transition to a target state, the population corresponding to a particular state-input symbol pair simply needs to provide a strong input to the population representing the target state. Furthermore, a random, non-empty subset of the state populations $X_{acc} \subseteq X$ is defined as the set of accepting states, while all other state populations correspond to reject states. One state population $x_0$ is defined to represent the initial state.

The corresponding population-based implementation of the FSM illustrated in fig. 3.1a is shown in fig. 3.1b. It is discussed elsewhere (Neftci, Binas, et al., 2013; Rutishauser & Douglas, 2009) how the neuron parameters have to be chosen to achieve the desired dynamics in networks of threshold-linear units. In this study, for simplicity, the activation of a unit is assumed to be either 0 or 1 and transient dynamics are not considered.

3.1.2 The learning rule

A neural network of the type described above is initialized with transition weights set to random values in $[w_{\text{min}}, w_{\text{max}}]$ where $w_{\text{min}}$ and $w_{\text{max}}$ are chosen such that the input to state populations is large enough to switch to a different state. The training is based on a number of trials, each of which consists of an input sequence of length $N_t$ and a reward signal $r_t$. At the beginning of a trial $i$, the agent is set to its initial state, i.e. the population $x_0$ that was specified
to correspond to the initial state is activated. The trial length $N_i$ is then chosen from a uniform distribution over $[1, N^{\text{max}}]$ and $N_i$ randomly chosen input symbols are provided at discrete times $t_1, \ldots, t_{N_i}$ for a period $\tau_0$ each. At the end of the sequence, it is checked whether the agent is in an accept state or not and whether this outcome corresponds to the outcome of the target automaton if provided the same sequence of inputs. If the outcomes are in agreement, a reward $r_i = 1$ is provided to the agent. If not, a negative reward signal (or punishment) $r_i = -1$ is provided.

To learn the right transitions, the learning rule outlined in the following solely needs to alter the ‘transition connections’, i.e. the connections from the gating populations to the state populations. All other connections remain fixed, although they could in principle also be learned with biologically plausible rules (Binas, Rutishauser, et al., 2014). Due to the competition mechanism, only one of the state populations will be active after an input symbol is provided and a gating population has been activated, namely the one receiving the greatest input from that gating population.

The connection from the gating population $y^k_j$ to the state population $x_i$ is denoted by $w^k_{ij}$. While inputs are provided and the system switches between internal states, each transition connection $w^k_{ij}$ locally records the traces $\Gamma^k_{ij}$ and $\tilde{\Gamma}^k_{ij}$, which are updated after every input presentation as follows:

$$\Gamma^k_{ij} \rightarrow (1 - \lambda) \Gamma^k_{ij} + \lambda y^k_j(t_n)x_i(t_n + \tau_0), \quad (3.1)$$

$$\tilde{\Gamma}^k_{ij} \rightarrow (1 - \lambda) \tilde{\Gamma}^k_{ij} + \lambda y^k_j(t_n)(1 - x_i(t_n + \tau_0)), \quad (3.2)$$

where $\lambda$ is a constant between 0 and 1 (in all simulations we set $\lambda = 1/3$) that determines the speed of decay of the traces. The activation of a population is assumed to be either 1 (active) or 0 (silent) at any time. Thus, the quantity $\Gamma^k_{ij}$ is increased whenever the pre- and postsynaptic populations of the corresponding connection $w^k_{ij}$ are co-active, while $\tilde{\Gamma}^k_{ij}$ is increased when only the presynaptic population $y^k_j$ is active.

At the end of the trial, the reward signal $r \in \{-1, 1\}$ is provided and the weights are updated proportional to the learning rate $\eta$ (set to 1 in all simulations) as follows:

$$w^k_{ij} \rightarrow w^k_{ij} + r\eta \left( g_r(w^k_{ij})\Gamma^k_{ij} - \tilde{g}_r(w^k_{ij})\tilde{\Gamma}^k_{ij} \right), \quad (3.3)$$
where $\tilde{g}_r = g_{-r}$ and

$$g_r(w^{k}_{ij}) = \begin{cases} w^{\max}_{ij} - w^{k}_{ij} & \text{if } r = 1, \\ w^k_{ij} - w^{\min}_{ij} & \text{if } r = -1. \end{cases} \quad (3.4)$$

The weight dependence $g_r$ ensures that the minimum and maximum weight values are never exceeded. Overall, the update can be regarded as a reward-modulated Hebbian learning rule. This is evident when noting that either $\Gamma^{k}_{ij}$ or $\tilde{\Gamma}^{k}_{ij}$ (or both) are zero for a given connection after the trial. This is the case because weights are not updated during the trial, so a transition is either taken always or never. If the reward is positive, connections whose pre- and postsynaptic populations have been active together are strengthened. On the other hand, connections which do not lead to activation of a postsynaptic population are weakened. If the reward is negative, the update is inverted, i.e. the connections corresponding to transitions that took place are weakened, while others are strengthened. Thus, over many trials, the connections should converge to values that lead to the maximum reward and therefore always fulfill the target dynamics.

### 3.1.3 Local structure in the network topology

In a realistic scenario, the number of available state populations will not exactly match the number of states required for a given target automaton. Since the exact amount of states required to implement a specific target behavior is not known a priori, the number of state populations should be large, so they can be recruited dynamically as the training progresses. This will typically lead to rather large systems (large in terms of the number of populations used) since there is no motivation for the agent to minimize the solution, as it can just recruit more populations if a new branch in the behavior is required. More complex (or larger) network structures, on the other hand, make it harder to learn the right transitions between internal states, as the number of possible transitions scales with the square of the number of states used. To encourage small solutions we assume the state populations to be uniformly distributed in some physical space, e.g. on a two-dimensional grid or on a line. Small solutions will then be guaranteed if only a small region of space is used to implement the desired dynamics. To make such local solutions more likely, local structure is introduced in form of a modulation of the maximum weight as a function of
3.2 Theoretical considerations

The distance between populations,

\[ w_{\text{max}}(w_{ij}^k) = w_{\text{max}}^0 - \mu d(x_i, x_j), \tag{3.5} \]

where \( w_{\text{max}}^0 \) and \( \mu \) are constants, and \( d \) is a function of the coordinates of the populations. In all simulations, \( w_{\text{max}}^0 \) was set to 1, and \( \mu \) to 0.025. For the remainder of this chapter, only the simple one-dimensional case is considered, where populations are arranged at equidistant points along a line with their indices corresponding to their coordinates. The Euclidean distance between two populations is thus given by \( d(x_i, x_j) = |j - i| \). This allows states that are represented by populations which are spatially close to each other to achieve higher transition weights, so these states are more likely to participate in a transition. Such local structure is also found in biology, where short-range connections between cells are much more likely than long-range ones.

The purpose of this section is to provide an intuition why the learning method works in practice. In general it cannot be guaranteed that for any finite set of input sequences the algorithm finds the exact finite state machine that generated the sequences, or that it can generalize perfectly for arbitrary longer or unobserved sequences. Denoting by \( \mathbf{X} \) the set of all state populations in the network and by \( \mathbf{X}_{\text{acc}} \subseteq \mathbf{X} \) the set of all accepting states among them, a solution is guaranteed to exist if these sets are sufficiently large, i.e. if \( |\mathbf{X}| \geq |\mathbf{S}| \) and \( |\mathbf{X}_{\text{acc}}| \geq |\mathbf{S}_{\text{acc}}| \), where \( \mathbf{S} \) and \( \mathbf{S}_{\text{acc}} \) are the sets of states and accepting states of the target automaton. Once the system has converged to a valid state where it receives positive reward after every trial, the connections \( w_{ij}^k \) that trigger a state transition asymptotically converge to \( w_{\text{max}} \), while all \( w_{i'j}^k \) with \( i' \neq i \) converge to \( w_{\text{min}} \).

Even though a full proof for convergence of the learning rule eq. (3.3) to a correct solution for any given training set is not provided, there are intuitive reasons why the algorithm works in practice:

- If a network state \( j \) is repeatedly visited and positive reward is obtained directly after the transition into state \( i \) following the same symbol \( k \), the corresponding weight \( w_{ij}^k \) will always grow, while all weights \( w_{i'j}^k \) with \( i' \neq i \) will shrink. Thus, a correct final transition will only be reinforced.
• If in the same situation the reward is always negative, the corresponding weight $w_{ij}^k$ will shrink, while all weights $w_{i'j}$ with $i' \neq i$ will grow. This will continue until eventually a different transition becomes active (as soon as $w_{ij}^k < w_{i'j}^k$ for some $i'$). Even if this is again an incorrect transition, after a finite number of updates the algorithm will lead to a correct transition, because weights leading to correct states can only grow, whereas weights that lead to incorrect states will shrink as soon as they are selected.

• If $\lambda < 1$, all weights along the state sequence leading to the positive or negative reward will be updated, and either all of them will be reinforced or punished. Weights of transitions near the end of the sequence will receive the strongest update, similar to reinforcement learning algorithms using exponentially decaying eligibility traces (Sutton & Barto, 1998). Thus, if only the final transition is incorrect, this is the most likely transition to change, whereas previously learned correct transitions will be reinforced by repeatedly seeing shorter sequences, assuming that sequences are presented repeatedly and in random order.

• A potentially problematic situation can arise whenever the algorithm has converged to a perfect solution for all sequences seen so far, and then receives negative reward for a longer, previously not observed sequence. In the worst case, this might require reorganization of all previously learned transitions. As a simple example, consider an FSM that accepts the string ‘1’, but not strings with 2 or more 1 symbols. If initially the string ‘1’ is presented many times, and assuming $x_0$ is an accepting state, this might lead to a learned network in which the weight $w_{00}^1$ is the maximum of all $w_{i0}^1$, i.e. the network will accept strings with an arbitrary number of ones. In this case, it is necessary that negative examples from the training set are seen often enough and repeatedly so that the self-transition can be unlearned. Since transitions into other accepting states would also be positively rewarded, and weights towards those states would grow for every incorrectly classified sequence, such a transition will be learned in the limit of repeated and randomly ordered presentation of sequences. Thus, eventually a different transition from $x_0$ for symbol 1 will be learned, which enables learning of a correct network.
3.3 Simulation results

The performance of our system, and particularly the effects of the local structural modulation described in section 3.1.3, are evaluated for different training and model parameters. This is done by learning a random target dynamics, generated by defining a target FSM with a fixed number of states and input symbols, a randomly initialized transition table, a random initial state, and a random, non-empty set of accept states. Note that occasionally this might lead to target FSMs for which equivalent models can be realized with fewer states, as the random transition tables might implement several equivalent paths to reach a particular state, using more states than necessary. To investigate the influence of different parameters, the number of states of the target model, the average length of the input strings, and the number of state populations available to the neural network were varied. Each system was trained according to the procedure described in section 3.1.2, by presenting random input strings of a maximum length $N_{\text{max}}$ and providing reward signals depending on the final state of the model system. When the system was able to produce the correct outcome for 1000 trials in a row we considered the system to have learned the correct transition dynamics. All experiments were repeated 100 times for both the simple model and the model with local modulation of the maximum weight. Each data point in fig. 3.2 thus corresponds to an average over 100 experiments.

Figure 3.2b shows the number of trials required as a function of the number of states the target model is based on. As expected for problems of this kind, the search time scales exponentially with the problem size. However, local modulation of the transition connections improves the performance by more than an order of magnitude. If the number of state populations available to the system is increased, this difference increases further, whereby the simple model performs worse while the performance of the structurally modulated model remains roughly the same.

The search time as a function of increasing target system complexity shows a similar scaling trend for the modulated and the simple system. However, a difference in scaling between the two models can be observed if the average length of the input strings is varied while keeping the number of states of the target system fixed. For fig. 3.2d, the number of states of the target model was held fixed at 4, and the maximum input string length was varied between 4 and 16. Networks with structural modulation show only a small increase in learning
Chapter 3  Learning optimized automata in recurrent neural networks

Figure 3.2  a System complexity vs. available states. Varying the number of available state populations, the number of actually used network states is measured with the target system size fixed at 4 states. For the system with local structural modulation, the number of states used remains approximately constant, while it increases linearly for the system without modulation.  b Search time vs. target system complexity. Varying the number of states of the target model, the number of trials needed to find the correct dynamics was measured. For the experiments, the average input string length was set to 16, using 2 input symbols, and networks had 32 available state populations. Networks with local structural modulation converged on average more than one order of magnitude faster.  c Search time vs. available states. Varying the number of available state populations, the number of trials required is shown with the target system size fixed at 4 states.  d Search time vs. input string length. The number of trials needed to find the correct dynamics was measured, while the average length of the input strings was varied. For the experiments, the target models had 4 states, using 2 input symbols, and networks had 32 state populations available. The training time for networks with local structural modulation shows only a minor increase for longer sequences. All results shown correspond to averages and standard deviations over 100 experiments.
optimized solutions due to local structure

As a key result, fig. 3.2a shows the dependence of the complexity of the learned system on the number of state populations available. While the size of the simple system without modulation grows approximately linearly with the number of state populations, i.e. a constant fraction of the available state populations tend to be used, there is no such dependence in the system with structural modulation. Regardless of the number of states available, the system only uses roughly as many populations as are minimally required for the optimal solution (the target system complexity was fixed to 4 states in this example). Figure 3.3 displays the amount of states used and the number of trials required for 100 random initializations of the simple and the locally modulated systems, and for a fixed target dynamics (shown in fig. 3.5b). The systems consisted of 32 state populations each, whereby population 15 was specified to represent the initial state and every third population \( x_0, x_3, x_6, \ldots \) was specified to represent an accepting state. While most of the implementations found by the locally modulated system used a number close to the optimum of three states, (on
Chapter 3 Learning optimized automata in recurrent neural networks

Figure 3.4  a shows the smallest system found in 100 experiments without structural modulation (cf. fig. 3.3) and b shows a typical system that is found when using structural modulation. The system in b is the minimal system that implements the target dynamics, and is confined to a small region in state space (states 14 to 16), whereas the implementation found by the non-modulated system consumes a large number of state populations and is distributed over the whole (linear) state space of size 32 to implement the same behavior.

average, about 6 states were used), the non-modulated system on average used more than 20 of the 32 available states with the smallest solution using 13 states. A typical solution found by the modulated system, and the smallest solution found by the simple system are shown in fig. 3.4. While the modulated system finds a locally confined solution, the solution of the simple system spans the whole range of state populations between 0 and 31.

