Chapter 13

# IMPLEMENTING NEURAL MODELS IN SILICON

*Leslie S. Smith*
University of Stirling

**Abstract**

Neural models are used in both computational neuroscience and in pattern recognition. The aim of the first is understanding of real neural systems, and of the second is gaining better, possibly brainlike performance for systems being built. In both cases, the highly parallel nature of the neural system contrasts with the sequential nature of computer systems, resulting in slow and complex simulation software. More direct implementation in hardware (whether digital or analogue) holds out the promise of faster emulation both because hardware implementation is inherently faster than software and the operation is much more parallel. There are costs to this: modifying the system (for example, to test out variants of the system) is much harder when a full application-specific integrated circuit has been built. Fast emulation can permit direct incorporation of a neural model into a system, permitting real-time input and output. Appropriate selection of implementation technology can help to make simplify interfacing the system to external devices. We review the technologies involved and discuss some example systems.

## 1 WHY IMPLEMENT NEURAL MODELS IN SILICON?

There are two primary reasons for implementing neural models: one is to attempt to gain better and possibly brainlike performance for some system, and the other is to study how some particular neural model performs. Current computer systems do not approach brainlike system performance in many areas (sensing, motor control, and pattern recognition, for example, to say nothing of the higher level capabilities of mammalian brains). There has been considerable research into how the neural system attains its capabilities. Implementing neural systems in silicon can permit direct applications of this research by permitting neural models to run rapidly enough to be applied directly to data. It is true that

increases in workstation performance have allowed some software implementations of neural models to run in real time, but the highly parallel nature of neural systems, coupled with increasing interest in the application of more sophisticated (and computationally more expensive) neural models, has caused interest in more direct implementation to be maintained. Interest in applying neural models to sensory and sensory-motor systems has made attaining real-time performance a critical factor. Real sensory systems are highly parallel, with multiple parallel channels of information, so even though each channel might be implementable in real-time in software, implementing multiple channels implies hardware implementation.

The study of how particular models of neural systems perform is one aspect of computational neuroscience. Such studies are usually carried out in software, since this allows easy alteration of and experimentation with systems. However, models of the highly parallel architecture of neural systems run slowly on standard computers. This has led to interest in the use of parallel computer systems for such models [1, 2] and to interest in silicon implementations. Some researchers in computational neuroscience would like to apply their models directly to real data (implying real-time operation). Even if parallel computers can provide the speed required, it is easier and cheaper to interface silicon implementations to external hardware.

Recently, another motivation for silicon implementation has arisen as well. The continuing applicability of Moore's Law (which states that the number of transistors on a chip doubles every 18 to 24 months) suggests that we shall soon have chips with more than $10^8$ transistors but that we may also have chips whose transistors may be relatively noisy. Such large numbers of transistors seem to entail highly parallel algorithms if these transistors are not to sit unused almost all of the time [3]. Further, biological systems seem to produce relatively robust solutions with relatively noisy components, something that standard computer systems cannot achieve. This has led to increased interest in the study and implementation of neural models directly in silicon.

## 1.1 What this review covers—and what it omits

This review covers the implementation of a number of different types of model neuron, ranging from the very simple McCulloch–Pitts neuron to highly complex multicompartment models. It includes implementations of integrate-and-fire neurons and other models of intermediate complexity. It does not cover those silicon chips that are primarily concerned with using these neural models to solve a particular problem. We do describe some of the implementation techniques used for the back-propagated delta rule and the Boltzmann machine, but we do not review all these chips, concentrating instead on specific issues such as synapse implementation or noise. A more detailed review of such chips may be found in [4].

## 1.2 Organization of this review

We start by outlining the organization and structure of a real neuron. This overview will allow us to see the different aspects of neuron behavior that are

being modelled. We review the different types of models for neural systems that have been proposed, differentiating between those that deal with simple vector input (in which time is either irrelevant or occurs only in terms of the order in which the input vectors are presented), and those in which the precise timing of the inputs matters. We then discuss the different technologies for implementation and describe how different types of model neurons have been implemented. We discuss some applications, and consider what has been and can be expected to be achieved by using these different implementation technologies.

## 2 AN OUTLINE OF A REAL NEURON

Real neurons, like all real cells, are very complex. The aim of this subsection is to describe a neuron at a level of detail and in a language that is informative to a wide range of scientists and that can also be used to illustrate what is actually being modeled in particular implementations. A detailed neurophysiological description of real neuron operation may be found in [5], part II, and in [6].

There are many different types of neurons, and these very enormously in morphology (shape) and extent, as well as in the details of their biophysics. Neurons are found in a very wide range of animal species: invertebrate, insect, and vertebrate. What they all share is operation using electric charge. The operation of the neuron relies on the neuron's excitable membrane. The membrane of any cell is its outermost layer: its boundary. In neurons, this membrane is a bilipid membrane that contains ionic channels (see Figure 13.2). What makes the membrane excitable is the way in which its characteristics alter depending on the (localized) voltage across the membrane. The purely bilipid part of the membrane is essentially a very thin insulator, separating the relatively conducting electrolytes inside and outside the cell. The ionic channels (and there are many different types of ionic channel) embedded in this membrane allow selected (charged) ions to cross the membrane. Unbalanced movement of ions into and out of the neuron alters the potential difference between the inside and the outside of the neuron (see Figure 13.3). The ions of particular significance here are potassium ($K^+$), sodium ($Na^+$), and calcium ($Ca^{++}$). There is some disagreement as to whether ion channels are static or can move around inside the membrane [7].

In the absence of any input to the neuron, the excitable membrane will maintain the inside of the neuron at a particular potential relative to the outside of the neuron. This resting membrane potential is usually on the order of –65 mV (millivolts) (though this does vary across different populations of neurons). This resting potential results from the movement of ions, primarily due to the different ionic concentrations inside and outside of the neurons, and this is maintained by the $Na^+$–$K^+$ pump which keeps the $Na^+$ concentration inside the cell low and the $K^+$ concentration inside the cell high (see Figure 13.3). External inputs to the neuron result in the increase of this potential (known as *depolarization* in the neurophysiology community) or decrease of this potential (*hyperpolarization*).

Before discussing the details of how this potential changes, we consider the overall structure of a neuron: see Figure 13.1. The neuron has a cell body (the soma), and in most neurons, this has projections. These projections are of two types: the dendrites and the axon. The dendrites have a treelike structure (hence

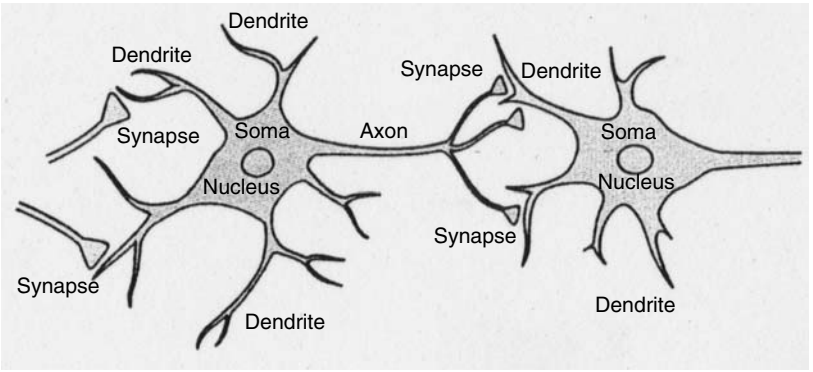


**Figure 13.1.** Overall structure of a neuron (actually, a local interneuron). Figure modified from [5] (Figure 2.8), with permission.

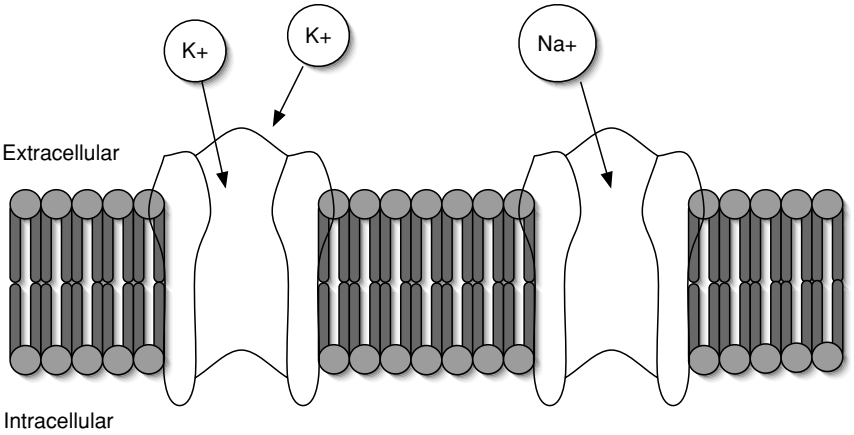


**Figure 13.2.** Patch of cell neuron membrane. Ion channels are embedded in the bilipid membrane. The membrane is made up of molecules each with a hydrophilic end (circle) and a hydrophobic end (lines), and is impermeable to ions. There are many types of ion channels, each consisting of a protein embedded in the membrane: different proteins have different permeabilities to ions because of the conformation of the protein. Additionally, the protein confirmation itself may be dependent on the voltage across the membrane, so the ion channel's behavior may be dependent on the voltage across the membrane as well.

their name, which comes from the Greek $\delta\epsilon\nu\delta\rho o\nu$ [dendron], a tree) and are located where inputs to the neuron arrive. The axon, which also has a branching structure, transfers the output of the neuron to other neurons. These two projections can be difficult to tell apart in electron micrographs, but they have different populations of ion channels in their membranes, and they function in different ways.

Connections between neurons take place at synapses. Mostly, each synapse is between the axon of one neuron (the presynaptic neuron) and the dendrite of

[1]There are also axo-axonic and dendro-dendritic synapses, as well as axonic synapses that contact the cell body.