We can conclude that simple constraints on the local connectivity structure encourage optimal solutions in terms of the number of states used. Globally, this modulation of the weight update leads to spatially confined solutions, such that the number of states used to implement the dynamics of the agent becomes independent of the number of states available in the system. This simplifies the learning process, because effectively the number of possible transitions, which scales with the number of states that can be reached, is reduced. Other parameters, e.g. the learning rate, can be modulated in a similar fashion and were found to lead to similar results. For the sake of clarity, only the results for the modulated maximum weight are presented here.
3.5 Learning to solve behavioral tasks

So far, we have considered random target dynamics for our system to learn. In this section, it is demonstrated how the mechanism can be applied to actual behavioral tasks. A common type of behavioral experiment in neuroscience is based on a rodent navigating through a maze, learning how to use implicit (visual) cues as a navigation aid (e.g. Harvey et al., 2012). Here, virtual instances of such experiments are generated in the form of random mazes, in which an agent equipped with the proposed learning mechanism has to find a target position. Specifically, the mazes are generated on a two-dimensional grid by defining an initial and a target position, and connecting them through a sequence of consecutive T-junctions (see fig. 3.5a). The agent moves along the grid in discrete steps, and at each junction has to decide whether to turn left or right. If a wrong turn is performed and the agent runs into a dead end, the trial ends and negative reward is provided. On the other hand, if the goal is reached after a number of correct turns, a positive reward signal is provided. As input symbols to the system, there are colored tiles distributed across the maze, encoding the correct direction to take at the next turn. These inputs are provided to the agent at discrete points in time, whenever its position coincides with the respective tile. In order to solve the maze correctly, the agent has to learn the meaning of the input symbols. In the example shown in fig. 3.5a, two consecutive red tiles indicate that a left turn is necessary, while the sequences blue-red and red-blue encode a turn to the right. At each turn, the internal state (the outcome of the computation) of the agent is checked. If in an accept state, a turn to one direction is performed, if not, the opposite action is carried out. One way of solving this task is to learn how to count to two (i.e. to reset the internal state after every two inputs received) and to detect whether blue is contained in a pair of consecutive inputs. During the training process, a new maze is generated in every trial, while the meaning of the input sequences remains the same. One possible system that successfully solves the task, and that was occasionally found by the agent, is shown in fig. 3.5b. Typically, the system required on the order of several 100 trials to learn the correct behavior.

3.6 Discussion

The model outlined in this chapter is interesting for two reasons. On the one hand, it hints at how sequential, deterministic processing might be learned and
implemented in biological systems, on the other hand, it points out technological advantages that arise through bio-inspired structural constraints.

To implement state-dependent behavior, biological nervous systems rely on some form of working memory. This can be realized, for example, by a recurrent neural network forming stable attractor states with different memories encoded as fixed points in the network dynamics. Attractor networks have been investigated extensively and are hypothesized to implement important dynamical elements in biological brains (Amit, 1992). Moreover, it has been shown that stable WTA dynamics, which form the basis of the types of attractor networks used in this work, can emerge in a self-organized way through an interaction of biologically realistic learning rules (Binas, Rutishauser, et al., 2014). Here, an intuitive update mechanism for the connections between neural populations is introduced on top of the WTA network structure, resulting in a simple reinforcement learning rule. While various models and algorithms for learning of finite automata exist (Jacobsson, 2005; Thathachar & Sastry, 2002), the presented neural network implementation closely matches connectivity patterns found in neuroanatomy. Interestingly, more realistic network structures (as compared to fully connected recurrent networks) seem to have a very beneficial effect on the performance of the system. As a key result of this study, bio-inspired local structure is introduced as a constraint to the network connectivity and is shown to lead to globally optimized solutions. In finite automata theory, there exist algorithms for minimizing a given automaton (Hopcroft, 1971; Moore, 1956). These are important tools as the minimal automaton ensures minimal computational cost for tasks such as pattern recognition (Hopcroft et al., 2007). In the investigated model, the preference for small solutions comes for free, as a side-effect of the locally modulated update rule, illustrating how local properties of a system can encourage global optimality. A less complex or even minimal system during training makes the training process less costly, as the number of possible state transitions is effectively reduced compared to the simple system based on a flat hierarchy. A future research direction could be the investigation of effects of local structure in non-deterministic automata, as in this case algorithmic minimization of a given automaton is much more expensive than in the deterministic case (Hopcroft et al., 2007).
a Example instance

Figure 3.5  a One of the randomly generated mazes with visual cues in the form of colored floor tiles. Consecutive occurrence of two red tiles indicates that the next turn should be left, while the sequences blue-red and red-blue signal that the next turn should be a right. The initial position of the agent is marked by a triangle, the goal (reward) by a star. Punishment is provided and the trial is ended if the agent makes a wrong turn (crosses). b The minimal automaton that solves the task, and which was occasionally learned by the system. At each turn, the state of the system is checked and if in an accept state (L), a left turn is performed. Otherwise, a right turn is carried out. a Reward obtained during learning (positive reward: black; and negative reward: red). As the agent improves, it receives more and more positive rewards until it converges.
PART II

Spiking neurons for efficient inference and learning
Introduction

Deep neural networks (LeCun, Bengio & Hinton, 2015) have produced impressive results in areas such as computer vision (He et al., 2015; Krizhevsky et al., 2012) or speech recognition (Hinton, Deng, et al., 2012), and yet are dwarfed by biological neural systems in terms of compactness, speed, and energy efficiency. As a consequence, there has been a growing interest in artificial neural network models whose style of processing is closer to biology in the sense that individual neurons communicate with each other asynchronously and through sparse events, or “spikes” (Cao et al., 2015; Diehl et al., 2015; Hunsberger & Eliasmith, 2015; O’Connor & Welling, 2016). Such event-based implementations can potentially compute results faster while requiring fewer operations than their synchronous, frame-based counterparts (Neil et al., 2016). In particular, an asynchronous event-based system can process individual input events (e.g. updates of individual pixels in the case of image data) immediately as they are provided, without having to wait for full frames (e.g. whole images) to be loaded. While the network computes, only those neurons whose states change emit spikes and may trigger computation in subsequent layers, thus saving computational resources. Another fundamental difference between artificial and biological neural networks is in the way they learn. Multilayer artificial networks are typically trained through stochastic gradient descent by means of the backpropagation algorithm, where parameter updates are computed as a function of precise error gradients, which are propagated backwards through the network (Rumelhart et al., 1986a). On the other hand, learning in biological neural systems is based on temporal correlations: individual weights are updated as a function of spike times of the corresponding neurons (Bi & Poo, 1998; Markram, Lübke, et al., 1997). Recent work has shed some light on how these seemingly different weight update mechanisms could be related (Scellier & Bengio, 2016; Schiess et al., 2016; Xie & Seung, 2003) and, to some extent, serves as an inspiration for some of the models presented here.

In chapter 4, a simple model for efficient event-based implementations of multilayer neural networks is presented, based on abstract neuron dynamics which map favorably onto digital hardware. In chapter 5, this model is extended with a learning mechanism, where weight updates are computed purely locally, as a function of the states of connected neurons.
Chapter 4

A simple neuron model for efficient event-based neural networks

The potential advantages of event-based implementations have sparked interest in dedicated spiking neural network electronic devices based on such processing schemes (Indiveri, Corradi, et al., 2015; Merolla, Arthur, et al., 2014). Achieving state-of-the-art deep learning accuracy in event-based systems has remained challenging, but recently a number of methods have been proposed which convert a previously trained conventional analog neural network (ANN) into a spiking one, showing promising results (Cao et al., 2015; Diehl et al., 2015; Esser et al., 2016; Hunsberger & Eliasmith, 2015). The principle of such conversion techniques is to approximate the continuous-valued activations of ANN units by the spike rates of event-based neurons. Although successful on several classical benchmark tasks, all these methods suffer from approximation errors, and typically require a multitude of spikes to represent a single continuous value, thereby losing some of the advantages of event-based computation.

Here, a minimalist event-based asynchronous neural network model is proposed, which processes input data streams continuously as they arrive, and which is shown to be formally equivalent to conventional frame-based models. A feed-forward network based on the proposed model yields first predictions of the output typically already after a few data packets have been received, and well before the full input pattern has been presented to the network. This model can be used to build efficient event-based processing systems, potentially in conjunction with event-based sensors (Delbrück et al., 2010; Posch et al., 2014).

In this chapter, it is demonstrated how the computation carried out by
counter networks exactly reproduces the computation done by a conventional frame-based network, however, with a power and resource efficient representation and communication scheme. Specifically, as a consequence of the event-driven style of processing, the resulting networks do not require computationally expensive multiplication operations. Initially, a simple model based on binary activations is introduced, and then extended to non-binary activations. The performance of networks based on the proposed models is evaluated on the MNIST dataset. Numerical results indicate that counter networks require fewer operations than previous approaches to process a given input, while enabling state-of-the-art classification accuracy.

4.1 Multiplication-free networks

Multiplications are the most expensive operations in artificial neural networks running on digital hardware. It is therefore desirable to reduce the number of required multiplications to a minimum. Previous work has shown how frame-based neural networks can be implemented using additions only, by either restricting all weights to binary (or ternary) values (Courbariaux, Bengio & David, 2015; Hwang & Sung, 2014; Lin *et al.*, 2015), or by using binary activations (Courbariaux & Bengio, 2016). The binary variable (weight or activation) then represents an indicator function, and all neural activations can be computed by simply adding up the selected weights or activations. The instantaneous output of an event-based neuron can be considered binary, too: either there is an output spike or not. However, non-binary values can still be encoded in terms of multiple spikes provided over time.

To introduce the event-based model in its most basic form, consider a set of neurons with binary activations, such that the output of all neurons $y_i^{(k)}$ in layer $k$ is given by

$$y^{(k)} = \sigma \left( W^{(k)} y^{(k-1)} - \theta^{(k)} \right),$$

(4.1)

where $W$ is the weight matrix, $\theta$ are threshold values corresponding to bias terms, and

$$\sigma(x) = \begin{cases} 1, & \text{if } x > 0, \\ 0, & \text{otherwise}. \end{cases}$$

(4.2)
Thus, the output of a neuron is 1 if its net input is greater than its threshold value $\theta$ and 0 otherwise. While this model does not pose any constraints on the weights and thresholds, we use low precision integer values in all experiments to keep the computational cost low and allow for highly efficient digital implementations. We consider here multi-layer networks trained through stochastic gradient descent using the backpropagation algorithm (Rumelhart et al., 1986a). Since the error gradient is zero almost everywhere in the discrete network given by eqs. (4.1) and (4.2), the binarization function $\sigma$ is replaced by a logistic sigmoid function $\tilde{\sigma}$ in the backward pass,

$$\tilde{\sigma}(x) = \frac{1}{1 + \exp(-\lambda x)},$$

(4.3)

where $\lambda$ is a scaling factor. Furthermore, during training, copies of the high-resolution weights and activations are kept, and used to compute the gradient in the backward pass, as proposed by Courbariaux, Bengio & David, 2015; Stro- matias et al., 2015. In addition to the activations and network parameters, the inputs to the network are binarized by scaling them to lie in the range $[0, 1]$ and rounding them to the nearest integer.

### 4.2 Lossless event-based implementation

The multiplication-free network proposed above can directly be turned into an asynchronous event-based neural network by replacing every unit of the ANN by a counter neuron, as described below. The weights and biases obtained through the training procedure above can be used in the event-based network without further conversion.

Each counter neuron is defined by an internal counter variable $c$, which is updated whenever the neuron receives positive or negative inputs in the form of binary events (or spikes). The neuron operation is illustrated in fig. 4.1 and is described by algorithm 1. A counter neuron essentially counts, or adds up, all the inputs it receives. Whenever this sum crosses the threshold $\theta$, a binary event $\pm 1$ is emitted. The sign of the event depends on the direction in which the threshold was crossed, i.e. a positive event is emitted when the threshold is crossed from below, and a negative event when $c$ falls below the threshold. Whenever neuron $j$ emits an event the quantity $\pm w_{ij}$ is provided as input to neuron $i$ of the next layer, with the sign determined by the output of neuron $j$. Thus, the neurons themselves do not continuously provide output signals,
Input

Counter state

Output

Time

Figure 4.1 Illustration of the basic counter neuron model. The internal counter variable \( c_i \) is initialized at the negative threshold level \(-\theta_i\) and accumulates inputs provided through events at discrete time steps. Whenever the counter changes its sign, i.e. when the accumulated input becomes greater or smaller than the threshold level \( \theta_i \), an event is emitted. The event is positive (+1) if the sign changes from negative to positive, and negative (−1) otherwise.

which makes information transmission and computation in the network very sparse. The input to the network is provided in event-based form as a stream of binary events (or spikes), i.e. the network at discrete points in time receives a list of indices of one or more pixels, indicating that these pixels are active. Specifically, a binary input image (or other data) can be streamed to the network pixel-by-pixel in an asynchronous manner, whereby the order and exact timing of the pixels does not matter. It can be formally shown with little effort that an event-based network based on counter neurons produces exactly the same output as its frame-based counterpart.

Proof of the equivalence. To prove the equivalence of the frame-based and the event-based model we have to show that the outputs of individual neurons are the same in both settings. In the following, we assume \( \theta_k > 0 \ \forall k \) without loss of generality. Let an event-based network be initialized at time \( t = 0 \), such that \( c_k(0) = -\theta_k \ \forall k \). Within the time interval \([0, T]\), neuron \( i \) of layer \( m \) in the event-based network receives a sequence of \( N \) inputs, \((w_{ij}^{(m)}s_1, \ldots, w_{ij}^{(m)}s_N)\) from a set of source neurons \( j_1, \ldots, j_N \) at times \( t_1, \ldots, t_N \), where \( s_k \) is the sign of the \( k \)th event, and \( 0 \leq t_k \leq T, \forall k \). It follows from algorithm 1 that the value of the
initialize: $c_i \leftarrow -\theta_i$

whenever neuron $i$ receives an input event $\text{inp}_i = \text{sign}_{j_*, w_{ij}}$ from a neuron $j^*$ or a sum of simultaneous input events $\text{inp}_i = \sum_{j \in J} \text{sign}_{j} w_{ij}$ from a set of neurons $J^*$ do

$c_i^{\text{prev}} \leftarrow c_i$

$c_i \leftarrow c_i + \text{inp}_i$

if $c_i^{\text{prev}} \leq 0$ and $c_i > 0$ then

emit $+1$

end

if $c_i^{\text{prev}} > 0$ and $c_i \leq 0$ then

emit $-1$

end

end

Algorithm 1: Basic counter neuron implementation.

counter variable $c_i^{(m)}$ at time $T$ is

$$c_i^{(m)}(T) = c_i^{(m)}(t_N) = \sum_{k=1,...,N} w_{ik}^{(m)} s_k - \theta_{i}^{(m)},$$ (4.4)

as it simply sums up the inputs. The sign of $c_i^{(m)}$ might change several times during the presentation of the input, and trigger the emission of a positive or negative event at every zero-crossing. Since $c_i^{(m)}$ is initialized at $-\theta_{i}^{(m)} < 0$, there are $2n + \sigma(c_i^{(m)}(t_N))$ sign changes in total, where $n$ is a non-negative integer, and thus the total input communicated to neuron $k$ of the next layer is

$$\sigma(c_i^{(m)}(t_N)) w_{ki}^{(m+1)} = \sigma(\sum_{k=1,...,N} w_{ik}^{(m)} s_k - \theta_{i}^{(m)}) w_{ki}^{(m+1)} =: \hat{y}_i^{(m)} w_{ki}^{(m+1)},$$ (4.5)

as the $2n$ sign changes cancel. On the other hand, recursively applying eq. (4.5) leads to

$$\hat{y}_i^{(m)} = \sigma(\sum_{j} w_{ij}^{(m)} \sigma(c_j^{(m-1)}(t_N)) - \theta_{i}^{(m)}) = \sigma(\sum_{j} w_{ij}^{(m)} \hat{y}_j^{(m-1)} - \theta_{i}^{(m)}).$$ (4.6)

Since $\hat{y}_k^{(0)} = y_k^{(0)}$ for the input layer, the equivalence must also hold for all higher layers, according to eq. (4.6).

With the notion of equivalence, it is clear that the event-based network, if configured with the parameters obtained for the frame-based network, is able to exactly reproduce the output of the latter. Unlike in previous work (Cao et al.,
2015; Diehl et al., 2015), the resulting event-based network is guaranteed to provide the same result as the ‘ideal’ frame-based implementation. Thereby, the respective output value can be obtained by adding up the events emitted by the output layer. Technically, the equivalence holds only in the case where the full stream of input events has been presented to the network, and propagated through all layers. In practice, however, a few input events are often enough to activate the right neurons and produce the correct output long before the full set of input events has been presented. As a consequence, on average far fewer operations than in the frame-based model are required to compute the correct output (see fig. 4.2 for illustration).

Figure 4.2 Example run of the event-based system. A single input pattern of the MNIST test set is presented over time (one pixel per timestep) to a trained counter network. Positive events are displayed in black, negative events in red. The correct output (class 3) for this input pattern can be read out long before the whole input has been presented. Shaded regions indicate that the output of the network is correct during these periods, i.e. only the output neuron for class 3 has produced more positive than negative output spikes.
4.2.1 Extension to non-binary inputs

The constraint that the input patterns are binary, and each input unit either produces an event or not, can be safely relaxed to integer-valued inputs without further modifications of the model. The framework thus supports finer grained input scales, which is important e.g. for input images using multiple gray-levels or RGB values to encode different colors. The simple modification is that each individual input unit produces over time a number of events that corresponds to the encoded integer value. While such integer-valued inputs would require multiplication in the frame-based network with non-binary (or non-ternary) weights, the event-based network remains free of multiplication operations, as the instantaneous output of any input unit is still binary.

4.2.2 Extension to non-binary activations

```
initialize: z_i ← 0; c_i ← −θ_i

whenever there is an input event inp_i = sign_j w_{ij} from a neuron j* or a sum of simultaneous input events inp_i = \sum_{j \in J*} sign_j w_{ij} from a set of neurons J* do
    c_i ← c_i + inp_i
    while c_i ≥ λ do
        emit +1
        c_i ← c_i − λ
        z_i ← z_i + 1
    end

while z_i > 0 and c_i < 0 do
    emit −1
    c_i ← c_i + λ
    z_i ← z_i − 1
end
```

**Algorithm 2:** Extended counter neuron implementation based on the discretized ReLU activation. The parameter θ_i represents the neuron’s threshold. The scaling factor λ allows adjusting the step size at which the neuron emits events, i.e. how much more input is required to trigger the next event.

Using binary outputs for neurons might be a disadvantage, since due to the limited output bandwidth of individual neurons it might be necessary to use more neurons in hidden layers to obtain the same accuracy as a non-binary
network. The counter network model can easily be extended to non-binary activations, without requiring multiplication in the event-based implementation.

In order to train a network based on neurons featuring multiple, discrete levels of activation, we use a discretized version of the ReLU activation function during training:

$$f(x) = \rho\left(\left\lfloor \frac{x + \epsilon}{\lambda} \right\rfloor \right), \quad (4.7)$$

where $\epsilon \ll 1$ is a small, positive constant, $\lambda \in \mathbb{Z}^+$ is a scaling factor, and $\rho$ is the typical ReLU half-wave rectification,

$$\rho(x) = \begin{cases} x, & \text{if } x > 0, \\ 0, & \text{otherwise.} \end{cases} \quad (4.8)$$

As in the binary case, the discrete activation is used during training in the forward pass, and a continuous approximation in the backward pass. Specifically,
Numerical results

Various networks were trained on the MNIST dataset of handwritten digits to demonstrate competitive classification accuracy. In particular, fully connected networks (FCNs) of three hidden layers (784-1000-1000-1000-10) and five hidden layers (784-750-750-750-750-750-10) to investigate how the depth of the network affects the processing speed and efficiency were evaluated. In addition convolutional networks (CNNs) with two (784-12c5-12c7-10) and four (784-12c3-12c5-12c7-12c9-10) hidden layers were trained. The network dimensions were chosen such that the number of parameters remains roughly
the same in the shallower and deeper networks (≈2.8 mio. parameters for the FCNs, and ≈50k for the CNNs.)

### 4.3.1 Training details

The networks were trained through stochastic gradient descent using the Adam algorithm (Kingma & Ba, 2014). The gradients for the backward pass were computed using the continuous approximations described by eqs. (4.3) and (4.9). All parameters were restricted to 8-bit integer precision in the forward pass, and floating point representations were used only in the backward pass, as suggested by Courbariaux, Bengio & David, 2015; Stromatias et al., 2015. The biases $\theta$ were restricted to non-negative values through an additional constraint in the objective function, and categorical cross-entropy was used as the loss function. The learning rate was set to 0.01 for the CNNs and to 0.005 for the FCNs. The MNIST dataset was split into a training set of 50000 samples, a validation set of 10000 samples, and a test set of 10000 samples, and a batch size of 200 samples was used during training. The networks were trained until a classification error of ≈1.5% on the validation set was obtained. The low-precision parameters were then directly used in an event-based network of equivalent architecture.

### 4.3.2 Fast and efficient classification

The main advantage of event-based deep networks is that outputs can be obtained fast and efficiently. This is quantified in Figure 4.4, where the processing speed is measured as the time taken to produce the same output as the frame-based model, and efficiency is quantified as the number of addition operations required to compute the output. For this analysis, individual pixels of the input image are provided to the network one by one in random order. In the systems based on the basic counter network model with binary neurons, the majority of events are emitted at the beginning of the input data presentation, with activity gradually declining in the course of the presentation. This reflects the fact that basic counter neurons cannot emit two events of the same sign in a row, leading to quick saturation in the output. The opposite is the case for the extended counter neuron based on the discretized ReLU, where activity gradually ramps up. Overall, this allows the extended model to produce the desired output with fewer operations than the basic model, as can be seen in fig. 4.5. The achieved efficiency is beyond what had been possible with previous conversion-based methods: the proposed method achieves classification of MNIST at <500k
Figure 4.4 Performance of the basic and extended counter neuron models. The top panel shows the distribution of input pattern lengths (event count or active pixels) of the MNIST test set. The middle panels show the fraction of classified patterns (classified here means that the output is the same as that of the corresponding frame-based network) as a function of the number of input events presented to the network for the different architectures that were tested (mean over MNIST test set; left: FCNs; right: CNNs). The arrows mark the positions where the 99 % threshold is crossed. The bottom panels show the number of addition operations triggered in the networks by an individual input event over the course of the input presentation (mean over MNIST test set).
Operations required (mio.)

Figure 4.5 Efficiency of the basic and extended counter neuron models. The diagram shows the average number of operations required by different architectures to classify a certain fraction of test set examples. The arrows mark the positions where the 99% threshold is crossed. The color and line style correspond to the ones used in fig. 4.4.

events (CNN based on the extended neuron model), while the best reported result of a conversion-based network, to our knowledge, is >1 mio. events (Neil et al., 2016). Despite the different network architectures, the behavior of FCNs and CNNs is qualitatively similar, with the main differences being due to the neuron model. In general, deeper networks seem to require a greater number of operations than shallower networks to achieve equivalent results.

4.4 Discussion

The two presented neuron models allow efficient event-based implementations of deep neural network architectures. While previous methods constructed deep spiking networks by converting parameters and approximating activations with firing rates, the output of the proposed model is provably equivalent to its frame-based counterpart. Training is done in the frame-based domain, where state-of-the-art neural network optimization methods can be exploited. The discrete nature of counter networks allows hardware-friendly digital implementations, and makes them very suitable to process data from event-based sensors (Posch et al., 2014). The resulting systems differ from traditional neural networks in the sense that units are updated ‘depth first’, rather than ‘breadth first’, meaning that any neuron can fire when its threshold is crossed, instead of waiting for all neurons in previous layers to be updated, as in conventional neural networks. This allows processing of input data as they arrive, rather than waiting for a whole frame to be transferred to the input layer. This can significantly speed
up processing in digital applications. Compared to other deep spiking neural networks based on parameter conversion (Neil et al., 2016), counter networks require fewer operations to process input images, even in our non-optimized setting. Presumably, adding further constraints to enforce sparsity or reduce neuron activations can make counter networks even more efficient. Further research is required to investigate the applicability of the counter neuron model in recurrent networks. Finally, event-based systems are appealing because they allow for purely local, event-based weight update rules, such as spike-timing dependent plasticity (STDP), as discussed in the following chapter.
Chapter 5

Event-based training of multilayer counter networks

The previously introduced counter networks had been trained through back-propagation, where continuous approximations of the discrete elements of the model were used in the backward pass to compute the gradients. In this chapter, it is shown how counter neuron dynamics naturally implement an event-driven learning rule, operating in a similar way as spike-timing dependent plasticity (STDP) found in biological systems (Bi & Poo, 1998; Markram & Tsodyks, 1996). Thereby, no separate channel for computing and backpropagating error gradients is needed: all quantities required to compute the weight update are computed locally by the neuron itself. The approach presented here is related to a class of contrastive learning rules (Movellan, 1991; O’Reilly, 1996) and recent work on connecting energy-based models and backpropagation (Scellie & Bengio, 2016; Xie & Seung, 2003). In contrast to previous approaches, the system presented here is fully discrete, and eliminates the need for lengthy relaxation during inference.

With an update mechanism of this form, the advantages of event-based systems, in addition to the inference phase, apply to the training phase of a network, potentially leading to more efficient learning systems. In particular, the simplicity of the proposed mechanism could lead to advances in dedicated, event-based hardware for learning and predicting at a strictly limited power budget (Azghadi et al., 2014; Indiveri & Liu, 2015).

As a main result, it is shown that a hierarchical network of counter neurons descends a Lyapunov function. This result is subsequently used to derive an event-based learning rule, which only relies on locally available information.
Hierarchical counter network descends energy functional

In the following, the state of a neuron is called $x_k$, and represents the activation of a stateful binary neuron with an internal state $c_k$ or, equivalently, the sum over all output events of a counter neuron with a counter variable $c_k$. The equivalence between both models permits using counter neurons and stateful neurons with binary activations interchangeably. The stateful neurons considered here, add their inputs to their internal state, which is retained over time.

Definition 5.1.1. Let $N(x_1, \ldots, x_n)$ be a hierarchical network of stateful binary neurons, described by a weight matrix $w$ with $w_{ij} = 0$ $\forall$ $j \geq i$, and biases $b_i \geq 0$ $\forall i$.

- We call the first $n_{in}$ neurons of the network the input layer, whereby $n_{in}$ is chosen such that $w_{ij} = 0$ $\forall$ $j \leq n$, $i \leq n_{in}$.
- We define a forward pass as updating the neuron states $x_{n_{in} + 1}, \ldots, x_n$ sequentially in ascending order $n_{in} + 1, \ldots, n$.
- Analogously, a backward pass is defined as updating the units in reverse order $n, \ldots, n_{in} + 1$ while the weight matrix is replaced by its transpose, $\tilde{w} := w^T$.

Theorem 5.1.1. The states of a network as defined in def. 5.1.1 descend an energy function $F$ during a forward pass and subsequent backward pass, such that

$$F(x(t)) - F(x(t - \mu)) \leq 0 \forall t, \mu > 0, \quad (5.1)$$

where $F$ is given by

$$F(x) = - \sum_i \sum_{j \neq i} w_{ij} x_i x_j + \sum_i b_i x_i. \quad (5.2)$$

Figure 5.1 displays the energy of a simulated random network over the forward and backward pass. The decrease in energy in every update step is evident. The proof of convergence of the system is similar to that of a Hopfield network (Hopfield, 1982).