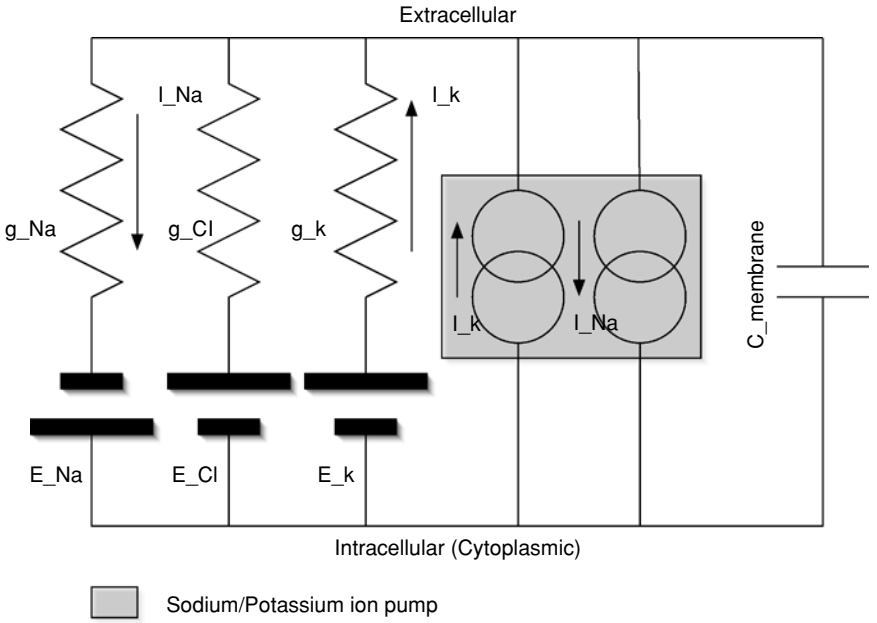


**Figure 13.3.** Equivalent circuit of a patch of membrane. The arrows show the direction of the ion movement (which is the same direction as current transfer). The sodium–potassium pump maintains the inside of the cell at a negative potential (more $Na^+$ ions are transferred out than $K^+$ ions are transferred in). The capacitance is provided by the (insulating) bilipid membrane.

another neuron (the postsynaptic neuron).[1] It is through the synapse that the potential at that point in the presynaptic axon alters the potential at that point in the postsynaptic neuron's dendrite. Brains contain a large number of highly interconnected neurons, and each interconnection consists of a synapse. Some neurons (e.g., cortical pyramidal neurons) may have as many as 10,000 synapses on their dendrites. There are therefore a very large number of synapses in animal brains. According to Koch [6], in primates there are about 100,000 cells, and about $6 \times 10^8$ synapses per cubic mm in the cortex.

In an animal brain, synapses are of many different types. Actual synaptic operation is complex. Many synapses operate by releasing small bubbles (called *vesicles*) of a chemical (called a *neurotransmitter*) from the presynaptic axon into the space (called the *cleft*) between the presynaptic axon and the postsynaptic dendrite (see Figure 13.4). In one type of synapse (ionotropic), this process directly affects the ionic channels on the dendrite, causing some of them to open and to allow influx or efflux of ions, altering the potential at that point in the postsynaptic dendrite. In another type of synapse (metabotropic), the effect is less direct, altering the ion transport of neighboring proteins. Clearly, both types of synapse require some time for the effect of the presynaptic pulse to be felt postsynaptically, and this effect (called *postsynaptic potentiation* or *PSP*) takes some time to decay as well. There are many types of both ionotropic and metabotropic synapses (often classified by the neurotransmitters used). Ionotropic synapses are faster in operation than metabotropic synapses.

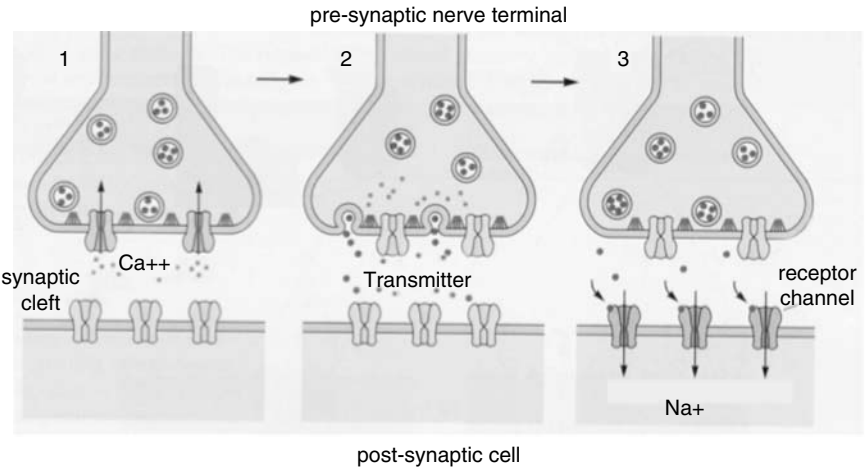


**Figure 13.4.** Diagram showing the operation of an ionotropic synapse. Modified with permission from [5] (Figure 10.7).

When the potential alteration is depolarizing, the synapse is said to be *excitatory*, and when the potential alteration is hyperpolarizing, the synapse is said to be *inhibitory*. These small alterations in potential are summed on the dendrites. On many neurons, this summation appears to be essentially linear within a certain range of potentials: outside of this range, ion channels alter their configuration, and the dendrite ceases to be linear. This nonlinearity may occur at some small portion of the dendrite, due, for example, to many nearby synapses being simultaneously stimulated. On some neurons, synapses are located on spines on the dendrite (spiny neurons, as opposed to smooth neurons), leading, it is believed, to greater ionic and electrical isolation of each synapse. Some researchers believe that the dendrites perform a considerable amount of processing (the neurophysiology is discussed in Section 19.3.2 of [6], and modeling in [8]), and that there are essentially nonlinear processes operating on the neuron that provide neurons with considerable information processing power.

In many neurons, it is the potential at a particular part of the neuron, the *axon hillock* (located on the soma of the neuron, at the root of the axon projection) that is of particular importance. At this trigger zone on the neuron, there is a large concentration of particular types of sodium channels. The result is that when the voltage at this location increases beyond a certain threshold value (usually about –48 mv), a particular set of voltage-sensitive ion channels opens and allows the influx of $Na^+$ ions, rapidly increasing the depolarization. This results in even more of these channels opening, causing a very fast and large rise in the membrane potential. As a result of this increased depolarization, two things occur: firstly, the sodium ion channels close, and secondly, another set of ionic channels opens, allowing the efflux of a different set of ions ($K^+$), causing the potential to drop nearly as rapidly as it rose (see Figure 13.5). This potential increase and decrease is regenerated along the axon, resulting in a spikelike signal passing along the axon, arriving at the synapses that this axon makes. Because the spike is regenerated, its shape is characteristic of the mechanism of its production and does not carry information. It is worth noting that (1) the sodium ion

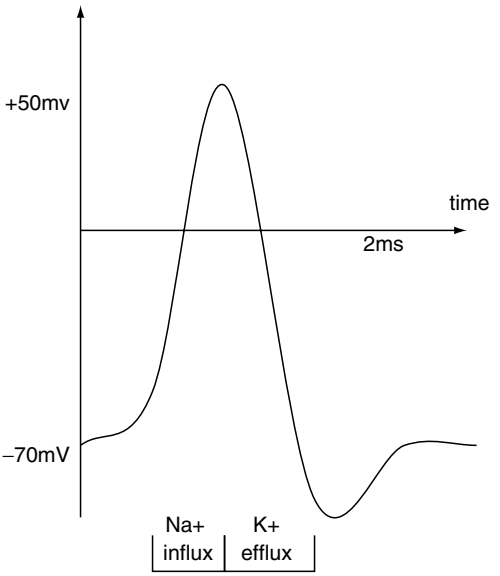


**Figure 13.5.** Graph of depolarization on an axon during an action potential (spike).

channels are not able to reopen immediately, so there is an inbuilt maximal rate at which these spikes can be produced by the neuron, and (2) the potassium efflux normally overshoots, causing a brief after-spike hyperpolarization. The delay in the reusability of the sodium channels results in the neuron's absolute and relative refractory period: that is, the period during which the neuron cannot fire again, and the period during which it is more difficult for the neuron to fire again.

The actual propagation speed of the spike is relatively slow due to both the nature of the conductance and the distributed resistance and capacitance of the axon. It can be speeded up by a process known as myelinization. In myelinization, glial cells form a myelin insulation a round the axon, reducing its capacitance, and allowing the spike to jump (by electrical conductivity, rather than by regeneration) from point to point (actually, to breaks in the myelin, known as Nodes of Ranvier) along the axon. Actual propagation speeds vary from 1 mm/sec to 100 mm/s inside brains (and faster along peripheral nerves).

Not all neurons actually produce spikes: some output graded potentials. Indeed, not all neurons have actual dendrites: some receive synapses only on the soma itself. In many synapses, the alteration in potential produced depends also on the potential at the synapse. In particular, some synapses (shunting synapses) tend to drive the potential back towards the resting potential (and thus are either excitatory or inhibitory, depending on the local potential). In addition, synapses do not always have exactly the same effect postsynaptically as a result of a presynaptic spike. Many synapses are depressing synapses: the effect of the first few spikes (after a period of presynaptic inactivity) is much larger than that caused by later spikes. Other synapses are facilitating: after a period of presynaptic inactivity, the effect of a train of spikes gradually increases. These effects appear to be due partly to depletion of presynaptic neurotransmitter, and partly to changes at the membrane on the postsynaptic dendrite (see [9], chapter 10).

One important aspect of real neural systems is that they alter in response to their inputs. They adapt, so identical inputs at different times can have different effects. This adaptation takes place over many time scales: it may occur rapidly, as a result of a single event, or very slowly over the lifetime of the animal. Early in the animal's life, the neural system grows. There is a great deal of evidence that the stimulation it receives is critical in adjusting the processing that takes place to the actual input arriving (e.g., in vision: see chapter 56 of [5]). In mammals many synapses are formed but do not last. Changes inside the system take many forms: in addition to growth and decay of synapses, there are structural and biochemical alterations at synapses, alterations in neuron morphology, and subtler changes due to hormones and diffusable neurotransmitters such as nitrous oxide (NO) and peptides. Neural models have tended to focus almost exclusively on changes at synapses. In addition to the short-term synaptic alteration above (called *dynamic synapse behavior*), synapses can also become stronger over a longer period (*long-term potentiation, LTP*), or become weaker over a longer period (*long-term depression, LTD*). Somehow, out of all these forms of adaptation, the system appears to learn: we see systemwide changes that provide appropriate changes in behavior.

There are many views on how much of the detail of the behavior of neurons is important for understanding their information-processing capabilities. These views range from the view that only the firing of the neuron matters to views that voltage-based processing on the dendrite is crucial in information processing, to views that it is the detail of the quantum effects upon the movement of ions and the conformation of proteins that matter. Some believe that the firing of neurons is essentially for information transfer, and that what happens on the dendrites is critical to information processing (see [6] chapter 20, and [10]). These differences in beliefs are at the root of the many models that we will now describe.