Proof of theorem 5.1.1. The neurons in the network are updated sequentially. In particular, let $t_k$ be the time at which neuron $x_k$ is updated. The energy
change at time $t = t_k$ is thus given by

$$\Delta F_k := F(x(t_k)) - F(x(t_{k-1}))$$  \hspace{1cm} (5.3)$$

$$\Delta F_k := - \sum_i x_i(t_k) \sum_{j \neq i} w_{ij} x_j(t_k) + \sum_i b_i x_i(t_k)$$ \hspace{1cm} (5.4)$$

$$+ \sum_i x_i(t_{k-1}) \sum_{j \neq i} w_{ij} x_j(t_{k-1}) - \sum_i b_i x_i(t_{k-1})$$ \hspace{1cm} (5.5)$$

$$\Delta F_k := -(x_k(t_k) - x_k(t_{k-1})) \left( \sum_{j \neq k} w_{kj} x_j(t_{k-1}) - b_k \right)$$ \hspace{1cm} (5.6)$$

$$- \sum_{i \neq k} x_i(t_k) w_{ik} x_k(t_k) + \sum_{i \neq k} x_i(t_{k-1}) w_{ik} x_k(t_{k-1})$$ \hspace{1cm} (5.7)$$

$$\Delta F_k := -(x_k(t_k) - x_k(t_{k-1})) \left( \sum_{j \neq k} (w_{kj} + w_{jk}) x_j(t_{k-1}) - b_k \right)$$ \hspace{1cm} (5.8)$$

where we have used $x_j(t_k) = x_j(t_{k-1})$ for $k \neq j$. In the forward pass, neurons $x_{n+1}, \ldots, x_n$ are updated at times $t_{n+1}^f, \ldots, t_n^f$, where $t_k^f < t_{k+1}^f \forall k$. With $b_i \geq 0 \forall i$, the sum in eq. (5.8) can be replaced by a sum over $j = 1, \ldots, k-1$, as $x_j(t_{k-1}^f) = 0$ for $j \geq k$. On the other hand, $w_{jk} = 0$ for $j < k$, such that during the forward pass eq. (5.8) can be rewritten as follows,

$$\Delta F_k^f := -x_k(t_k^f) \left( \sum_{j < k} w_{kj} x_j(t_{k-1}^f) - b_k \right). \hspace{1cm} (5.9)$$
The term in brackets on the RHS of eq. (5.9) is just the total input neuron $k$ has received so far, and corresponds to its internal state variable,

$$c_k(t^f_k) = \sum_{j<k} w_{kj} x_j(t^f_{k-1}) - b_k. \tag{5.10}$$

Since $x_k(t_f) = 1$ if and only if $c_k(t_f) > 0$, and $x_k(t_f) = 0$ otherwise, we conclude that $\Delta F^f_k \leq 0 \ \forall k$.

In the backward pass, the weight matrix is transposed, $w \mapsto \tilde{w}$ (note that the energy (5.2) is invariant under this transformation), and the update order is reversed, such that neurons $x_{n+1}, \ldots, x_n$ are updated at times $t^b_{n-1}, \ldots, t^b_1$, where $t^b_k > t^b_{k+1} > t^f_k \ \forall k$. With $\tilde{w}_{jk} = 0 \ \forall j > k$, eq. (5.8) takes the form

$$\Delta F^b_k := -\left( x_k(t^b_k) - x_k(t^b_{k-1}) \right) \left( \sum_{j\neq k} (\tilde{w}_{kj} + \tilde{w}_{jk}) x_j(t^b_{k-1}) - b_k \right) \tag{5.11}$$

$$= -\left( x_k(t^b_k) - x_k(t^b_{k-1}) \right) \left( \sum_{j>k} \tilde{w}_{kj} x_j(t^b_{k-1}) + \sum_{j<k} w_{kj} x_j(t^b_{k-1}) - b_k \right). \tag{5.12}$$

Since $x_j(t^b_{k-1}) = x_j(t^f_{k-1})$ for $j < k$, the final part of eq. (5.12) corresponds to the neuron’s internal state variable before the backward pass, given by (5.10). On the other hand, the remaining sum corresponds to the total input the neuron has received during the backward pass, such that both terms together correspond to the internal state variable after the update:

$$\Delta F^b_k = -\left( x_k(t^b_k) - x_k(t^b_{k-1}) \right) \left( \sum_{j>k} \tilde{w}_{kj} x_j(t^b_{k-1}) + c_k(t^f_k) \right) \tag{5.13}$$

$$= -\left( x_k(t^b_k) - x_k(t^b_{k-1}) \right) c_k(t^b_k). \tag{5.14}$$

With $x_k(t^b_k) - x_k(t^b_{k-1}) < 0$ only if $c_k(t^b_k) \leq 0$ we conclude that $\Delta F^b_k \leq 0 \ \forall k$. \hfill \Box

## 5.2 Contrastive update rule leads to event-based learning

In this section, it is shown how the neuron dynamics naturally lead to an update rule for the network parameters, which minimizes a given objective function. This is done in the context of supervised training, where a network is presented labeled data and learns a relation between an input layer (input data) and an
output layer (labels). Analogous to the input layer (see theorem 5.1.1), the output layer of the network is defined as the final $n_{\text{out}}$ neurons, where $w_{ij} = 0 \forall i \leq n, j \geq n-n_{\text{out}}$. The neuron update procedure described in theorem 5.1.1 can be modified to include the supervision signal (labels) as follows: after the forward phase, the output layer neurons are clamped to values $v_{n-n_{\text{out}}}, \ldots, v_n$ before the backward pass is carried out over neurons $n-n_{\text{out}}, \ldots, n_{\text{in}} + 1$.

Clamping the output layer neurons adds an offset to the system energy,

$$F^c - F^f = \sum_{i \geq n-n_{\text{out}}} (x_i(t_f^n) - v_i) \sum_{j \neq i} w_{ij} x_j(t_f^n), \quad (5.15)$$

where $F^f$ and $F^c$ denote the value of the system energy after the forward pass and after the clamping, respectively. However, the integrity of the proof of theorem 5.1.1 is not affected: every update step in the forward and backward pass still has a non-increasing effect on the energy.

The fact that the system dynamics decrease the energy permits application of a simple contrastive learning rule to update the system parameters. During training, many samples and corresponding labels from the training set are presented to the network in the way described above, each causing a small update. After the forward pass, the network is in a state $x^f$, corresponding to an energy $F^f$. Clamping the output layer neurons and running the backward pass leads to a new network state $x^b$ and corresponding energy $F^b$. The backward pass perturbs the network in such a way that its output corresponds to the target output, while the input neurons are still clamped to a given data point. We can now reshape the energy in such a way that the network state after the forward pass, $x^f$, is moved towards the target state $x^b$. As illustrated in fig. 5.2, this is done by increasing $F^f$ and lowering the energy in direction of the target state, thus minimizing

$$\Delta F := F\left((1 - \delta)x^f + \delta x^b\right) - F(x^f), \quad (5.16)$$

where the term $(1 - \delta)x^f + \delta x^b$ linearly interpolates between the states $x^f$ and $x^b$, and $\delta$ is a small constant, such that the resulting state is close to $x^f$. Alternatively, this interpolation can be written as $x^f + \delta \Delta x$, where we have defined $\Delta x = x^b - x^f$ as the difference between the forward and backward states. Note that $\Delta x_k$ corresponds to the output event ($\pm 1$) of neuron $k$ in the backward pass. Optimizing eq. (5.16) with respect to the network parameters
leads to a set of simple update rules,

\[
\Delta w_{ij} = -\lim_{h \to 0} \frac{(\Delta F(w_{ij} + h) - \Delta F(w_{ij}))}{h}
\]

\[
= \left(x_i^f + \delta \Delta x_i\right) \left(x_j^f + \delta \Delta x_j\right) - x_i^f x_j^f
\]

\[
= \delta \left(x_i^f \Delta x_j + x_j^f \Delta x_i\right) + O(\delta^2)
\]

\[
\approx \delta \left(x_i^f \Delta x_j + x_j^f \Delta x_i\right), \text{and}
\]

\[
\Delta b_i = \delta \Delta x_i.
\]

Equations (5.20) and (5.21) can be considered event-based as \(\Delta w_{ij}\) and \(\Delta b_i\) are non-zero only if \(\Delta x_i\) or \(\Delta x_j\) are non-zero, i.e. if neurons \(i\) or \(j\) produce output events in the backward pass. The update then corresponds to the product of the events (sign) and the current states of the respective neurons. Importantly, the update only relies on local quantities, namely the states and the output events of the connected neurons.

**Figure 5.2** Illustration of the update rule. The energy corresponding to the network state after the forward pass (black dot) is increased (black arrow), while the energy in direction of the target state (red dot) is lowered (red arrow), such that the forward state ‘moves’ towards the target state. The three panels show how the forward state approaches and eventually converges to the target state.

### 5.2.1 Adaptation for counter neurons

Note that in the derivation above it is assumed that in the backward pass the activation \(x_k\) of a neuron \(k\), rather than the output event \(\Delta x_k\), is multiplied by the respective weight and passed as input to neurons in the previous layer. This is not fully compatible with the counter neuron model, which normally only produce outputs when their state changes. The learning rule can be turned into a fully event-based one, where computation is triggered solely by output events of neurons, through a small modification. To derive an adapted learning rule for fully event-based counter neurons, the update procedure is reformulated as
Event-based training of multilayer counter networks follows:

1. a forward pass is run as described in definition 5.1.1, leading to a system state $x^f$,

2. the connectivity matrix is transposed, $w \mapsto \tilde{w}$, and another forward pass is run, leading to a system state $\tilde{x}^f$,

3. a backward pass is run as described in definition 5.1.1, with the modification that only changes in the activations ($\Delta x$) are communicated backwards, rather than the actual activations, resulting in a system state $\tilde{x}^b$.

It is now easy to see that the state $\tilde{x}^b$ corresponds to state $x^b$ of the original model, since

$$\tilde{x}^b_k = \sigma \left( \sum_{j < k} w_{kj} x_j^f + \sum_{j > k} \tilde{w}_{kj} (x_j^f + \Delta x_j) + b_k \right) = x^b_k. \quad (5.22)$$

Analogously to the original derivation, it can be shown that the first and the second forward passes, as well as the subsequent backward pass do not increase the system energy. Rather than updating the system parameters based on the first forward pass and the backward pass, we can use the state after the second forward pass instead, leading to the following weight update rule:

$$\Delta w_{ij} = \delta \left( \tilde{x}_i^f \Delta x_j + \tilde{x}_j^f \Delta x_i \right). \quad (5.24)$$

The update procedure described here is compatible with counter neurons, as all communication happening in the network can be expressed in terms of output events. Since the activations of all units are initially zero, in the forward pass the output events of neurons correspond to their activation values. The second forward pass can be implemented in an event-based fashion by back-propagating output events occurring during the forward pass, to neurons in the previous layer. Thus, emission of an event by neuron $k$ during the forward pass sends the value $\tilde{w}_{ik}$ to all connected neurons $i < k$, where the fact is used that output events during the forward pass can only be $+1$. The backward pass is then run by propagating the output events of neurons backwards through the
layers, and triggering the weight updates when the respective neurons become active.

5.3 Numerical results

To demonstrate the effectiveness of the learning rule, a multilayer perceptron model is trained to classify samples from the permutation-invariant MNIST handwritten digits dataset. To reduce the input dimensionality, only the 196 on average most active pixels were used. The input data was binarized by rounding to the nearest integers, and the labels were represented one-hot encoded. The learning rate was set to 1, such that weight updates were either $-1$, $0$, or $1$. The network was trained by providing samples from the training set as input to the network and propagating it through the layers. The output error was computed and provided as input to the network during the backward pass. Note that all neurons of the output layer were provided the error at the same time, such that individual layers would be updated in parallel during the backward pass. Figure 5.3 shows the reduction of the training error of the network over time. Experiments show that the method reliably works with many more layers than contained in the relatively small example system.

![Figure 5.3](image)

**Figure 5.3** Training error of a 196$-200-200-10$ network trained on the MNIST handwritten digits dataset with the event-based learning rule.

5.3.1 Improving convergence by encouraging active participation

A potential problem with training discrete networks as outlined above is that, due to the discretization, some units might never become active or stay active at all times, and therefore will not be able to propagate the error signal. To
address this issue, a mechanism termed “tune-in” here, switching on inactive units, or switching off active units when no state change has been detected for a certain number of timesteps.

![Graph](image)

**Figure 5.4** Comparison of identical networks (fully connected; 196-200-200-10) trained without (blue) and with tune-in (red) on the MNIST dataset. The network with active participation modulation converges significantly faster.

Figure 5.4 clearly shows the effects of this mechanism: the time it takes for the network to converge is roughly halved. In the presented experiments, the timeout for forced state changes was initially set to 100 sample presentations, and a schedule was implemented to increment this value by 10 after every epoch. Gradually reducing the strength of the mechanism makes sense as it turned out to be particularly helpful at the beginning of the training, whereas later on, the effects would degrade the classification accuracy.

5.4 **Discussion**

This chapter presents a contrastive learning rule for hierarchical neural networks with binary activations, which turns into an event-based update rule when the binary neurons are replaced by counter neurons. This mechanism is related to but, to our knowledge, also different from previous approaches. In particular, it is known that a contrastive update rule applied to a multilayer network with bidirectional connections in the limit of weak feedback connections is equivalent to classical backpropagation (Xie & Seung, 2003). The proposed model does not involve such constraints, as the direction of connectivity is switched between the forward and backward pass. The model can be related to a Hopfield network which is updated in a particular order (Hopfield, 1982), and is similar to a fully discretized version of “equilibrium propagation” (Scellier & Bengio, 2016),
applied to a multilayer architecture. However, due to the discrete dynamics and the directed connectivity, only a single pass of updates through the layers is required, rather than multiple iterations to find a fixed point. Moreover, in previous continuous models, the error signal decays along the layers, which is not the case in the presented discrete system.

The proposed model is considered a proof of concept and further research might be required to achieve competitive learning performance. In particular, variations of several commonly used techniques for speeding up neural network training might be applicable here, providing ground for further investigation.
PART III

Computing with imprecise and noisy substrates
Introduction

Due to their organic nature, no two cells of a biological system, and therefore no two processing units of a brain, behave exactly the same. Signals are communicated electrically, chemically, on a molecular basis, and even mechanically, through the action of ion channels, whereby all of these processes are influenced by external factors, such as temperature, the abundance of various substances, and various sources of noise (Faisal, White, et al., 2005; Shadlen & Newsome, 1994). In order to implement some form of precise computation, or even just to survive, such systems have to be able to cope with substantial amounts of variability and noise on many different timescales (Faisal, Selen, et al., 2008).