## 3 SIMPLE (TIME-FREE) NEURON MODELS

The simplest neural models do not include time: that is, each neuron's input is considered as a vector, and the output is computed from this input without regard for what the neuron's previous input (or output) had been. There is no internal state inside the neuron that would allow previous inputs to affect current operation. Networks of such neurons can be made sensitive to previous inputs if the network contains loops (because the state information is contained in these new inputs), but even then, these networks are sensitive only to the order of the inputs and not to their actual timing. This type of neuron model is the basis for most of the current work in neural networks for pattern recognition. Such models have been implemented on analogue computers, digital computers, and in various types of hardware.

### 3.1 The McCulloch–Pitts model

The earliest model was the McCulloch-Pitts neuron [11]. This model forms the weighted sum of its (vector) input and produces a binary output, which is 1 if the weighted sum exceeds some threshold, and 0 otherwise. This can be written

$$A = \sum_{i=1}^{n} w_i X_i \tag{1}$$

followed by $Y = 1$ if $A > \theta$, and $Y = 0$ otherwise. Here $w_i$ is the weight characterizing the synapse from input $i$, $X_i$ is the $i$th input, $A$ is the activity of the neuron, $\theta$ is the threshold, and $Y$ is the neuron's output.

The model has been formed by (1) considering each spiking neuron as a two-state device, in which the neuron is either firing (output = 1) or not (output = 0), and (2) considering each synapse as characterized by a single number ($w_i$). An excitatory synapse has $w_i > 0$, and an inhibitory synapse has $w_i < 0$. The effect of the presynaptic neuron on the postsynaptic neuron is found by simple multiplication. The overall effect of all the presynaptic neurons—the activity, $A$—is a simple linear sum: the dendrite is reduced to a single point. The nonlinearity is introduced only at the end, where the activity is thresholded to produce the output.

What makes this very simple model interesting is that it can be used to do computation. It is straightforward to design simple NOT, AND, and OR gates, and these can be assembled to provide any logical predicate. The addition of a clock allows one to build a digital computer from such devices.

## 3.2 Learning systems

Many extensions to this simple model have been proposed. In terms of basic operation, these extensions have often been relatively minor, such as graduating the output. The knowledge that real neural systems are not preprogrammed (at least in vertebrates) but adapt or learn has been very influential, partly because useful adaptation has proven very difficult to achieve in traditional computer systems, and partly because there are many problems for which a purely algorithmic solution is virtually impossible to find, whereas examples of correct behavior are quite simple to produce. A system based on learning might be able to solve such problems.

The earliest form of neural learning was suggested by Hebb [12]. In this form of learning, synapses that connect neurons that fire together are strengthened. This type of learning can be applied to make simple learning systems. These have been investigated in the context of both time-free models and models that include time: in the time-free case, they can provide a basis for certain self-organizing systems [13]. We will discuss the case including time in more detail in Section 5.4.3. We first discuss learning systems that have a teacher: that is, learning systems in which there is a known correct output for many of the possible inputs. We return to systems without a teacher in Section 3.3.

### 3.2.1 Perceptrons

One of the earliest learning systems was the perceptron [14], in which some of the geometry of the dendrite was reintroduced. What the perceptron is best known for is the perceptron learning rule [14]. This rule (described in many Neural Networks textbooks, (e.g., [15,16])) was the first one discussed that allowed the neural model to adapt itself so as to produce the desired input:output mapping. It was limited to a single layer of simple perceptrons (i.e., perceptrons

which had the dendrite geometry removed) with binary outputs (which are the same as McCulloch–Pitts neurons), but was shown to be able to generate any logical predicate that this architecture could permit. This was the first truly adaptive system, and it was hugely influential. It led to various forms of implementation (see Section 5).

### 3.2.2 The Delta rule

The Delta rule is another learning algorithm for the same architecture [17, 18]. This rule minimizes the Euclidean (least squares) distance between the actual output and the desired output by adjusting the weights (and is sometimes known as the *least mean squares rule*). It is applicable to units whose output is a continuously increasing function of the weighted sum of the inputs. The unit output function may be linear (i.e., the output is simply a constant times the activity $A$ from Equation 1), or may be a squashing function such as a logistic:

$$Y = 1/(1 + \exp(-k_1 A + k_2)) \tag{2}$$

where $Y$ is the output, and $k_1$ and $k_2$ are constants that determine the magnitude and location of the maximum slope. The logistic function has a value that is always between 0 and 1. Other squashing functions (e.g., tanh) have also been utilized. Again, it has been shown that the Delta rule can produce any output that the particular single-layer architecture could produce, and given small enough weight changes, will converge to a solution (see, e.g., Section 5.4 of [15]). The way in which the network is used is that a set of (input, output) pairs is produced, and these are then applied to the network as the input and the desired output for this input. The weights are then adjusted to reduce the error: that is, the square of the sum of the differences between the desired and actual outputs.

However, the limited computational ability of the single-layer architecture was proven in [19]. The architecture can only produce linearly separable mappings. Minsky and Papert's doubt as to whether it could be extended either to more complex perceptron networks or to a larger class of functions led to a decrease in the effort extended in neural computing (see [15], Section 1.2) in the 1970s and early 1980s.

### 3.2.3 The Hopfield network and the Boltzmann machine

Two new adaptation algorithms were introduced for similar types of neurons in the early 1980s, one for binary neurons (the Hopfield model, and its extension, the Boltzmann machine), and the other an extension of the Delta rule (the back-propagated Delta rule). Both of these networks were hugely influential, and both were implemented in various forms in hardware.

Hopfield's network [20] is symmetrical: that is, $w_{ij} = w_{ji}$, where $w_{ij}$ is the weight from presynaptic neuron $j$ to postsynaptic neuron $i$. This network is not a simple layer of neurons, but has cycles. Updating the network was done neuron by neuron, asynchronously, and the Hopfield proved that the network eventually settles into a stable state. It was therefore the first network to have a dynamical behavior, although this was not normally used in its operation. The network is considered to have an overall energy

$$E = -\frac{1}{2}\sum_{i,j=1}^{N} w_{ij} X_i X_j \tag{3}$$

where the neuron's output, $X_i$, is either +1 or 1, rather than +1 or 0, and updating each neuron's state minimizes this total energy, $E$. The network could be trained to be an associative memory by applying the vectors to be stored and then adjusting the weights so as to minimize $E$. Hopfield and others (as is clearly explained in [15]) showed that such a network could remember a maximum of $0.138N$ vectors. These could be recalled by providing the network with an incomplete vector, thus providing content-addressable memory.

An important extension to this network was the Boltzmann machine [21]. In this network, the original Hopfield network is extended by adding new nodes that are not connected to the outside world. These so-called hidden nodes can learn to form internal representations that can allow the network to learn additional vectors and can be used to allow the network to classify its inputs by examining the hidden unit state. However, the learning technique also has to change (since the Hopfield learning recipe cannot train weights to and from hidden nodes). The learning algorithm used is statistical in nature: essentially, it uses concepts from statistical physics and Boltzmann distributions (hence the algorithm's name) to set these weights. A comprehensible description may be found in [15], chapter 7 or in [16], chapter 11. Using such techniques in software is exceedingly slow. However, the idea that this type of network could learn some form of internal representation helped rekindle interest in the whole area, and the slowness of the algorithm in software helped motivate implementations of this type of network in silicon.

### 3.2.4 The back-propagated Delta rule

The best known of the simple neural network learning rules is the back-propagated Delta rule. Discovered independently at least five times [22–26], it permits a Delta rule like network to be extended from a simple single layer to a feed-forward network (see Figure 13.6). The basic idea is that errors at the output layer are funneled back to the units of each hidden layer: for details see any book on neural networks, e.g., [15] chapter 6, or [16], chapter 4. Once the error at a unit is known, it can be used to adjust the weights to that unit, essentially using the original Delta rule.

There are two problems with the back-propagated Delta rule: firstly, it is no longer the case that continued application of the learning rule will necessarily allow the network to learn the input:output mapping, even although it may be possible for the architecture to do so; and secondly, learning tends to be slow. The result of the first problem is that one cannot be sure that the network produced is the best network possible given the (input:output) pairs that have been provided. So-called local error minima can result in the network stopping learning before it has done as well as it can. Further, if the (input:output) pairs contain some noise (perhaps the result of measurement errors), it is quite possible for the network to attempt to learn this noise. A great deal has been written about the best ways in which to use this type of network. Certainly, like the Boltzmann machine, it is capable of extracting information about the (input:output) pairs provided and coding this into its weights. Learning is slow because the mapping

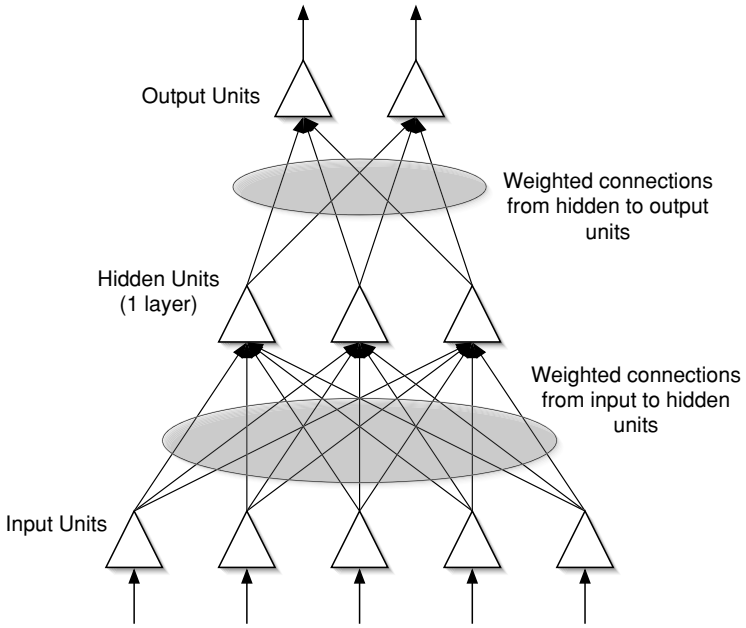


**Figure 13.6.** Feed-forward neural network. The input layer simply transfers its inputs through (adjustable) weighted synapses to the hidden layer. There may be a number of hidden layers, with different numbers of units. The radial basis function network [16] has a similar structure, with one hidden layer, whose units have a peak response at one point in the input space.

between the weights and the error (the so-called error surface) can be very complex: gradient descent methods applied to high-dimensional complex surfaces must move slowly because they otherwise risk missing the desired minima of the surface. The error surface may also contain local minima: if the weights are trapped in one of these, the performance will be suboptimal.