Modern microchip technology, on the other hand, due to shrinking transistor dimensions and lower supply voltages, is faced with challenges very similar to the ones brains have to deal with (Laughlin & Sejnowski, 2003). Smaller transistor sizes and lower operating currents mean that the devices are increasingly subject to quantum effects and external sources of noise, resulting in reduced signal-to-noise ratios (Shepard & Narayanan, 1997). Contemporary approaches for digital design put great efforts into alleviating these effects to some extent through careful design considerations and elaborate computational optimization and calibration techniques (Chandrakasan et al., 1992). However, as Moore’s law (Moore, 1998) continues to be met, the known methods become less and less effective, and at the same time more costly.

It therefore seems reasonable to consider alternative approaches, such as algorithms that can deal with or even benefit from the inherent variability of the physical substrate. The following chapters describe two different ways of achieving this. In chapter 6, the stochasticity due to inherent thermal noise is used as a computational resource to implement a stochastic sampler in a massively parallel network of interacting neurons, which can be used to solve high-dimensional optimization problems. Chapter 7 introduces a method for using imprecise analog hardware to implement precise neural network computation. This is done by optimizing the network configuration with measured device characteristics as constraints. Similar techniques might be applicable to future microelectronic devices which, rather than being hard-wired or programmed based on an ideal reference hardware model, would be optimized for a particular task, taking into account properties of the actual device instance.
Chapter 6

Noisy analog devices for probabilistic processing

Constraint satisfaction problems (CSPs) include some of the most prominent problems in science and engineering. A CSP is defined by a set of variables and a set of conditions (constraints) on those variables that need to be satisfied simultaneously. Solutions to a given CSP typically form a vanishingly small subset of an exponentially large search space, rendering this type of problem NP-hard in the general case. Many CSP solvers therefore involve problem-specific heuristics to avoid searching the entire space. Alternatively, one can consider a massively parallel system of simple computing elements that tackle different parts of the problem simultaneously, and discover a global solution through communicating local search mechanisms. Here, event-based neural hardware is explored as a substrate of computation, however, there are various other approaches, such as quantum annealing (Farhi et al., 2001), special cellular automata hardware (Fujita et al., 2010), or oscillator networks (Mostafa et al., 2015). The idea of using recurrent neural networks for finding solutions to computationally hard problems goes back to Hinton (Hinton, Sejnowski, et al., 1984) and Hopfield (Hopfield & Tank, 1985). While the deterministic Hopfield network fails in the general case because of local minima it can get caught in, the probabilistic Boltzmann machine proposed by Hinton, Sejnowski, et al., 1984 overcomes this limitation by sampling its states from a probability distribution rather than evolving along a deterministic trajectory.

It has been shown recently, how such stochastic samplers can be implemented in networks of spiking neurons (Buesing et al., 2011; Maass, 2014). The resulting neural sampling framework has been applied to constraint satis-
faction problems by Jonke et al., 2014, demonstrating advantages of the spiking network approach over standard Gibbs sampling in certain cases. The main technological advantage of using such neural samplers in practical applications lies in the ability to implement spiking neurons efficiently in neuromorphic hardware. However, the models proposed by Buesing et al., 2011; Jonke et al., 2014 are difficult to directly transfer onto spiking VLSI neurons, because they require individual neurons to emit spikes probabilistically, sampling from a probability distribution of a specific form. This is not easily implementable in an electronic circuit without explicit sources of noise, or random number generators. Here it is shown how a similar sampling mechanism can be implemented in a system of analog electronic integrate and fire neurons. Rather than relying on dedicated noise sources on chip, the proposed mechanism makes use of the small amount of (thermal) noise that is present in any analog electronic system to achieve the desired stochastic network dynamics. These principles are demonstrated using a standalone system, based on a reconfigurable neuromorphic processor (Qiao et al., 2015), comprising a configurable network of adaptive exponential integrate and fire neurons and synapse circuits with biophysically realistic dynamics (Chicca et al., 2014). Once programmed for a given CSP, the system will output streams of candidate solutions. Experimental results show that these samples predominantly represent configurations with no or few violated constraints.

6.1 Solving CSPs with spiking neural networks

Solving an NP-complete decision problem stochastically through sampling means transforming it into an NP-hard optimization problem, and solving it using some kind of annealing mechanism rather than sophisticated algorithmics. Thereby, a cost function is formulated such that the solutions to a given problem are transformed into optima. It is not clear whether the corresponding optimization problem is easier to solve in general, however, typically the conversion can be done in polynomial time, and in some cases the optimization problem can be parallelized more easily or more efficiently than the decision problem. Many types of problems can be transformed to simple graph structures with little effort (Lucas, 2014), and thus almost naturally map onto a network of nodes that interact through positive and negative links. In contrast to most conventional algorithmic solvers, the optimization-based approach does not depend on problem-specific heuristics, and can thus be regarded as more general.

As a first step, it is outlined how arbitrary discrete CSPs can be mapped onto
a network of neurons. We consider constraint satisfaction problems defined by a set of \( n \) discrete variables \( \{x_1, \ldots, x_n\} \) on finite domains and a set of constraints, each linking several of those variables, e.g. \((x_i = a \lor x_j = b) \land x_i \neq x_j \) etc. Without loss of generality, any such problem can be expressed in terms of binary variables by using a one-hot scheme, i.e. by representing each discrete variable \( x_i \) as a vector of binary variables \((x_{i,k})_k\), where exactly one is active at any time, \( x_i = a \iff x_{i,a} = 1, x_{i,k \neq a} = 0 \). Furthermore, a CSP can be written in conjunctive normal form, i.e. in the form \( \land_i (\lor_j l_{ij}) \), where the \( l_{ij} \) are literals (binary variables or their negations).

In a network of spiking neurons, following the models introduced in Buesing 

et al., 2011; Nessler et al., 2013, the state of each variable is represented by the spiking activity of multiple cells. There is one cell per value a variable can assume, and a cell is called active at time \( t \) if it has emitted a spike within a certain time window \([t - \tau_{\text{bin}}, t]\). The neural sampling framework (Buesing et al., 2011) describes how a network of stochastically spiking cells can generate samples from a Boltzmann distribution, where the samples are represented by network states. A problem in conjunctive normal form can be mapped to an Ising model (Choi, 2010) and therefore be solved by the spiking network sampling from the corresponding Boltzmann distribution. Alternatively, special network motifs implementing the OR relation between several variables can be used to achieve a smaller and more efficient network (Jonke et al., 2014). The energy function which defines the probability distribution of outputs, i.e. the sampled states of all variables, is designed such that solutions to the CSP occur at particularly high rates. Note that such a sampler does not know whether its current state represents a solution and will continue exploring the state space. However, potential solutions can be validated by a secondary mechanism in polynomial time, for example by testing their conformity with the pairwise interactions between cells in the network.

To illustrate the experimental results, a 4×4 Sudoku solver was implemented on spiking analog hardware. A reduced version of the standard 9×9 Sudoku problem was chosen due to the limited number of neurons available on the device. In this example problem, there are 16 variables \( x_i \in \{1, 2, 3, 4\}, i = 1, \ldots, 16 \) that are aligned in a 4×4 grid and restricted by the constraints that no two variables in a row, in a column, or in a 2×2 quadrant must assume the same value. As described above, this can be written in binary form, by introducing four binary variables for each variable \( x_i \), whereby exactly one of them must be true at any time. Whenever one cell becomes active it shuts off
A variable $x \in \{1, 2, 3, 4\}$ is one-hot encoded and its value represented by activity of one of four cells, which mutually inhibit each other. The condition $x_i \neq x_j$ can be implemented by setting negative connections between cells representing the same value. The whole Sudoku network can be implemented using constraints of the type $x_i \neq x_j$ (only a subset of the 64 cells comprising the $4 \times 4 \times 4$ cube is shown). A $9 \times 9$ Sudoku solver implemented based on the same scheme would require $9^3 = 729$ neurons.

all the others representing the same variable for a certain period by providing a strong inhibitory post-synaptic potential (IPSP) of duration $\tau_{inh}$. On the other hand, all cells in the array receive a constant excitatory input current, such that one of them will spike if none are active, ensuring that the respective variable is in a defined state at any time. The activity of a single cell is limited by the refractory period, inactivating a cell for a duration $\tau_{ref}$ immediately after a spike is emitted. The implementation of the Sudoku solver is illustrated in fig. 6.1. The constraint $x_i \neq x_j$, which exists for any two variables of the same row, column, or $2 \times 2$ quadrant, is implemented by specifying inhibitory interactions between cells representing the same value (fig. 6.1 middle), such that the two variables cannot assume the same value at the same time.

6.2 **Stochastic dynamics in an analog VLSI neural network**

In this section, a simple neuron model that can be used to describe the analog VLSI implementation of neural sampling is introduced. In principle, the stochastic spiking neurons used in previous work (Buesing et al., 2011; Jonke et al., 2014; Nessler et al., 2013) can be approximated by integrate-and-fire neurons, which are injected large amounts of noise, as proposed by Merolla, Ursell, et al.,
2010; Neftci, Das, et al., 2013; Petrovici et al., 2013. This approach, however, requires an independent noise source for every cell, and therefore cannot easily be implemented directly in hardware. Instead, we propose a mechanism that is based on conventional deterministic neuron models, and becomes stochastic through slight jitter in the duration of temporally extended pulses in analog VLSI. Such small (thermal) fluctuations are inherent to any analog electronic system, and thus can be exploited in analog hardware implementations of the proposed model.

![Figure 6.2 Abstract neuron model and hardware measurements. EPSPs (blue) are provided to a cell at times $t_1$ and $t_2$, triggering spikes and subsequent state transition of the cell into refractory period (red). Due to the variability in the duration of the refractory period and postsynaptic potentials, sometimes one or the other is longer, even though on average, they are roughly of the same length. For instance, the EPSP provided at $t_2$ lasts slightly longer than the refractory period triggered by the spike at $t_2$, leading to a second spike at $t_3$. Equivalent effects are observed for IPSPs (functionally, the refractory period is equivalent to an IPSP in our model). On the right, measurements of refractory period duration from the actual hardware are shown.](image)

As a starting point, assume simple leaky integrate-and-fire neurons that produce an output spike and remain in refractory period for a duration $\tau_{\text{ref}}$ when their membrane potential crosses a threshold $\Theta$. A spike in one cell triggers an excitatory or inhibitory postsynaptic potential (PSP) at synapses connecting it to other cells. In the simplest case, which is also at the core of previous models (Buesing et al., 2011; Nessler et al., 2013), this can be thought of as a rectangular signal of duration $\tau_{\text{inh}}$ or $\tau_{\text{exc}}$, respectively. It is assumed here that the magnitudes of these signals are large enough to either trigger a spike in the target cell almost instantaneously (for excitatory inputs), or silence the cell completely (for inhibitory inputs), such that additional excitatory inputs have no effect. Note that the refractory period can be thought of as a strong inhibitory input of a cell to itself. The stochasticity in the system is then introduced by
small amounts of noise in the duration of those PSPs and refractory periods. As a consequence, we can regard $\tau_{\text{inh}}$, $\tau_{\text{exc}}$, and $\tau_{\text{ref}}$ as mean values, and in practice the durations are jittered around those values, as shown in fig. 6.2.

As an example, assume two neurons that are coupled through inhibitory connections and are driven by a constant external current, and assume further that $\tau_{\text{ref}} \ll \tau_{\text{inh}}$. In this circuit, whichever cell became active first would keep inhibiting the other cell. This is due to the short refractory period, which lets the active neuron spike again before the IPSP it provides to the other cell ends. This network would end up in a local optimum, and would never explore the other possible state where the second cell is active. If, however, the refractory period $\tau_{\text{ref}}$ and the inhibitory pulse width $\tau_{\text{inh}}$ are of similar size, small amounts of noise in the analog system, leading to jitter in the duration of both pulses, will sometimes cause the inhibitory PSP to be longer than the refractory period, and vice versa. Such a system could indeed explore all possible states. This mechanism can be considered a kind of noise amplification or, alternatively the system can be regarded as being close to a critical point, where vanishingly small fluctuations can lead to dramatically different behavior. Intuitively, longer refractory periods cause more explorative behavior, whereas short refractory periods let the network settle into local energy minima. Note that all time constants in the system are defined relative to each other, and the relation to real time is of little relevance. Thus, extending the refractory period is equivalent to shortening the PSPs, i.e. weakening the links between nodes. In that sense, the refractory period can be regarded as a temperature parameter, that could be used in an annealing schedule to steer the dynamics of the network.

The assumptions made to construct the model are fulfilled in intrinsically noisy analog neural hardware which can, with this method, be configured such that very small fluctuations in the electronic signals can lead to large deviations in the network dynamics. The jitter in the refractory period or PSP durations is thereby introduced by thermal fluctuations in the analog signals representing those variables. As signals in analog electronics are affected by fabrication-induced variability, in practice systematic deviations will be observed between PSPs of different synapses. In order for the sampling mechanism to function properly under these conditions, the variability in PSP duration due to thermal noise needs to be at least of the same order as the variability due to fabrication-induced mismatch. If this is not the case, the distribution gets biased and might not represent the problem accurately enough. Note that this is not necessarily problematic as long as the minima are conserved. For the experimental setup,
the programmable neuromorphic device described in Qiao et al., 2015, comprising 256 integrate and fire neurons and 128k programmable synapses, was used to implement a network that evolves in real time and produces a stream of output events that can be interpreted as states of the system. The network described in fig. 6.1 was programmed into the hardware by setting the respective inhibitory connections and run by injecting small amounts of direct current into each cell. The mean values of the tunable time constants $\tau_{\text{ref}}$ and $\tau_{\text{inh}}$ were set to approximately 100 ms for all cells. The sampling rate at which the network states were evaluated was set to 10 Hz, such that the network activity was binned over 100 ms for each sample. While the system could be run much faster, the time constants were set to relatively large values to allow for visualization of the network evolution in real time. Accelerating the system is possible because both the variability due to thermal noise and the variability due to device mismatch are roughly proportional to the PSP time constants and therefore scale with reduced PSP duration.

### 6.3 Experimental results

Figure 6.3 shows representative spiking activity of the Sudoku solver network implemented and running on an analog VLSI chip. The system occasionally converges to states solving the problem (0 constraints violated), however, it is also able to escape from those local optima and explore other possible states. The “temperature” or “exploration rate”, can be controlled by tuning the neuron parameters, i.e. the IPSP duration or the length of the refractory period. In fact, it can be observed that the system is constrained to lower energy regions by lowering the temperature parameter, i.e. decreasing the refractory period.

Figure 6.4 shows the distribution of the number of constraints violated, i.e. the measure of “energy” that we intend to minimize. As expected, fundamentally different behavior is observed for the case where the refractory period is larger than the IPSP, compared to the case where it is smaller. This leads to a phase transition-like phenomenon when this threshold is crossed. For longer refractory periods, the distribution can be well fitted by an exponential function, indicating a strong concentration around the low energy states. If the refractory period is shorter than the IPSP, however, the distribution is even more concentrated around zero, and can be approximated by a double exponential function. As shown in fig. 6.5, a similar effect can be observed in the distribution of energy jumps, i.e. the difference in the number of constraints violated between
Figure 6.3  Sudoku solver on an analog VLSI chip. The panels show representative spiking activity of a subset of the neurons used in the Sudoku solver implementation (bottom), and the temporal evolution of the number of constraints violated (top) over a period of 20 s. A binary sample vector is acquired by binning the network activity over 100 ms and assigning a 1 or 0 to each cell, depending on whether it has spiked in the given interval or not. Three example states are shown, where a red cell indicates that the respective variable violates one or more constraints. The system frequently converges to states that represent solutions to the problem (0 constraints violated), but, due to the noise it is able to escape from those energy minima. Note that the solutions found at around 2 s and 16 s are not identical (3 and 4 are swapped).
Figure 6.4 Empirical analysis of the samples generated by the hardware CSP solver. The plots show histograms of the number of constraints violated, based on 10 mins of spiking activity of the Sudoku network. Solid lines are least-squares fits to the data. Slightly varying one of the network parameters (the mean refractory period, in this case) leads to fundamentally different behavior of the system. If the refractory period is longer than the IPSP, the frequency of states of a certain energy decreases exponentially with the number of constraints violated. For shorter refractory periods, the system samples almost exclusively from very low energy states, and the distribution can be described by a double exponential function. For these measurements, $\tau_{\text{inh}}$ was fixed to $\approx 110$ ms, while $\tau_{\text{ref}}$ was set to $\approx 100$ ms (red) or $\approx 120$ ms (blue).

Discussion

This work presents, to our knowledge, the first analog VLSI implementation of a CSP solver based on neural sampling with spiking neurons. At the core of the proposed mechanism is a simple neuron model that achieves stochasticity without the external noise sources required by previous approaches, but instead exploits small variations and noise in the duration of temporally extended signals. The empirical results obtained from an analog neuromorphic processor demonstrate the function and performance of the proposed method. While the hardware system used in the experiments is deliberately slowed down to operate at timescales similar to real neurons, the presented approach could, without
Figure 6.5 Empirical analysis of the dynamics of the analog VLSI Sudoku solver. The plots show the distribution of energy jumps, i.e. the difference in the number of constraints violated between two consecutive states for different network parameters. For these measurements, $\tau_{\text{inh}}$ was fixed to $\approx 110$ ms, while $\tau_{\text{ref}}$ was set to $\approx 100$ ms (red) or $\approx 120$ ms (blue).

restrictions, be used with much faster hardware to solve computationally hard problems quickly and efficiently. The duration of the refractory period, or alternatively the duration of PSPs, can be empirically related to a temperature parameter, such that those time constants could be varied in an annealing schedule to control the temperature. The work is considered a proof-of-concept and further research is required to optimize performance and analyze theoretical properties of the model.
Chapter 7

Precise computation on imprecise analog electronics

Modern information technology requires an increasing amount of computational resources to process massive volumes of data in real time. This rapidly growing need for computing power has led to the exploration of computing technologies beyond the predominant von Neumann architecture. In particular, due to the separation of memory and processing elements, traditional computing systems experience a bottleneck when dealing with problems involving great amounts of high-dimensional data (Backus, 1978; Indiveri & Liu, 2015), such as image processing, object recognition, or probabilistic inference. These problems can often best be tackled by conceptually simple but powerful and highly parallel methods, such as deep neural networks (DNNs), which in recent years have delivered state-of-the-art performance on exactly those applications (LeCun, Bengio & Hinton, 2015; Schmidhuber, 2015). DNNs are characterized by stereotypical and simple operations at each unit, of which many can be performed in parallel. For this reason they map favorably onto the processing style of graphics processing units (GPUs) (Scherer et al., 2010). The large computational demands of DNNs have simultaneously sparked interest in methods that make neural network inference faster and more power efficient, whether through new algorithmic inventions (Courbariaux, Bengio & David, 2015; Han et al., 2015; Hinton, Vinyals, et al., 2015), dedicated digital hardware implementations (Cavigelli et al., 2015; Chen, Luo, et al., 2014; Gokhale et al., 2014), or by taking inspiration from real nervous systems (Farabet et al., 2012; Indiveri, Corradi, et al., 2015; Merolla, Arthur, et al., 2014; Neil et al., 2016; O’Connor, Neil, et al., 2013).
With synchronous digital logic being the established standard of the electronics industry, first attempts towards hardware deep network accelerators have focused on this approach (Cavigelli et al., 2015; Chen, Krishna, et al., 2016; Griffin et al., 1991; Park et al., 2016). However, the massively parallel style of computation of neural networks is not reflected in the mostly serial and time-multiplexed nature of digital systems. An arguably more natural way of developing a hardware neural network emulator is to implement its computational primitives as multiple physical and parallel instances of analog computing nodes, where memory and processing elements are co-localized, and state variables are directly represented by analog currents or voltages, rather than being encoded digitally (Alspector & Allen, 1987; Andreou et al., 1991; Borgstrom et al., 1990; Rosenblatt, 1958; Satyanarayana et al., 1992; Vittoz, 1990). By directly representing neural network operations in the physical properties of silicon transistors, such analog implementations can outshine their digital counterparts in terms of simplicity, allowing for substantial advances in speed, size, and power consumption (Hasler & Marr, 2013; Masa et al., 1994). The main reason why engineers have been discouraged from following this approach is that the properties of analog circuits are affected by the physical imperfections inherent to any chip fabrication process, which can lead to significant functional differences between individual devices (Pelgrom, Duinmaijer, et al., 1989).

In this work a new approach is proposed, whereby rather than brute-force engineering more homogeneous circuits (e.g. by increasing transistor sizes and burning more power), neural network training methods are employed as an effective optimization framework to automatically compensate for the device mismatch effects of analog VLSI circuits. The diverse measured characteristics of individual VLSI devices are used as constraints in an off-line training process, to yield network configurations that are tailored to the particular analog device used, thereby compensating for the inherent variability of chip fabrication. Finally, the network parameters, in particular the synaptic weights found during the training phase can be programmed into the network, and the analog circuits can be operated at run-time in the sub-threshold regime for significantly lower power consumption.

Compact and low-power candidate VLSI circuits are proposed and used to implement an example system, demonstrating the effectiveness of the approach. A closed-loop demonstration of the framework is shown, based on a fabricated prototype chip, as well as detailed, large-scale simulations. The resulting analog electronic neural network performs as well as an ideal network, while offering
Training with heterogeneous transfer functions

A deep neural network processes input signals in a number of successive layers of neurons, where each neuron computes a weighted sum of its inputs followed by a non-linearity, such as a sigmoid or rectification. Specifically, the output of a neuron $i$ is given by $x_i = f \left( \sum_j w_{ij} x_j \right)$, where $f$ is the non-linearity, and $w_{ij}$ is the weight of the connection from neuron $j$ to neuron $i$. Thus, the basic operations comprising a neural network are summation, multiplication by scalars, and simple non-linear transformations. All of these operations can be implemented in analog electronic circuitry very efficiently, that is with very few transistors, whereby numeric values are represented by actual voltage or current values, rather than a digital code. Analog circuits are affected by fabrication mismatch, i.e. small fluctuations in the fabrication process that lead to fixed distortions of functional properties of elements on the same device, as well as multiple sources of noise. As a consequence, the response of an analog hardware neuron is slightly different for every instance of the circuit, such that $x_i = \hat{f}_i \left( \sum_j w_{ij} x_j \right)$, where $\hat{f}_i$ approximately corresponds to $f$, but is slightly different for every neuron $i$.

The weights of multi-layered networks are typically learned from labeled training data using the backpropagation algorithm (Rumelhart et al., 1986b), which minimizes the training error by computing error gradients and passing them backwards through the layers. In order for this to work in practice, the transfer function $f$ needs to be at least piece-wise differentiable, as is the case for the commonly used rectified linear unit (ReLU) (Glorot et al., 2011). Although it is common practice in neural network training, it is not necessary for all neurons to have identical activation functions $f$. In fact, having different activation functions makes no difference to backpropagation as long as their derivatives can be computed. Here this principle is exploited by inserting the heterogeneous but measured transfer curves $\hat{f}_i$ from a physical analog neural network implementation into the training algorithm, with the goal of finding weight parameters that are tailored for a particular heterogeneous system given by $\hat{f}_1, \ldots, \hat{f}_N$.

The process of implementing a target functionality in such a heterogeneous system is illustrated in Fig. 7.1. Once a neural network architecture with modifiable weights is implemented in silicon, the transfer characteristics of the dif-
Figure 7.1 Implementing and training analog electronic neural networks. a The configurable network is realized on a physical substrate by means of analog circuits, together with local memory elements that store the weight configuration. b The transfer characteristics of individual neurons are measured by applying specific stimuli to the input layer and simultaneously recording the output of the network. Repeating these measurements for different weight configurations and input patterns allows to reconstruct the individual transfer curves and fit them by a model to be used for training. c Including the measured transfer characteristics in the training process allows optimization of the network for the particular device that has been measured. d Mapping the parameters found by the training algorithm back to the device implements a neural network, whose computation is comparable to the theoretically ideal network. Arrows indicate the sequence of steps taken as well as the flow of measurement/programming data.
Different neuron instances can be measured by controlling the inputs specific cells receive and recording their output at the same time (see Methods). If the transfer curves are sufficiently simple (depending on the actual implemented analog neuron circuit), a small number of discrete measurements yield sufficient information to fit a continuous, (piece-wise) differentiable model to the hardware response. For instance, the rectified linear neuron $f(r) = \max\{0, a \cdot r\}$ is fully described by a single parameter $a$, which is simply the ratio of output to input, and therefore can easily be measured. The continuous, parameterized description is then used by the training algorithm, which is run on traditional computing hardware, such as CPUs or GPUs, to generate a network configuration that is tailored to the particular task and the physical device that has been characterized.

7.2 Analog circuit implementation

To achieve a compact and low-power solution, a multilayer network is constructed using the circuits shown in Fig. 7.2 and operate them in the subthreshold region. The subthreshold current of a transistor is exponential in the gate voltage, rather than polynomial as is the case for above threshold operation, and can span many orders of magnitude. Thus, a system based on this technology can be operated at orders of magnitude lower currents than a digital one. In turn, this means that the device mismatch arising due to imperfections in the fabrication process can have an exponentially larger impact. Fortunately, as the proposed method neither depends on the specific form nor the magnitude of the mismatch, it can handle a wide variety of mismatch conditions.

As a demonstration of the framework, a feed-forward network is implemented in which every neuron consists of one soma and multiple synapse circuits, and trained for different classification tasks. Multiple layers of soma circuits are connected through matrices of synapse circuits. A soma circuit (Fig. 7.2a) takes a current as input and communicates its output in terms of voltages, which are passed as input signals to a row of synapse circuits. A synapse circuit (Fig. 7.2b), in turn, provides a current as output, such that the outputs of a column of synapses can be summed up simply by connecting them together. The resulting current is then fed as an input current to the somata of the next layer. The first transistor of the soma circuit rectifies the input current. The remaining elements of the soma circuit, together with a connected synapse circuit, form a set of scaling current mirrors, i.e. rudimentary amplifiers, a subset of which can be switched on or off to achieve a particular weight value by setting
Figure 7.2 Basic current-mode analog circuits used to construct a multilayer neural network. A network is constructed by connecting layers of soma circuits through matrices of synapse circuits. The output of a soma circuit is communicated as a voltage and passed to a row of synapse circuits, implementing multiplications by scalars. The output of a synapse is a current, such that the outputs of a column of synapses can be summed up by simply connecting them through wires. The summed current is then passed as input to a soma of the next layer, which implements the non-linearity. 

a) Proposed soma circuit, taking a current as input and providing two output voltages $V_n$ and $V_p$, which in the subthreshold region are proportional to the log-transformed, rectified input current.

b) Proposed programmable synapse circuit with 3 bit precision, taking voltages $V_n$ and $V_p$ as inputs and providing an output current corresponding to an amplified version of the rectified soma input current, where the gain is set by the digital signals $w_\pm, w_i$. 
the respective synapse configuration bits. Thus, the output of a synapse corre-
sponds to a scaled version of the rectified input current of the soma, similar to
the ReLU transfer function.

In the proposed example implementation signed 3-bit synapses are used,
which are based on $2 \times 3$ current mirrors of different dimensions (3 for positive
and 3 for negative values). One of $2^4$ possible weight values is then selected
by switching the respective current mirrors on or off. The scaling factor of a
particular current mirror, and thus its contribution to the total weight value,
is proportional to the ratio of the widths of the two transistors forming it. The
weight configuration of an individual synapse can be stored digitally in memory
elements that are part of the actual synapse circuit. Thus, in contrast to digital
processing systems, our circuit computes in memory and thereby avoids the
bottleneck of expensive data transfer between memory and processing elements.

Although this is just one out of many possible analog circuit implementations,
the simple circuits chosen offer several advantages besides the fact that they
can be implemented in small areas: First, numeric values are conveyed only
through current mirrors, and therefore are temperature-independent. Second,
most of the fabrication-induced variability is due to the devices in the soma
with five consecutive transistors, whereas only one layer of transistors affects
the signal in the synapse. This means that the synapse-induced mismatch can
be neglected in a first order approximation.