Because of the wide possible applicability of this network, and because it is slow to train, many attempts have been made to implement it directly into hardware. These are reviewed in Section 5.3.

Many extensions to this rule have been described and many have been concerned with improving the form of the gradient descent, attempting to make it closer to steepest descent (some are described in [16], chapter 4). Others have attempted to replicate the hidden layer's effects by recoding the input. The idea here is that what the back-propagated Delta rule does in its hidden layers is to recode the input so that the mapping from the recoded input to the output becomes separable, thus permitting the Delta rule to be used. This is essentially the basis for the Radial Basis Function network [27], (see also [16] chapter 5), which performs recoding and uses the simple Delta rule between the recoded input and the desired output.

Bishop [28] has shown that these types of network essentially implement a form of statistical algorithm. This does not reduce the utility of these systems, and indeed helps to explain why they are so useful. However, it does show that the

limitations of this type of algorithm are essentially the same as the limitations of the statistical techniques.

### 3.2.5 Learning sequences

All the above rules can be turned into systems that learn sequences of inputs, either classifying the sequence or attempting sequence completion. In such sequences it is the order of the elements, not their precise times, that matters. Learning can be achieved in a number of ways: a window through the sequence can be used as the input to the network (i.e., the last $n$ elements of the input are used as input, and the output target might be the next element in the sequence), or a network with loops may be used, in which case information about the previous sequence element is held internally inside the network (see, e.g., [29]). What networks of these types cannot achieve is learning anything that requires information about the precise timing (as opposed to order) of the input vectors.

## 3.3 Self-organizing systems

Self-organizing systems are systems that adjust their behavior in response to their input. No correct output is provided: instead, the system adjusts its internal parameters so as to detect some regularity in the input. Such situations commonly occur in sensory perception: the input is of very high dimensionality (for example, there is one value per light sensor in a camera system, or one value per bandpass filter in a sound sensor), yet although this suggests a very high number of possible inputs, real inputs are confined to some relatively small subspace. In other words, the probability distribution functions of each of the (scalar) inputs are not independent. It is usually the case that the aim of self-organizing systems is to adjust the weights in the system so as to produce outputs (usually of lower dimensionality than the input) that catch the important aspects of the variation in the input.

The idea of neural processing as data reduction goes back to [30]. Simple Hebbian learning systems have some utility in this area: consider a number of inputs that converge on a single output. Assume that the synapses are excitatory, and that a number of coincident inputs are required to make the output neuron fire. Inputs which co-occur in large enough numbers to make the output unit fire will tend to increase their weights, making the output neuron more sensitive to these inputs. However, simple Hebbian learning alone fails to work effectively because the weights increase without limit. Below, we discuss two algorithms that add something to Hebbian learning and that have been candidates for silicon implementation. A useful introduction to this field may be found in [16], chapter 8.

### 3.3.1 Learning vector quantisation

Learning vector quantization (LVQ) is used to map a number of inputs (each with a scalar value) into one of a number of outputs. LVQ is one of a class of algorithms known as competitive learning algorithms (see [15], chapter 9). This class of algorithms clearly produces outputs of lower dimensionality: the mapping is from some subset of $R^N$ to $\{1..M\}$ where $N$ is the number of inputs and

$M$ the number of output units. Normally, all the input units have synapses to all the output units, initially with random weights. The learning algorithm has a Hebbian aspect, in that weights between input units and output units that fire are increased. However, usually only one output unit is allowed to be active at a time, and the weights to that unit are adjusted in such a way that the total weight (or the total squared weight) remains the same. Some variants also reduce the weights on some of the synapses on inactive output units.

LVQ algorithms are of particular interest in compressive coding: by replacing the input vector with the code for the output unit that best represents it, a very considerable reduction in data volume can be achieved. Further, the LVQ network adjusts itself to the statistics of the data. Because such coding is frequently required in real time (for example, for transmitting coded images), there is considerable interest in the hardware implementations of LVQ systems.

### 3.3.2 The Kohonen mapping network

The Kohonen mapping network is a variant of LVQ in which not only the weights of the winning unit are adjusted but also weights to nearby units are adjusted (see [16], chapter 9, or [31]). This description presupposes a definition of "nearby", forcing the designer to place some form of topology on the output units. For example, the output units might be organized in one dimension (as points on a line or a circle) or in two dimensions (as points on a grid or on the surface of a sphere or cylinder or torus). The network is trained by being exposed to many input vectors, and the weights to the output units are adjusted. Usually, the number of units whose weights are adjusted for each winning pattern is gradually reduced.

After training, novel inputs will normally result in some localized area of the output units being activated. In this way, high-dimensional data are mapped into some area on a surface. Such data compression can be very useful for sensory information, for example, in robotics or surveillance. Often the requirement is that training can be relatively slow, but operational results are required quickly for real-time applications. This situation has led to interest in silicon implementations of this technique.

## 4 MODELS THAT INCLUDE TIME

Model neurons that include time are those in which the actual timing of the input (as opposed to the order of the input) matters. Models of this form can be sensitive to the actual timing of their inputs, as opposed to their order: the neurons contain internal time-varying state. The simplest form of neural model that includes time is the integrate-and-fire model. Such neurons can process general time-varying signals, but their outputs are normally spike trains. In common with spike trains of real neurons, the actual shape of the spike is irrelevant. All that matters is the timing of the spike. Thus, the output can be characterized by

$$S = \{t_i : i = 1 \ldots n\},\ t_i < t_{i+1}$$

where $t_i$ is the time of the $i$th spike train in a train of $n$ spikes. More complex models model the neuron in more detail, sometimes including the membrane itself and sometimes including the actual production of the spike.

## 4.1 The leaky integrate-and-fire model

The leaky integrate-and-fire neuron has a very long history: the concept can be traced back to 1907 [32]. In this neuron model, the dendrites are modeled as single points at which the synaptic inputs are summed, while current leaks away linearly: a detailed description can be found in [6], chapter 14. Below threshold, the voltagelike state variable at that point, $A$, is described by the equation

$$\frac{dA}{dt} = -\frac{A}{\tau} + I(t) \tag{4}$$

where $\tau$ is the time constant of the point neuron (i.e., a [reciprocal] measure of its leakiness), and $I(t)$ is the total external input to the neuron (see Figure 13.7). In the presence of positive input, the activity $A$ can rise to the threshold $\theta$. When this threshold is crossed from below, the neuron emits a spike, and $A$ is reset to some initial value. The mechanism of spike generation is generally ignored in the model, and the output is characterized entirely by the sequence of spike times. This type of neuron is sometimes known as a *point neuron*, because all the geometry of the dendrite has been shrunk to a single point. If $R$ is infinite, then the neuron is not leaky, and it simply integrates its input until it reaches the threshold. If $\tau$ is small, then more recent inputs have a larger effect on $A$. If $I(t)$ is made up of a number of excitatory synaptic inputs, each of which is not large enough to cause $A$ to exceed $\theta$, then the neuron will act as a coincidence detector, firing when a number of its excitatory inputs occur at about the same time, allowing $A$ to reach $\theta$ in spite of the leakage.

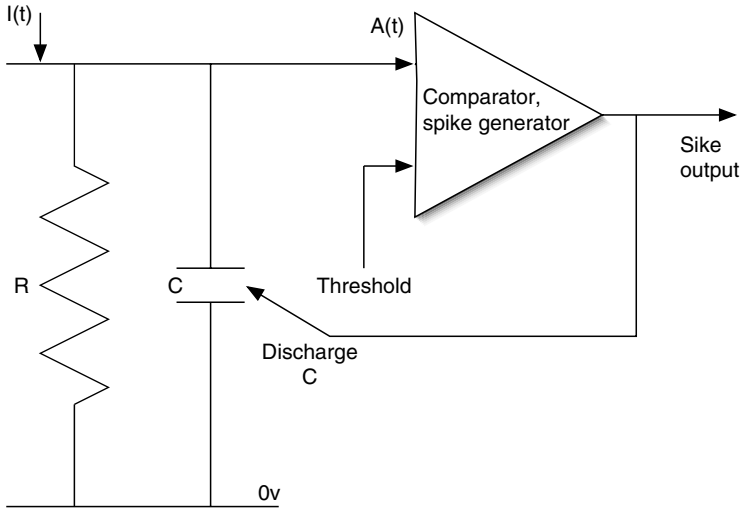


**Figure 13.7.** Leaky integrate-and-fire unit. The resistor $R$ models the overall (fixed) leakage of the membrane (if omitted, there is no leakage), and the capacitor $C$ models the overall capacitance of the membrane. The time constant $\tau = RC$. When a spike occurs, the capacitor is discharged.

### 4.1.1 Other point neuron models

The leaky integrate-and-fire model is the best known (and most frequently implemented) of the models that represent the dendrite as a single point. Another important model is Gerstner's spike response model [33,34], in which the threshold is dynamic and the shape of the postsynaptic potential is modeled. The dynamic threshold permits the neuron's refractory period and relative refractory period to be included in this relatively simple model.

In Equation 4, the leakage is linear. Feng and Brown [35] suggest a nonlinear leakage coefficient (equivalent to making $\tau = \tau(A)$), with the result that under certain conditions, inhibitory input can increase the firing rate [36]. Izhikevich [37] reviews a number of point neuron type models, both from the point of view of neural plausibility and computational efficiency. Not surprisingly, the more biologically plausible models take much more computing time. Izhikevich has proposed a new model [38] based on bifurcation analysis, which can generate realistic neural output from a simple simulation.

## 4.2 More detailed neuron models

Compartmental neuron models divide up the neuron into a number of sections (or compartments), each of which is modeled individually, with electrical current feeding into neighboring sections. The advantage is that the whole neuron (soma, dendrites, and axon) may be modeled with a degree of accuracy that can be determined by the modeller. The morphology may be simulated (at least as far as branching and neurite diameter is concerned), and each section may be given different properties. The usual techniques are based on the Hodgkin–Huxley equations (see [6], chapter 6, and [5] chapter 7), and these allow different populations of ion channels in each compartment to be modeled. Essentially, a nonlinear leakage current is associated with each ion type. There are some standard simulation tools developed for this type of simulation, most notably Neuron [39].