Once an analog electronic neural network has been implemented physically
as a VLSI device, the transfer characteristics of the individual soma circuits are
obtained through measurements. The transfer function implemented by the
proposed circuits can be well described by a rectified linear curve, where the
only free parameter is the slope, and thus can be determined from a single
measurement per neuron. Specifically, the transfer curves of all neurons in a
layer $k$ can be measured through a simple procedure: A single neuron in layer $k - 1$
is connected, potentially through some intermediate neurons, to the input layer
and is defined to be the ‘source’. Similarly, a neuron in layer $k + 1$ is connected,
potentially through intermediate neurons, to the output layer and is called the
‘monitor’. All neurons of layer $k$ can now be probed individually using the
source and monitor neurons, whereby the signal to the input layer is held fixed
and the signal recorded at the output layer is proportional to the slope of the
measured neuron. Note that the absolute scale of the responses is not relevant,
i.e. only the relative scale within one layer matters, as the output of individual
layers can be scaled arbitrarily without altering the network function. The same
procedure can be applied to all layers to obtain a complete characterization of
the network. The measurements can be parallelized by defining multiple source
and monitor neurons per measurement to probe several neurons in one layer
simultaneously, or by introducing additional readout circuitry between layers
to measure multiple layers simultaneously.

7.3 Accurate inference on mismatched hardware

To assess the effectiveness and flexibility of the proposed method, behavioral sim-
ulations of networks constructed based on the circuits described in section 7.2
were run for a number of different scenarios. Specifically, multilayer perceptron
architectures of dimensions 196 – 200 – 200 – 10 were trained for a classifi-
cation task on the MNIST handwritten digits dataset (see training details in
section 7.7.4). Thereby, the neurons were modeled as rectified linear functions,
whereby the slope of each neuron’s response curve was drawn from a proba-
bility distribution. Since the variability in the scaling factor of a sub-threshold
current mirror is described by a log-normal distribution, the modeled slopes
of the activations were drawn from such a distribution, whereby the standard
deviation of the underlying normal distribution is referred to as $\sigma_{\text{act}}$ in the fol-
lowing. In the same way, since a synapse circuit represents just a set of current
mirrors, the variability in the weights is described by a log-normal distribution
with an underlying normal distribution, described by a standard deviation $\sigma_{\text{wgt}}$.
In all experiments, the weight resolution was limited to 3 bits plus a sign per
weight, corresponding to the circuits described in section 7.2. Note that during
training only the activations were modeled explicitly, whereas the variability
in the weights was represented as noise in the weight values with only their
statistics corresponding to the actual simulated device. This is supposedly a
more realistic scenario than including the actual synaptic values as constraints
in the training, as they would be rather expensive to determine in a real world
setting.

Figure 7.3 shows the network performance (classification error on the test
set) for a range of different mismatch conditions. In fig. 7.3a, both the mismatch
in the activations and the weights was modeled, whereby the actual slopes of
the variations were used, and $\sigma_{\text{wgt}}$ was fixed to 0.4 in the model used for
training. The results show that the method is able to deal with a wide range
of mismatch values in the activations and weights. As can be expected, more
variability can be tolerated in the explicitly modeled activations, whereas a
Figure 7.3 Analysis of the training method for different mismatch conditions. Each point corresponds to the classification error of a 196–200–200–10 network on the MNIST test set (averaged over 10 trained networks). The variations in the activations (soma circuits) are fixed before training and the resulting slopes are used by the training algorithm. The mismatch in the synapses is not modeled explicitly, and is fixed only after training, during the testing phase. The worst-case scenario (greatest mismatch) is marked by crosses for the simulated circuit (see section 7.4), and by circles for the fabricated prototype chip (see section 7.5) for the smallest and largest weight value, respectively.

A decline in performance is observed if the synaptic mismatch becomes very large. In general, the performance is reduced if only the mismatch in the activations (fig. 7.3b) or only the synaptic mismatch (fig. 7.3c) is modeled. Note that, for the sake of simplicity, in the following sections the synaptic mismatch is not included in the model used for training, i.e. $\sigma_{wgt}$ is assumed to be 0.

### 7.4 Handwritten and spoken digit classification

Large-scale SPICE simulations of systems consisting of hundreds of thousands of transistors are employed to assess power consumption, processing speed, and the accuracy of the proposed analog implementation.

After simulating measurements and parameterizing the transfer characteristics of the circuits as described previously, software networks were trained on
Chapter 7  Precise computation on imprecise analog electronics

Figure 7.4  Analog circuit dynamics allow classification within microseconds. The curves represent the activities (currents) of all hidden (top) and output (bottom) units of the $196 - 50 - 10$ network shown on the left. When a new input symbol is presented (top), the circuit converges to its new state within microseconds. Only a few units remain active, while many tend to zero, such that their soma circuits and connected synapses dissipate very little power.

the MNIST dataset of handwritten digits (LeCun, Cortes, et al., 1998) and the TIDIGITS dataset of spoken digits (Leonard & Doddington, 1993) by means of the ADAM training method (Kingma & Ba, 2014). In order to optimize the network for the use of discrete weights in the synaptic circuits dual-copy rounding (Courbariaux, Bengio & David, 2014; Stromatias et al., 2015) was used (see Methods). By evaluating the responses of the simulated circuit on subsets of the respective test sets, its classification accuracy was found to be comparable to the abstract software neural network (see Tab. 7.1 for comparison). Fig. 7.4 shows how inputs are processed by a small example circuit implementing a $196 - 50 - 10$ network, containing around $10k$ synapses and over $100k$ transistors. Starting with the presentation of an input pattern in the top layer, where currents are proportional to input stimulus intensity, the higher layers react almost instantaneously and provide the correct classification, i.e. the index of the maximally active output unit, within a few microseconds. After a switch of input patterns, the signals quickly propagate through the network and the outputs
Figure 7.5 Processing performance of a network for handwritten digit classification. All data shown was generated by presenting 500 different input patterns from the MNIST test set to a trained 196–100–50–10 network with the average input current per input neuron set to 15 nA (blue) or 45 nA (orange), respectively. The time to output is plotted against the average power dissipated over the duration of the transient (from start of the input pattern until time to output). The distributions of the data points are indicated by histograms on the sides. Changing the input current causes a shift along the equi-efficiency lines, that is, the network can be run slower or faster at the same efficiency (energy per operation).

of different nodes converge to their asymptotic values. The time it takes the circuit to converge to its final output defines the ‘time to output’, constraining the maximum frequency at which input patterns can be presented and evaluated correctly. Measured convergence times are summarized in Fig. 7.5 for different patterns from the MNIST test set, and are found to be in the range of microseconds for a trained 196–100–50–10 network, containing over 25k synapses and around 280k transistors. Note that observed timescale is not fixed as the network can be run faster or slower by changing the input current, while the average energy dissipated per operation remains roughly constant.

The processing efficiency of the system (energy per operation) was com-
I_{avg} = \frac{nA}{5}$

Energy/op. (pJ) = \frac{I_{avg}}{5}$

Figure 7.6  a Energy dissipated per operation for different run times, corresponding to different fixed rates at which inputs are presented (mean over 500 samples; standard deviation indicated by shaded areas). b The average energy consumed per operation was computed from the data shown in a. The data corresponds to the hypothetical case were the network would be stopped as soon as the correct output is reached.

computed for different input patterns by integrating the power dissipated between the time at which the input pattern was switched and the time to output. Fig. 7.5 shows the processing efficiency for the same network with different input examples and under different operating currents. With the average input currents scaled to either 15 or 45 nA per neuron respectively, the network takes several microseconds to converge and consumes tens or hundreds of microwatts in total, which amounts to a few nanowatts per multiply-accumulate operation. With the supply voltage set to 1.8 V, this corresponds to less than 0.1 pJ per operation in most cases. With the average input current set to 15 nA per neuron, the network produces the correct output within 15 µs in over 99% of all cases (mean 8.5 µs; std. 2.3 µs). Running the circuit for 15 µs requires 0.12 ± 0.01 pJ per operation, such that about 1.7 trillion multiply-accumulate operations can be computed per second at a power budget of around 200 µW if input patterns are presented at a rate of 66 kHz. Without major optimizations to either process or implementation, this leads to an efficiency of around 8 TOp/J, to our knowledge a performance at least four times greater than that achieved by digital single-purpose neural network accelerators in similar scenarios (Cavigelli et al., 2015; Park et al., 2016). General purpose digital systems are far behind such specialized systems in terms of efficiency, with the latest GPU generation achieving around 0.05 TOp/J (NVIDIA, 2016).
Table 7.1 Classification accuracy and power-efficiency of a $196 - 100 - 50 - 10$ network trained on the MNIST and TIDIGITS datasets. The classification accuracies of the behavioral models of the ideal as well as the inhomogeneous systems are averaged over 10 networks trained with different initializations. The parameters of the best performing one out of the 10 networks were used in the $\text{SPICE}$ circuit simulations. As detailed circuit simulations are computationally expensive, subsets of the actual test sets were used to compute the classification accuracy of the simulated circuits (the first 500 samples from the MNIST test set; 500 random samples from the TIDIGITS test set).

<table>
<thead>
<tr>
<th></th>
<th>14 × 14 MNIST</th>
<th>TIDIGITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homogeneous model</td>
<td>97.6 ± 0.1 / 98.0</td>
<td>87.3 ± 4.2 / 93.4</td>
</tr>
<tr>
<td>(mean / best acc. in %)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inhomogeneous model</td>
<td>97.6 ± 0.2 / 98.0</td>
<td>88.0 ± 3.8 / 94.3</td>
</tr>
<tr>
<td>(mean / best acc. in %)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{SPICE}$ simulation acc. (%)</td>
<td>98.0</td>
<td>94.6</td>
</tr>
<tr>
<td>Energy-efficiency (TOp/J)</td>
<td>7.97</td>
<td>6.39</td>
</tr>
</tbody>
</table>

Tab. 7.1 summarizes the classification accuracy for different architectures and datasets for a software simulation of an ideal network without mismatch, a behavioral simulation of the inhomogeneous system with the parameterized transfer curves implemented in an abstract software model, and the full circuit simulation of the inhomogeneous hardware network. Additionally, the computed power efficiency is shown for the different architectures.

7.5 VLSI implementation

As a closed-loop demonstration of our framework, a prototype VLSI chip was fabricated and trained for a classification task. A design based on the circuits shown in Fig. 7.2, containing three layers of seven neurons each, was fabricated in 180 nm CMOS technology. After characterizing the individual neuron circuits through measurements as described in Sect. 7.2 (extracted parameters shown in fig. 7.7b), a $4 - 7 - 3$ network was trained on 80% of the Iris flower dataset (Fisher, 1936), the device was programmed with the resulting parameters, and the remaining 20% of the data were used to test the classification performance. The hardware implementation was able to classify 100% of the test data correctly (see Fig. 7.7d for the output of the network).
**Figure 7.7** Running a classification task on the prototype VLSI implementation. a Photograph of the fabricated device. The neural network is a small block at the center of the chip. b Measured slopes of all $3 \times 7$ neurons of the prototype device (means and standard deviations; slopes normalized per layer). c Measurements of a single neuron (red; corresponding to the marked point in b) and the line fitted to the measurements (black).

**Figure 7.8** Running a classification task on the prototype VLSI implementation. a Visualization of the $4 - 7 - 3$ network which was implemented and trained on the Iris flower dataset (positive weights are displayed in gray, negative ones in red; line thickness corresponds to weight value). b Correct classification of the test set performed by the programmed chip (responses of the three output neurons normalized to 100%, displayed in barycentric coordinates; dot color represents the target class).
7.6 **Discussion**

The theory of analog neural networks and electronic realizations thereof have a rich history that goes back to the 1950s (Alspector & Allen, 1987; Rosenblatt, 1958). However, the demonstrated accuracy of the electronic networks is typically below the theoretical performance and therefore, their full low-power potential was never fully leveraged.

Instead, digital designs have flourished in the interim and almost all current deep network designs are implemented in digital form (Cavigelli *et al.*, 2015; Chen, Krishna, *et al.*, 2016; Park *et al.*, 2016). Although small transistors are possible in digital implementations, the typical size of a multiplier-accumulator (MAC) block usually means that these implementations use a smaller subset of functional blocks and therefore the use of MACs is time-multiplexed by shifting data around. As a consequence, the processing speed of digital implementations is limited by their clock frequency.

The simplicity of the analog VLSI circuits needed for addition - namely connecting together wires - allows for an explicit implementation of each processing unit or neuron where no element is shared or time-multiplexed within the network implementation. The resulting VLSI network is maximally parallel and eliminates the bottleneck of transferring data between memory and processing elements. Using digital technology, such fully parallel implementations would quickly become prohibitively large due to the much greater circuit complexity of digital processing elements. While the focus in this work has been on an efficient analog VLSI implementation, hardware implementations using new forms of nano devices can also benefit from this training method. For example, memristive computing technology which is currently being pursued for implementing large-scale cognitive neuromorphic and other technologies still suffers from the mismatch of fabricated devices (Ambrogio *et al.*, 2013; Kim *et al.*, 2011; Prezioso *et al.*, 2015). The training method proposed here can be used to account for device non-idealities in this technology (Niu *et al.*, 2010).

In fact, any system that can be properly characterized and has configurable elements stands to benefit from this approach. For example, spike-based neuromorphic systems (Indiveri, Linares-Barranco, *et al.*, 2011) often have configurable weights between neurons. Similar to the method outlined in this work, the relationship between an input spike rate and an output spike rate of a neuron can be measured in such a system, and the transfer functions then used as constraints during the training process so as to achieve accurate results from...
the whole network even if the neuron circuits themselves are varied and non-ideal. In addition to the alternate hardware implementations, other network topologies such as convolutional networks can be trained using the proposed method. However, as all weights are implemented explicitly in silicon, the system design here would not benefit from the small memory footprint achieved via weight sharing in traditional convolutional network implementations. In principle, even recurrent architectures such as LSTM networks (Hochreiter & Schmidhuber, 1997) can be trained using the same methods, where not only the static properties of the circuit are taken into account but also their dynamics.