There are also simplified models, often based on the Hodgkin–Huxley equations—for example, the FitzHugh–Nagumo equations and the Morris–Lecar equations (both reviewed in [6], chapter 7, and discussed in terms of computational efficiency in [37]). Indeed, Feng and Brown's model [35] is a version of the FitzHugh–Nagumo model. These models can aid the speed of computation (in software) and possibly the complexity of a hardware implementation.

## 4.3 Learning in models that include time

Neural models that include time generally have a spike-based output. This spike output may be thought of either as coding a value in terms of its instantaneous spiking rate (rate coding) or by the precise timing of the spikes. In rate coding (and also in the case where the output is not a sequence of spikes, but a continuously varying value), it is possible to apply, for example, Hebbian-type learning rules as discussed in section 3.2. There are no equivalents of the Delta rule or the perceptron learning rule that make specific use of these types of code. These learning rules are based on the idea of a single vector input producing a single vector output. One can still use this formulation of a network that uses

rate-coded spiking neurons, but no advantage is being taken of the neurons including time.

Although rate-coded and graded-output neurons clearly can have more sophisticated learning rules, there has been more interest in learning rules for spike timing-coded neurons. This case is of particular interest to computational neuroscientists, since it may inform brain science. There has been particular interest in temporal versions of Hebbian learning rules (reviewed in [40] and also in chapter 10 of [34]). To apply Hebbian learning to spiking neurons, we need to reconsider what "firing together" means: the usual view is that, for excitatory synapses, those whose postsynaptic currents assist in making the postsynaptic neuron fire are strengthened, while those whose postsynaptic currents do not help are weakened. Generally, these new rules do not alter synaptic strength unless the postsynaptic neuron fires: thus their effect is to strengthen synapses that were active just before the postsynaptic neuron fired and to weaken those that were active just after the postsynaptic neuron fired. Although such learning rules have not yet been demonstrated to be effective in applied neural networks, there is considerable interest in silicon implementations of this type of rule (see Section 5.4.3).

## 5 TECHNIQUES FOR HARDWARE IMPLEMENTATION

Hardware implementation of neural models and networks of neural models can allow these systems to operate in real time and to use the massive parallelism inherent in these types of design. Sequential computers cannot provide true parallelism, and parallel computers are expensive: further, there is often a mismatch between the very intense intercommunication required for neural computers and the relatively low bandwidth parallelism provided by the cheaper forms of parallel computer, such as networks of transputers or Beowulf clusters [41].

In fact, direct hardware implementation of neural systems and networks has a relatively long history. Prior to the advent of the workstation, neural modelers were forced either to use mainframe computers or to develop their own hardware. Models of excitable membranes using discrete components were developed [42–45], as well as full neurons [46]. More computationally oriented models of perceptron-based machines were built [47,48]. This chapter is not the place for a full review of this historically interesting material: however, the history does show that dedicated hardware for neural systems is not a new idea. Modern neural hardware developers are primarily interested in chip-based implementation. This focus has certainly made the resulting hardware smaller (the neural model of the avian retina developed by Runge et al. [49] ran to 50 circuit boards!), though more difficult to test and modify.

There are many possible ways of organizing a review of implementations of neural models. In a much earlier article [50], these were organized by chip type. In [4], they are organized by actual chip, and in [51], a table of chips and their characteristics is provided. Here, we review some of the issues, then discuss the analogue versus digital issue, and then look at the question of whether the implementations use static (time-free) or dynamic (including time) approaches.

## 5.1 VLSI implementation of neural models

Chip-based implementations are very attractive to the neural system implementor. Not only are they small (and easily incorporated into complete systems) but also most design systems proffer at least some facilities for testing the design prior to actual chip manufacture. Further, if the implementation is successful, the designer will normally receive a number of chips, allowing more than one researcher to work with the implementation. In addition, reusing designs or sections of designs is relatively straightforward. However, implementors of neural systems in silicon do not have the luxury of developing a new silicon technology, and so must use technologies that were developed for other purposes, such as for high-speed digital processors.

The basic implementation techniques are summarized in Table 13.1. There are many different possibilities within each of these classes of implementation technique. Analogue implementations are normally custom integrated circuits. These may use the linear range of the field-effect transistors (above threshold) or the very-low-power exponential part of their range (subthreshold). Digital implementation techniques range from software (i.e., implementation on a normal computer) to field-programmable gate array (FPGA: a technology in which an array of electrically programmable gates can be interconnected in an electrically programmable way) to application-specific digital integrated circuits (digital ASIC). Of course, these technologies may be mixed, even on the same chip. We note in passing that field-programmable analogue arrays (FPAAs) are in development (see, for example, [52]), although they are not yet nearly large enough to replace complex analogue ASICs. The downside of hardware implementation is the length of the timescale from design (or modification) to implementation. For all the hardware implementation techniques (except FPGA), change of design means refabrication, and this process generally takes months. On occasion, focused ion beam (FIB) machines (see, e.g., http://www.feico.com/support/fiblab.htm) can be used to modify devices, but this option is often not available, or else is inappropriate for the modification required. FPGAs can be reprogrammed quickly: they are a technology with aspects of both hardware and software.

In Table 13.1, the "Degree of Implementation" [53] column relates to the extent to which all the elements of all neurons exist as separate hardware components. Fully implemented systems have identifiable (and different) circuit elements for each entity being modeled. Real neural systems are fully implemented. Most analogue implementations are also fully implemented. However, full imple-

**Table 13.1.** Summary of characteristics of different implementation techniques for implementing neural systems.

| Implementation Technology | Degree of Implementation | Real-time Speed | Power System | Consumption |
|---|---|---|---|---|
| Subthreshold a VLSI | High | High | Yes | Very low |
| Above threshold a VLSI | High | Very high | Yes | Medium |
| dVLSI | Low | High | Possible | Medium to high |
| FPGA | Low-medium | Medium | Possible | Medium to high |
| Workstation software | Minimally low | Low | Not usually | High |
| DSP based software | Low | Medium-high | Possible | High |

mentation is not usually possible for digital VLSI implementations since replicating, e.g., digital multipliers at each synapse would make the circuit impossibly large: instead, the same functional unit may be reused frequently. For example, one digital multiplier may well be used as part of the implementation of many synapses. Such a virtual design (again using the terminology of [53]) trades off the speed of the functional unit against its area and the switching involved in multiplexing signals to the functional unit. By careful design, real-time performance may still be possible, but even with fast digital electronics, it is not guaranteed. FPGA- and DSP-based implementation are not normally fully implemented. Depending on the design chosen, component re-use will occur to a greater or lesser extent. Pure software implementations use the CPU(s) of the workstation for all computational tasks and have the lowest degree of implementation. Even implementations on parallel sets of workstations (e.g., Beowulf) simply tend to distribute the different parts of larger simulations across a number of workstations. DSP chips are also software driven and are normally controlled from a workstation. The degree of implementation depends on the details of the design (for example, on the number of chips used). The systems are easily reconfigurable, but because they are special purpose, they require specific software packages and can be difficult to program.

## 5.2 Analogue or digital VLSI

The first choice facing a designer intent on implementing a neural model in VLSI is whether to use an analogue or a digital design. If an ASIC is being produced, it is very likely that the technology being used for chip manufacture might have been developed for digital designs. When the implementor is attempting to build an analogue ASIC, or, indeed, any target except a digital ASIC (for example, a mixed (or hybrid) design: part digital and part analogue), problems arise. The quoted feature size for a particular technology ($\lambda$) is intended for use in the production of digital gates. For such gates, all that matters is that the realized circuit conforms with the designed circuit and that the switching voltage between an FET being on and off is within a particular range. For above-threshold analogue VLSI, the implementor is attempting to use the linear part of the transistor's characteristic, and so is reliant on the actual placing and shape of the transistor's $I_{ds}/V_{gs}$ characteristic. This reliance can lead to matching problems, though it does appear to be the case that these problems are not major. However, for subthreshold aVLSI, the designer is reliant on the characteristic of the transistors before they turn on (i.e., $I_{ds}/V_{gs}$ below threshold). This is not a characteristic that digital chip designers generally care about since it does not impact on their designs.

Why then would anyone consider analogue implementation? We discuss below some of the differences in implementation characteristics implied by these two different approaches.

### 5.2.1 Signal coding

The primary difference between digital and analogue systems is in how signals are coded. Digital signals are discrete values, valid at specific instants, and analogue

signals are continuous values in continuous time. In a digital system, the two primary parameters of a signal are sampling rate and sample length. In an analogue system, the parameters are bandwidth, slew rate (maximal rate at which a signal can change), noise level, and drift. (Drift causes the analogue signal to change slowly [perhaps due to temperature variation], again reducing overall accuracy.) There is a third form of coding, namely, spike encoding, that provides spikes at specific instants, which we discuss further in Section 5.4.

In a digital system, the sampling rate determines the signal bandwidth: the maximal bandwidth is half the sampling rate. The bandwidth determines the maximal rate at which values (such as postsynaptic potentials) can change, For fully implemented systems, both analogue and digital systems normally have plenty of bandwidth in hand compared with real neural systems. However, digital systems are not normally fully implemented, so they need to have a higher bandwidth. If a particular piece of circuitry is used in $P$ different ways (for example, a digital multiplier might be used in $P$ different synapses), then its processing bandwidth (or speed) must be at least $P$ times the actual required bandwidth. The sampling rate also determines the accuracy with which the time of an event can be determined: this can be important for spiking neurons (see also Section 5.4).

In a digital system, sample length determines the accuracy with which a value can be held: theoretically, an analogue system holds a value precisely, but the effect of noise is that the value is no longer precise, and drift causes further difficulties. Maximizing sample length leads to space problems: for most circuitry, the number of gates required is at best proportional to sample length.

### 5.2.2 Memory technologies

Memory is required in neural systems to hold constant values (such as thresholds, delays, or characteristics for ion channels) as well as variable values such as those characterizing synapses or any other aspect of the model that can alter. Digital memory techniques are well known: memory consists of a string of bits, each held either as a static RAM (sRAM) or a dynamic RAM (dRAM) cell. dRAM requires frequent refreshing, and both sRAM and dRAM are volatile and thus require reinitializing on power cycling. Another possibility is to use electrically erasable programmable read-only memory (EEPROM or flash memory) techniques to provide nonvolatile but rewritable memories.