With every device requiring an individual training procedure, an open question is how the per-device training time can be reduced. Initializing the network to a pre-trained ideal network, which is then fine-tuned for the particular devices is likely to reduce training time.

In the current setting, the efficiency of our system is limited by the worst-case per-example runtime, i.e. there may be a few samples where outputs require significantly longer to converge to the correct classification result than the majority. This can lead to unnecessarily long presentation times for many samples, causing unnecessary power consumption. Smart methods of estimating presentation times from the input data could accelerate convergence for slowly converging samples by using higher input currents, and conversely, faster samples could be slowed down to lower the variability of convergence times and overall reduce energy consumption. Future research will focus on such estimators, and alternatively explore ways of reducing convergence time variability during network training.

This proof-of-principle study is an important step towards the construction of large scale, possibly ultra-low-power analog VLSI deep neural network processors, paving the way for specialized applications which had not been feasible before due to speed or power constraints. Small, efficient implementations could allow autonomous systems to achieve almost immediate reaction times under strict power limitations. Scaled-up versions can allow for substantially more efficient processing in data centers, allowing for a greatly reduced energy footprint or permitting substantially more data to be effectively processed. Conversely, digital approaches and GPU technology are aiming for general purpose deep network acceleration, and thus naturally have an advantage in terms of flexibility compared to the fixed physical implementation of the proposed analog devices. However, there is increasing evidence that neural networks pre-trained on large datasets such as ImageNet provide excellent generic feature detectors
(Donahue et al., 2014; Razavian et al., 2014), which means that fast and efficient analog input pre-processors could be used as an important building blocks for a large variety of applications.

### 7.7 Implementation details

#### 7.7.1 Description of the example circuit

The example networks described in Sect. 7.2 have been implemented based on the circuits shown in Fig. 7.2. With $M_0$ as a diode-connected nFET, the soma circuit essentially performs a rectification of the input current $I_{in}$. Further, the current is copied to $M_1$ and, through $M_2$ and $M_3$, also to $M_4$, such that $M_2$ together with pFETs from connected synapse circuits, as well as $M_4$ together with nFETs from connected synapse circuits form scaling current mirrors, generating scaled copies of the rectified input current $I_{in}$. The scaling factor is thereby determined by the dimensions of $M_{10}$ to $M_{15}$. The transistors $M_{16}$ to $M_{20}$ operate as switches and are controlled by the digital signals $w_{\pm}, w_0, w_1,$ and $w_2$. The value of $w_{\pm}$ determines whether the positive branch (pFETs $M_{13}$ to $M_{15}$; adding current to the node $I_{out}$) or the negative branch (nFETs $M_{10}$ to $M_{12}$; subtracting current from the node $I_{out}$) is switched on and thereby the sign of the synaptic multiplication factor. Setting $w_0, w_1,$ and $w_2$ allows switching on or off specific contributions to the output current. In the example implementation the widths of $M_{10}$ to $M_{12}$, and $M_{13}$ to $M_{15}$, respectively, were scaled by powers of 2 (see Tab. 7.2), such that a synapse would implement a multiplication by a factor approximately corresponding to the binary value of $(w_0, w_1, w_2)$. While our results are based on a signed 3-bit version of the circuit, arbitrary precision can be implemented by changing the number of scaling transistors and corresponding switches. The dimensions of $M_3$ and $M_4$ were adjusted such that the currents through transistors of the positive and the negative branch of one particular bit of a synapse were roughly matched when switched on.

Multilayer networks were constructed using the circuits described above by connecting layers of soma circuits through matrices made up of synapse circuits. The first stage of a network constructed in this way thereby is a layer of soma circuits, rather than a weight matrix, as is typically the case in artificial neural network implementations. This is because we prefer to provide input currents rather than voltages and only soma circuits take currents as inputs. As a consequence, due to the rectification, our network can not handle negative
Table 7.2  Transistor dimensions used in all circuit simulations.

<table>
<thead>
<tr>
<th>Device</th>
<th>W (µm)</th>
<th>L (µm)</th>
<th>W/L</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₀−M₄</td>
<td>2.7</td>
<td>0.45</td>
<td>6</td>
</tr>
<tr>
<td>M₁₀, M₁₃</td>
<td>0.27</td>
<td>0.54</td>
<td>0.5</td>
</tr>
<tr>
<td>M₁₁, M₁₄</td>
<td>0.54</td>
<td>0.54</td>
<td>1</td>
</tr>
<tr>
<td>M₁₂, M₁₅</td>
<td>1.08</td>
<td>0.54</td>
<td>2</td>
</tr>
<tr>
<td>M₁₆−M₂₀</td>
<td>0.54</td>
<td>0.54</td>
<td>1</td>
</tr>
</tbody>
</table>

input signals. To obtain current outputs rather than voltages, one synapse is connected to each unit of the output layer and its weight set to 1 to convert the output voltages to currents.

7.7.2  Circuit simulation details

All circuits were simulated using Ngspice release 26 and Bsim3 version 3.3.0 models of a TSMC 180 nm process. The SPICE netlist for a particular network was generated using custom Python software and then passed to Ngspice for DC and transient simulations. Input patterns were provided to the input layer by current sources fixed to the respective values. The parameters from Tab. 7.2 were used in all simulations and Vdd was set to 1.8 V. Synapses were configured by setting their respective configuration bits $w_1$, $w_0$, $w_1$, and $w_2$ to either Vdd or ground, emulating a digital memory element. The parasitic capacitances and resistances to be found in an implementation of our circuits were estimated from post-layout simulations of single soma and synapse cells. The main slowdown of the circuit can be attributed to the parasitic capacitances of the synapses, which were found to amount to 11 fF per synapse.

Individual hardware instances of our system were simulated by randomly assigning small deviations to all transistors of the circuit. Since the exact nature of mismatch is not relevant for our main result (our training method compensates for any kind of deviation, regardless of its cause), the simple but common method of threshold matching was applied to introduce device-to-device deviations (Lakshmikumar et al., 1986). Specifically, for every device, a shift in threshold voltage was drawn from a Gaussian distribution with zero mean and standard deviation $\sigma_{\Delta V_T} = \Delta V_T / \sqrt{W/L}$, where the proportionality constant $\Delta V_T$ was set to 3.3 mV/µm, approximately corresponding to measurements from a 180 nm process (Pelgrom, Tuinhout, et al., 1998).
7.7.3 Characterization of the simulated circuit

Figure 7.9 Illustration of the measurement procedure applied to the simulated circuits. The diagram shows one possible weight configuration that might come up during the parameter extraction procedure of a network with one input, one hidden, and one output layer. Circles represent soma circuits and squares synapse circuits. Voltages are represented by double lines, whereas currents are represented by single lines. Only synapses set to non-zero values are shown. Every unit receives exactly one input signal, and produces, together with a connected synapse circuit, at maximum one output current, which can be measured as the input to a unit of the consecutive layer. The input to the network is provided in terms of a set of input currents, the output is transformed to currents by means of an additional array of synapses after the last layer.

To determine the transfer curves of individual neurons, the input-output relations of the respective soma circuits need to be measured. To save simulation time, a parallel measurement scheme was applied, based on the assumption that each neuron can be measured directly, rather than just the neurons in the output layer. Rather than measuring the log domain output voltages $V_n$ and $V_p$ we chose to record the input currents $I_{in}$ to subsequent layers. The advantages of this approach are that quantities are not log-transformed and that potential distortions arising from the synapse circuits are taken into account. Furthermore, with this method only one probe is required per neuron, rather than two separate ones for in- and output signals. Moreover, the unit weight of a synapse (which is not know a priori) here becomes a property of the soma, so that weights are automatically normalized. To determine the transfer curves of the units in the different layers the weights were set to a number of different configurations and the input currents to the various units were measured for different input patterns provided to the network. Specifically, by setting the respective synapse circuits to their maximum value, every unit was configured to receive input from exactly one unit of the previous layer. One such config-
uration is shown in Fig. 7.9. The input currents to all units of the input layer were then set to the same value and the inputs to the units of the deeper layers were recorded. By generating many such connectivity patterns by permuting the connectivity matrix, and setting the input currents to different values, multiple data points (input-output relations) were recorded for each unit, such that continuous transfer curves could be fitted to the data. For the example networks described in Sect. 7.2, 40 measurements turned out to be sufficient, resulting in roughly 10 data points per unit. Rectified linear functions $f(r) = \max\{0, a \cdot r\}$ were fitted to the data and the resulting parameters $a$ were used as part of the training algorithm. The parameters were normalized layer-wise to a mean slope of 1. Even though the sizes of the transistors implementing the positive and negative weight contributions are identical, their responses are not matched. To characterize their relative contributions, inputs were given to neurons through positive and negative connections simultaneously. Comparing the neuron response to its response with the negative connection switched off allows to infer the strength of the unit negative weight, which can then be used in the training algorithm.

### 7.7.4 Training and evaluation details

The networks were trained on the MNIST and TIDIGITS datasets using the ADAM optimizer (Kingma & Ba, 2014) and the mean squared error as loss function. The low-precision training (three signed bits per synapse) was done using a high-precision store and low-precision activations in the manner of the method simultaneously described by Stromatias et al., 2015 and Courbariaux, Bengio & David, 2014. To reduce the number of negative inputs to neurons, an L1 regularization scheme was applied to negative weights only, as they would slow down the circuits. The Keras software toolkit (Chollet, 2015) was used to perform the training. A custom layer consisting of the parameterized activation function $f(x) = \max\{0, a \cdot Wx\}$, using the extracted parameter $a$ was added and used to model the neuron activation function. Different sets of empirically found hyperparameters were used during training for the MNIST and TIDIGITS datasets. A reduced resolution version ($14 \times 14$ pixels) of the MNIST dataset was generated by identifying the 196 most active pixels (highest average value) in the dataset and only using those as input to the network. The single images were normalized to a mean pixel value of 0.04. The learning rate was set to 0.0065, the L1 penalty for negative weights was set to $10^{-6}$, and
the networks were trained for 50 epochs with batch sizes of 200. Each spoken digit of the TIDIGITS dataset was converted to 12 mel-spectrum cepstral coefficients (MFCCs) per time slice, with a maximum frequency of 8 kHz and a minimum frequency of 0 kHz, using 2048 FFT points and a skip duration of 1536 samples. To convert the variable-length TIDIGITS data to a fixed-size input, the input was padded to a maximum length of 11 time slices, forming a 12x11 input for each digit. First derivative and second derivatives of the MFCCs were not used. To increase robustness, a stretch factor was applied, changing the skip duration of the MFCCs by a factor of 0.8, 0.9, 1.0, 1.1, and 1.3, allowing fewer or more columns of data per example, as this was found to increase accuracy and model robustness. A selection of hyperparameters for the MFCCs were evaluated, with these as the most successful. The resulting dataset was scaled pixel-wise to values between 0 and 1. Individual samples were then scaled to yield a mean value of 0.03. The networks were trained for 512 epochs on batches of size 200 with the learning rate set to 0.0073, and the L1 penalty to $10^{-6}$.

### 7.7.5 Performance measurements

The accuracy of the abstract software model was determined after training by running the respective test sets through the network. Due to prohibitively long simulation times, only subsets of the respective test sets were used to determine the accuracy of the SPICE-simulated circuits. Specifically, the first 500 samples of the MNIST test set and 500 randomly picked samples from the TIDIGITS test set were used to obtain an estimate of the classification accuracy of the simulated circuits. The data was presented to the networks in terms of currents, by connecting current sources to the $I_{in}$ nodes of the input layer. Individual samples were scaled to yield mean input currents of 15 nA or 45 nA per pixel, respectively. The time to output for a particular pattern was computed by applying one (random) input pattern from the test set and then, once the circuit had converged to a steady state, replaced by the input pattern to be tested. In this way, the more realistic scenario of a transition between two patterns is simulated, rather than a ‘switching on’ of the circuit. The transient analysis was run for 7 µs and 15 µs with the mean input strength set to 45 nA and 15 nA, respectively, and a maximum step size of 20 ns. At any point in time, the output class of the network was defined as the index of the output layer unit that was the most active. The time to output for each
pair of input patterns was determined by checking at which time the output class of the network corresponded to its asymptotic state (determined through an operating point analysis of the circuit with the input pattern applied) and would not change anymore. The energy consumed by the network in a period of time was computed by integrating the current dissipated by the circuit over the decision time and multiplying it by the value of $V_{dd}$ (1.8 V in all simulations).

### 7.7.6 VLSI prototype implementation

A $7 \times 7 \times 7$ network, consisting of 21 neurons and 98 synapses was fabricated in 180 nm CMOS technology (AMS 1P6M). The input currents were provided through custom bias generators, optimized for sub-threshold operation (Delbruck et al., 2010). Custom current-to-frequency converters were used to read out the outputs of neurons and send them off chip in terms of inter-event intervals. The weight parameters were stored on the device in latches, directly connected to the configuration lines of the synapse circuits. Custom digital logic was implemented on the chip for programming biases, weights, and monitors. Furthermore, the chip was connected to a PC, through a Xilinx Spartan 6 FPGA containing custom interfacing logic and a Cypress FX2 device providing a USB interface. Custom software routines were implemented to communicate with the chip and carry out the experiments. The fabricated VLSI chip was characterized through measurements as described in Sect. 7.2, by probing individual neurons one by one. The measurements were repeated several times through different source and monitor neurons for each neuron to be characterized to average out mismatch effects arising from the synapse or readout circuits. The mean values of the measured slopes were used in a software model to train a network on the Iris flower dataset. The Iris dataset was randomly split into 120 and 30 samples used for training and testing, respectively. The resulting weight parameters were programmed into the chip and individual samples of the dataset were presented to the network in terms of currents scaled to values between 0 and 325 nA. The index of the maximally active output unit was used as the output label of the network and to compute the classification accuracy.
Bibliography


44. Fisher, R. Iris Flower Data Set 1936.


47. Galvani, L. & Aldini, G. De Viribus Electricitatis In Motu Musculari Commentarius Cum Joannis Aldini Dissertatione Et Notis; Accesserunt Epistolae ad animalis electricitatis theoriam pertinentes (Apud Societatem Typographiam, 1792).


140. Werbos, P. *Beyond regression: New tools for prediction and analysis in the behavioral sciences* PhD thesis (Harvard University, 1974).