Analogue memory elements are more problematic. In discrete systems, fixed values may be held by selecting discrete components (usually resistors and/or capacitors) with particular values. This approach is not practical on analogue VLSI chips: resistors can be fabricated, but their accuracy is low, and capacitors of any reasonable size take up too much space. One method of keeping values in analogue systems is to use a digital storage solution combined with a digital-analogue convertor (DAC). Such a system can provide accurate storage, with storage for each value taking up little space. If many values are required (as might be the case for synapse weight storage), this usually means using a smaller number of DACs and sharing them, with a consequent need for additional routing of signals.

True analogue VLSI storage generally uses either the charge on a capacitor or floating gate technology [54]. The simplest technique relies on simply storing some charge on a capacitor, which is essentially isolated. However, this charge

tends to leak away, and so a refresh system is often introduced. A variant on this technique for increasing the quality of this form of representation is to use the ratio of the charge stored on two neighboring capacitors, relying on them both leaking at the same rate [55]. Such memories are essentially volatile. Restoration of these values often makes use of external digitally held values and an on-chip DAC. Floating-gate technology proffers the possibility of longer-term nonvolatile analogue storage: it is based on the same techniques that are used for EEPROM, but attempts to retain an analogue value [54, 56, 57]. Extended analogue storage is not a requirement of standard digital technology, and so is not supported in design systems. This can make chip development more difficult because the devices are often not supported in simulation environments.

The above techniques are for storing constant values. However, an important aspect of neural simulations (and particularly of neural networks) is adaptivity: we need to be able to adjust values, and to adjust them gradually. This process consists of first determining what the parameter alteration should be (discussed in Sections 5.3.1 and 5.4.3), and second, implementing some mechanism for on-chip parameter alteration. For digital storage, there is no difficulty in adjusting a binary string: what is required is either an adder or a step-up/step-down counter, or each value may be rewritten, having been recalculated elsewhere. For analogue systems, the problem requires novel solutions. This is not a new problem: specialized devices for weight storage and updating in the analogue domain have a long history (see Section 8.2 of [53]). The original Perceptron Mark 1 used motor-driven potentiometers. Later, Widrow introduced the memistor, a copper/electrolyte variable-resistance electrochemical cell. Some systems expect weight adjustments to be determined and effected from outside of the chip: weights are recalculated and then updated using a digital computer interface (e.g., [55, 58]). If the neural simulation is to be be trained without an external computer, then it should incorporate internal adaptation. For capacitative storage, there must be some mechanism for gradually increasing or gradually decreasing the charge stored on the capacitor. For floating-gate techniques, there needs to be a mechanism for charging and discharging the floating gate. Meador [56] suggested using pairs of floating-gate transistors and transferring charge between them. Diorio [57] uses hot-current injection to add electrons to its floating gate and Fowler–Nordheim tunneling to remove them. External checking of the actual weight may be required because of variations in chip processing. This is still an area of active research: Hsu et al. [59] have developed Diorio's ideas in a competitive learning chip, and Morie et al. [60] are developing a multinanodot floating-gate technique for postsynaptic pulse generation.

### 5.2.3 Simple arithmetic operations

Whether one is using a simple neuron like that in Equation 1 or a more complex neuron with an explicit dendrite, one needs to use arithmetic operations both for calculating neuron output and for any internal parameter alteration. For example, to calculate the postsynaptic activation one requires at least a multiply; to compute the total activation, one requires addition. In a digital implementation, these process imply the use of adders and multipliers, and in an analogue implementation the use of circuitry that can sum voltages (or currents) and

perform multiplication on whatever circuit value is being used to represent the output, activation, or synaptic data.

Such arithmetic operators occur very frequently in neural models. In real neurons, these operations are accomplished using (for synapses) the effects of altering release probabilities for presynaptic neurotransmitter vesicles and changing the probabilities of opening postsynaptic ion channels, and (for the activation summation) by charge summation inside the dendrite. Both these operations take up very little space indeed. In digital systems, very fast adders and multipliers can easily be built. Adders tend to be relatively small, but multipliers tend to be larger. Depending on the multiplier implementation, one has a choice between having the latency and the size of the multiplier increase linearly with operand length (or having the latency increase as the log of the operand length) and having the size of the multiplier increase as the square of operand length [61]. In either case, it is not practical to use a separate multiplier per synapse for neural network implementation, although it can be practical to use one adder per neuron for activation summation.

In analogue implementation, simple multiplication of positive values (single-quadrant multiplication) is relatively straightforward. Thus, if a neuron's output can be guaranteed to be positive, and the weight is known to be excitatory (inhibitory), the product can be added to (subtracted from) the postsynaptic activity. However, the most popular time-free neural model (back-propagation) has neurons whose weights can be either excitatory or inhibitory, and can change between these during training. In addition, some versions of back-propagation use a $\tanh(A)$ output function, rather than a logistic $(1/(1 + \exp(-A)))$ function, resulting in outputs being either negative or positive. Thus, either two-quadrant or even four-quadrant multiplication is required. This can be problematic, since it is very easy for the product to be outside the linear range of the multiplier (see [62], chapter 6).

In analogue implementations, it is possible to use the transfer characteristics of MOSFETs (or of circuits of MOSFETs) directly, even when these are nonlinear. This option was one of the driving forces behind the Mead's original proposal to use subthreshold a VLSI for neural modeling (and for neuromorphic systems) [62]. In this way, exponential functions, differentiators, and integrators can be built directly (see also [63]). This approach is clearly much more space efficient than developing digital circuits for the same function, and this is the reason why subthreshold a VLSI systems have a very high degree of implementation. However, design is more difficult (or perhaps more skilled), and one is reliant on the silicon implementation behaving in exactly the same way as the designer's model, which, as discussed earlier, may be difficult to achieve.

## 5.3 Implementing simple time-free neuron model networks

An implementation of a simple time-free neuron model consists of an implementation of the synapses, of the dendrites, and of the generation of the output of the model neuron. In addition, it is necessary to implement the interconnection between the neurons. Further, for adaptive systems, one must also implement both parts of the mechanism for adaptation. The primary difficulties arise at

synapses. The problems are computation of postsynaptic potential and computation (and implementation) of synaptic parameter alterations. If there are many neurons, there may also be problems associated with neuron interconnection.

The dendrites accumulate the activity passed to them by the synapses. This is a simple additive process (see Eq. 1). In a digital implementation, this is simple addition, with the number of bits used determining both the precision of the result and when overflow or underflow might occur. In an analogue implementation, either currents or voltages may be summed. Accuracy is then a function of noise, drift, and the linearity of the system. Analogue equivalents to overflow and underflow occur when the current or voltage reaches its limit.

The output of the neuron may be binary (for McCulloch–Pitts neurons or perceptrons, for example), or it may be graded (for a linear threshold unit, for example). In digital implementations, the former is achieved by numerical comparison with a fixed (binary-coded) threshold, and therefore requires an adder. In analogue implementations, this adder is replaced by a comparator, and the threshold is required to be stable. Where the output is graded (as is the case for Delta rule [plain and and back-propagated] and for the Radial Basis Function networks), some function must be applied to the activity. This may be simple multiplication (for linear units) or a logistic function (Eq. 2) or some other function. Accurate implementation may be quite complex in a digital implementation. Sometimes look-up tables are used to speed up this operation. Generally, the output function is shared between a number of neurons on the same chip (partial implementation). In analogue implementations, it may be virtually impossible to achieve exactly the output function required. However, in both the Delta and back-propagated Delta rule, it is not the exact function that matters but rather that the function is a squashing function, which is smooth and always has a positive derivative. Given suitable limits to the activity of the neuron, this outcome can often be achieved relatively easily and compactly in an analogue implementation. One can claim some biological plausibility for this approach as well, since the activation at the axon hillock (where spiking is initiated) will necessarily limit as it tends towards both positive and negative values due to the opening of additional ion channels. Both this form of limitation and the limitation on maximal spiking rates are likely to have similar forms of characteristics, but are unlikely to follow some analytical mathematical function.

Lastly, model neuron outputs must be connected to the appropriate synapses. Each neuron output may be connected to many different synapses, though each synapse is normally connected to only one neuron output. In a digital implementation, this outcome is best achieved by the use of some form of bus, particularly if the synapses are not fully implemented. The bus allows values to be directed to whichever element of circuitry is implementing that synapse at that time: it is straightforward to calculate whether the bus speed is sufficiently high, and to replicate it if required. In an analogue system, it is more common to use a rectangular array of synapses, as discussed in the next section.

### 5.3.1 Synapses for time-free neurons

The emulation of synapses is critical in silicon implementations of model neurons. As with real neurons, synapses are by far the most frequently occurring

element of model neural networks. Because a single model neuron may have so many synapses, the system designer is faced with a choice between replicating a small amount of circuitry and hence a simple synapse (full implementation) or sharing the circuitry between a number of synapses (partial implementation). Replicating large amounts of circuitry is generally not practical.

The basic function of a synapse in a network of time-free neurons is to allow a presynaptic input to affect the postsynaptic neuron. Simple implementations of synapses are generally multiplicative: the change in postsynaptic activity is proportional to the presynaptic input, and the constant of proportionality is known as the weight, as in Eq. (1). For simple binary neurons, this set up can be implemented by adding or substracting a constant (weight-dependent) amount from the activity. For graded output neurons, multiplication of the output of the presynaptic neuron and the weight is required. Digital multipliers are standard circuit components but contain a considerable amount of circuitry. Full implementation of such multipliers results in the synapse numbers becoming the limiting factor in what can be placed on a single chip, while partial implementation implies precise switching of the presynaptic input and the appropriate weight, and of the resulting product.

Mechanisms for weight storage were discussed in Section 5.2.2. Chips normally have the weights on-chip, although some may require the weights to be downloaded at start-up. Analogue synapses are often stored in a rectangular array, as illustrated in Figure 13.8. For example, the Intel 80170NX chip [55] has a 160 by 64 array of synapses. Each set of synapses belonging to a single neuron is in a vertical column. The presynaptic inputs from a single neuron are in a hor-

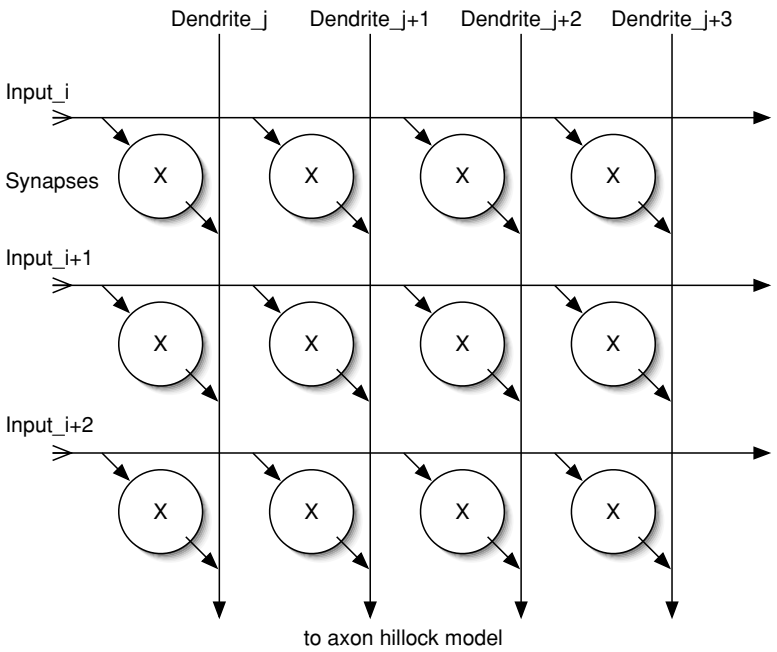


**Figure 13.8.** Synapses (each a simple multiplier) are arranged in a rectangular formation. Dendrites accumulate current from all synapses to that neuron.

izontal line. Where the two meet, there is a synapse (though some may have no effect). The vertical lines accumulate this input (whether as a current $I_{syn}$ or a voltage $V_{syn}$) and apply this to the simulated axon hillock. Weight storage precision can affect the system in terms of both the system displaying the correct behavior once trained and the system being able to work correctly during training. (This is a general problem in digital signal processing: see [64].) In general, attaining correct behavior once trained is less demanding than attaining appropriate behaviour during training: 4 to 8 bits is enough for almost any application [65,66].

For specific fixed applications, weights may be set externally and fixed. Generally, synaptic adaptivity is attained by weight alteration, which requires that the weights be updatable. We discussed mechanisms for updating the weight in Section 5.2.2: here we are interested in determining what this weight update should be. This calculation may take place on-chip or off-chip. Different neural network algorithms make different changes: with the exception of the perceptron rule and the Hopfield network, however, these changes are often small. Further when using the back-propagated Delta rule, small changes often occur a long way from the best solution due to nearly flat areas in the error/weight space. If the weight update calculation is off-chip, this situation may not present a problem since high-precision arithmetic will be available off-chip. However, if the changes are calculated on-chip, there can be difficulties with digital weight storage update calculation when the weight change becomes less than one bit. (Indeed, these problems apply equally at weight update, even if the changes are calculated at high precision.) This critical point results in a sudden performance breakdown [67] in training, although such precision is not required in recall. Digitally stored analogue weights suffer from exactly the same problem. There has been considerable software exploration of this problem [68]. In a purely analogue system, weights can often be adjusted by very small amounts (exactly how small depends on the details of the implementation), limited by the noise and drift in the system. Failure from this source tends to be less sudden. Changing purely analogue weights can be an imprecise affair, and some systems allow a "chip in the loop" form of updating (e.g., the Intel 80170 [69]), where the effect of the weight update is tested immediately and the update is possibly repeated.

### 5.3.2 Developed hardware for time-free neurons

Hardware time-free neuron implementations have been around for some time (see [4,51]): a number of chips have been produced commercially and by University Departments (see [70] for a list). A number of the major semiconductor manufacturers have also produced chips: Intel produced the 80170NX (or Electronically Trainable Artificial Neural Network, ETANN) [69], an essentially analogue device directly implementing a number of neurons. Synapses were implemented using the difference between voltages on two neighboring floating gates. The values were externally determined and nonvolatile, with analogue multipliers at each synapse. IBM produced the ZISC036 (ZISC, for zero instruction set computer) [71], a digital chip implementing a radial basis function with on-chip learning. This chip could load and evaluate a vector in about 4 microseconds. Motorola collaborated with Adaptive Solutions to develop CNAPS [72], which is essentially a specialized DSP device that can be programmed to implement neural

network applications in a highly efficient fashion. Phillips produced Lneuro [73] and Lneuro 2.3 [74], both digital implementations. Both were intended as special-purpose processors used in conjunction with a computer. Lneuro2.3 was intended also for other signal and image-processing applications. Siemens produced the SYNAPSE-3 neurocomputer, based on their MA16 chip [75,76], a digital chip that can be programmed to perform many different neural network algorithms at high speed. Many other smaller companies (and many university departments) also developed neural network chips in the early 1990s.

Very few of these chips appear to be currently in production, even though the technology of neural networks is quite widely applied. There are two reasons why neural network chips have not taken off. The first is that workstation prices have tumbled while at the same time their performance has rocketed. The result is that (1) training up neural networks does not take an unreasonably long time, even when large amounts of data are involved, and (2) using neural network software after training is very fast: real-time performance is often possible without special hardware. Since most users do not really care, how long training takes, so long as recall is fast, there is no commercial advantage in building systems a round neural network chips. The second reason is that neural networks themselves (and therefore neural network chips) are components in larger systems. These systems are required in order to massage the data into a form where it can be used directly with a time-free neural network: generally these systems already require a PC, so adding on some neural network software to complete the system is a much more attractive proposition than adding on neural network hardware. Neural network chips tend to be in use either in specialized defense applications (e.g., Irvine Sensors 3DANN devices, see http://www.irvine-sensors.com) or in visual sensors (e.g. NeuriCam, see www.NeuriCam.com). There is still interest in developing neural network chips for algorithms: the Boltzmann machine's capability for generating representations (and for using noise in the algorithm itself) has led to continuing interest in that algorithm ([77,78]). In addition, there has been interest in hardware for the more recent products of expert algorithms [79], resulting in a mixed-signal (hybrid) implementation [80].

Perhaps a third reason can be added as well: as is clear from the paragraph above, there has been no agreement among chip designers as to the best way to implement this type of device. Technologies have varied from specialized analogue systems to specialized digital systems to systems that were essentially adapted digital signal processors. All these approaches work, but none had a specific competitive edge.

## 5.4 Implementing spiking neurons

The earliest implementations of spike-based neurons used existing pulse-based technologies. Interest in this approach appears to have decreased in favor of more biologically plausible systems based on integrate-and-fire neurons.

### 5.4.1 Pulse-based neuron implementations

Pulse-based techniques have been used in signal processing for many years. Signals take the form of a train of pulses, usually with the signal in an inactive

(zero) state most of the time. Such signals have advantages over level coded signals: they are low power (assuming that power consumption is minimal during the zero period), reasonably noise immune, and easily regenerated if the pulse edge is flattened. There are essentially three basic techniques for coding (modulating) values onto pulses: pulse height modulation, pulse width modulation, and pulse frequency modulation. These techniques are, up to a point, independent of each other. One can argue that these pulse-based techniques do have a degree of neural plausibility: pulse frequency modulation is the same as biological spike-rate coding. One can argue that pulse height modulation is what is happening at synapses, although the postsynaptic smearing of the precise spike timing could also be interpreted as pulse width modulation.

A number of groups have developed pulse-based neural systems. Murray's group [81–83] used pulse frequency modulation for neuron-to-neuron communication, and pulse width modulation inside the neuron for neuron state (or activity). Their chips were used in robot controllers. Richert's group [84] also used pulse height modulation. Hamilton [83] uses pulse height modulation for postsynaptic currents. The systems produced are relatively compact and low power, and can process and produce time-varying signals (for example, by modulating the pulse frequency). One problem is that it takes time to decode such pulse outputs: one needs to sample pulses for some time in order to estimate the value represented by a pulse frequency coded signal. Lehmann describes circuits for implementing classical conditioning [85] and for biologically inspired learning [86] in pulsed neural networks.

### 5.4.2 Point neurons

Point neurons such as leaky integrate-and-fire (LIF) neurons are more accurate models than time-free models because, even although they reduce the dendrite to a single point, they do model behavior in time. The mathematical model for this neuron is described in Eq. (4). Implementing such a neuron can be achieved very directly in discrete analogue electronics, as was shown in Figure 13.7. The capacitor $C$ models the membrane capacitance, and the resistor $R$ models the (constant) membrane leakage (in Eq.( 4), $\tau = RC$). The threshold $\theta$ is modeled using a comparator. Circuitry to generate the spike is required, as is circuitry to discharge the capacitor when a spike is generated. Additional aspects of LIF neurons, such as an absolute refractory periods (the period after spiking during which the neuron cannot fire), relative refractory periods (the period following the absolute refractory period during which it is more difficult to make the neuron fire), postsynaptic current pulse shaping, and spike output shaping can, if required, also be implemented directly in analogue circuitry. The problem with such analogue models in size and complexity: researchers are usually interested in experimenting with networks of LIF neurons, and in adaptation in such networks. It is impractical (or perhaps just unfashionable) to produce discrete analogue implementations of such networks. Such direct implementations are larger, and one has to build each one individually. However, considering the difficulties involved in VLSI implementation, and the fact that many hardware implementations are used for experimenting with relatively small networks (taking advantage

of speed, rather than size), there may still be a place for such discrete component-based systems.

Researchers are often more interested in software and hardware implementations of such networks. Software for such neurons is straightforward to develop. There are two basic techniques used. The direct approach involves modeling the development of the voltage on each neuron using discreticized time (where the timestep is chosen to be small enough to capture the behavior being studied). This approach is useful for small numbers of neurons and can permit the modeling of postsynaptic current pulse shaping. Where large numbers of neurons are to be simulated, this approach can be slow. The alternative is the next spike time approach. In this case, the effect of each spike's arrival is modeled. Membrane voltages are updated only when a spike arrives, relying on the fact that for a neuron with fixed threshold and no noise, spiking is always the direct effect of the most recent excitatory presynaptic pulse. The effect is that the simulation time becomes dependent on the level of spiking and on the degree of interconnection. This technique has been used by [2, 87–89] for simulating large numbers of neurons. In addition, Grassman and Cyprian [89] have developed special-purpose hardware to support this.

Neither of these software techniques will work in real time unless the network being simulated is small. Hardware implementations offer this possibility. Both digital [90, 91] and analogue [58, 92–95] implementations have been built. Digital implementations using the direct approach are attractive, since we can update the representation of the membrane voltage with each timestep. Turning Equation 4 into voltage and discreticizing gives

$$V(t+\Delta t) = V(t) - \frac{\Delta t}{\tau} V(t) + \frac{I(t)\Delta t}{C} \tag{5}$$

where $V(t)$ is the voltage on the membrane, $\Delta t$ is the timestep, and $I(t)$ is the postsynaptic current injected. If we use floating-point arithmetic throughout, this presents few problems. However, using fixed-point (which takes up much less chip space), we run into problems when $\frac{\Delta t}{\tau}$ V (t) or $\frac{I(t)\Delta t}{C}$ disappears because they are less than the smallest number representable. This occurs when either number is less than $\frac{\theta}{2^n}$ for an $n$-bit representation. This problem is serious, particularly for attempts at fully implemented chips [91]. The problem can result in the failure of continuous small inputs to push the $V$ over the threshold. Further, attempting to gain better accuracy for spike times by decreasing $\Delta t$ makes the problem worse. Only increasing the length of the representation really helps.

Including a refractory period (relative or absolute) presents few problems: the absolute refractory period uses a timer, and the neuron simply may not fire during this time. The relative refractory period requires adjusting the value of $\theta$: though not implemented in the examples above, it could be implemented either by setting $\theta$ to a high value and then decrementing it towards its rest value, or using a number of $\theta$ values and setting the values with the aid of another timer.

Analogue implementations suffer from different problems. The most crucial problem is that the timing expected from LIF neurons does not match well with the values of $R$ and $C$ (and hence $\tau$) that can be produced with standard analogue technologies. (Meador's design [56] appears to integrate signals in less than 1 $\mu$s.)

We would like values for $\tau$ of around 20 ms. This would imply that $RC = 0.02$. Capacitors are produced using areas of metal (often deposited aluminium) separated by a thin layer of silicon dioxide. The capacitance is directly proportional to the area, making it impossible to fabricate a number of large capacitors on a single chip. The maximal value realistically achievable is of the order of 1 pF, or $10^{-12}$ F. This value implies a value for $R$ of $2*10^{10}$, or 20 GΩ. Resistors are produced either as tracks of polysilicon or by using transistors with fixed $V_{gs}$ as resistors. The former produces only resistors with low values: the latter can produce much higher values of resistance by utilizing the part of the transistor characteristic just below the transistor's conduction threshold. However, in this region, the drain-source resistance is an exponential function of $V_{gs}$, so precision (and stability) of this resistance requires both precision (stability) in $V_{gs}$. Unless one is willing to manually trim $V_{gs}$ for each neuron, this also requires reproducibility of below-threshold currents across the chip. Chicca et al. [95] used careful layout, with an additional metal layer, but report about 16% variation in leakage current over one chip.

Switched capacitor techniques [96] have been used to increase the value of $R$ achievable, and hence to reduce the value of $C$ that needs to be implemented. Switched capacitor techniques introduce a digital switching signal to partially discharge the capacitor. This results in problems associated with hybrid systems, particularly adding noise to circuitry that is attempting to use precise analogue values. This situation can be problematic, requiring very careful circuit and system design. Additionally, the use of switched circuit designs also can make the precise timing of spike generation (resulting from the activation exceeding the threshold) become phase entrained to the switching signal [97].

Liu and Minch [94] have achieved a degree of adjustment in firing rate in response to perturbations in the neuron's overall input by adapting the integrate-and-fire neuron's threshold upwards in response to each generated spike, and gradually downwards otherwise. The decrease uses a tunneling mechanism with a time constant of seconds or minutes. Indiveri [98] achieves spike frequency adaptation by charging a capacitor. In addition, this low power chip has a refractory period. A different variety of point neuron has been implemented by Patel and DeWeerth [99]. Their approach implements a more complex (but more biologically realistic) model neuron: the Morris–Lecar model [100]. Their aVLSI implementation is particularly relevant to the design of neural oscillators, since it can produce outputs with frequencies in the range of 0.1 Hz to 1 Khz, depending on circuit parameters.

### 5.4.3 Synapses for spiking neurons

Spiking neuron synapses receive a train of pulses, rather than values. These spike trains are digital in the sense that a spike is an all-or-nothing event, yet analogue in the sense that in an unclocked implementation, the spike time is unconstrained. Although real neuron spikes are of the order of 1ms in duration, implemented spikes are often much shorter (about 100 ns in [93]), or they may be coded simply as event times, with no duration assumed at all. Implementing these synapses means translating these pulse trains (or event lists) into activity changes. One way of achieving this outcome is to inject a small amount of current for each spike. The exact amount and the direction of current injection depend on the

synaptic weight and on whether the weight is inhibitory or excitatory. Such current pulses may be fixed length and height modulated (as in [83, 93]), or could use other pulse modulation techniques. The use of pure pulse-based techniques does tend to result in relatively small synapses [81, 83].

Simple modulated current injection for each spike assumes that the shape of the postsynaptic current is rectangular. One result of this is that if the activity is near threshold, and a spike arrives at an excitatory synapse, then the threshold is instantly reached, and the postsynaptic neuron fires at once. Though there are occasions when this outcome can be useful, resulting in instant synchronization of firing neurons, it is certainly not biologically realistic, and can cause problems if neurons are reciprocally connected without a refractory period. In simulations, the effect of the synapse is often approximated by an alpha function, $\alpha t \exp(-\alpha t)$: in hardware implementations, the noninstantaneous effect of the synapse can be implemented using capacitances (only really practical in subthreshold aVLSI where minute currents are used), or by using a table lookup (in a digital system).

Weight storage and manipulation can be the same as for time-free neurons. The time parameter means that there are additional options in terms of synaptic weight changes. In addition to the long-lasting changes discussed earlier, synapses may have shorter-term changes—for example they may be depressing or facilitating (see Section 2). A simple depressing synapse has been implemented by Rasche and Hahnloser [101]. The weight on this synapse is set by the charge on a capacitor, which each incoming spike discharges. This capacitor is slowly being charged up to its maximal level (which corresponds to the synapse's original weight). The result is that a sequence of closely spaced presynaptic spikes have a gradually decreasing effect: if, however, there is then a gap, the synapse recovers to its initial weight. Liu and Minch [94] have also implemented a depressing synapse, but with a longer time constant: their work is aimed at maintaining neural processing in the face of rising input spike frequencies.

A number of different mechanisms for altering weights in spiking neuron networks have been suggested. Some of these are extensions of techniques used in time-free networks. Maass has suggested how spiking neuron firing times might be interpreted in order to implement a spiking neuron equivalent of the backpropagation learning algorithm [102]. However, such rules do not take advantage of the capabilities of spiking networks to use patterns over time, and have low biological plausibility. Designers of spiking neural networks have generally been interested in more biologically plausible rules, perhaps because there has not been a spiking equivalent of a perceptron network or a Delta rule. Instead, such designers have been interested in variations on the original Hebbian learning rules, particularly temporally asymmetric Hebbian learning [40].

There has been considerable interest in the implementation of such rules. Bofill et al. [103] have produced one possible circuit. This form of a VLSI implementation has been used to detect synchrony by taking advantage of the tendency of this implementation of the rule towards making weights go to one of their endpoints [104]. Chicca et al. [95] have implemented a bistable excitatory Hebbian synapse. Paired presynaptic spiking input and postsynaptic neural activity result in the synapse being strengthened towards its higher level, but otherwise the synapse decays towards its lower level. There is also a stochastic element in the

synaptic strength variation. In [105] these, authors report that each synapse uses 14 transistors.

One specific synapse that has received a great deal of attention is the synapse between the inner hair cell of the the organ of Corti (in the cochlea, in the inner ear) and the neurons of the spiral ganglion whose axons form the auditory nerve. The reason for interest in this synapse is that this synapse is part of the transduction of the movement of the membranes in the cochlea into a neural signal, and maintaining precise timing is known to be important for finding the direction of sound. Software models have been built (reviewed in the similar manner as in [106]), as have hardware implementations. These often include depression (since the biological synapses appear to be depressing). Hardware implementations are popular, as they permit real-time implementations of biologically inspired auditory models. The first silicon implementation is discussed in [107], and the field is reviewed in [108]. The most sophisticated version is in [109].

### 5.4.4 Interconnecting spiking neuron systems

Single chips may contain a number of spiking neuron implementations, and for small networks, it is sometimes possible to produce the whole network on a single chip. In general, however, one will want to connect up neurons on different chips. In addition, it is often the case that the inputs to the network and the outputs from the network will be required off-chip. On chips that contain a small number of neurons, one can connect neurons and the appropriate synapse using point-to-point wiring. For larger numbers of neurons, this approach is impractical.

The address/event representation (AER) was introduced for this purpose (see [110] for a tutorial introduction). This is a time-division multiplexing system that uses a digital bus to transfer spikes from neurons to the appropriate synapses. It allows for interconnection to be described in biologically natural ways, and also for reprogrammable configuration. "Virtual" wiring is possible as well. There is ongoing work on chip-based support at the Institute for Neuroinformatics in Zurich, Switzerland.

## 5.5 Implementing more complex neuron models

Many researchers are not satisfied with time-free or point neuron models. It is well known that real neurons are far more complex than either of these models. The computational properties of time-free neural models have been well investigated over many years. Networks of point neurons and learning mechanisms for point neurons are still under research. Point neurons make the implicit assumption that there is no interaction between the different inputs that arrive on the dendrite. Even although relatively complex postsynaptic functions may be used, what arrives at the thresholding element is simply the (linear) sum of these inputs.

Yet there is a school of thought (discussed in [111]), which holds that the spikes from neurons are simply the mechanism whereby neurons communicate their results, and that complex processing can take place on the neuron itself, possibly even without any spiking occurring at the axon hillock. Such a view seems attractive when one considers both the complex morphology of many neurons

and the nonuniform placing of ion channels on these neurons. Even the briefest inspection of neural images shows that the dendrites have very considerable complexity: indeed, many types of neuron are differentiated by their dendrite shapes.

The limiting factors in the accuracy of neuron simulation are time and space. One could model neurons right down to the molecular or atomic level. Before a researcher produces a model, the researcher normally has some particular idea that they want to investigate. More complex models of full neurons have normally been either compartmental models or models of dendrites: others have gone further and have modeled patches of membrane (though such models are rarely modeling full neurons).

### 5.5.1 Multicompartment neurons

Software implementations of compartmental models model the dendrites, cell body, and axon as an interconnected set of cylinders and branches. Each modeled element has its inputs and outputs to and from adjacent elements, as well as its various cross-membrane leakage currents modeled. In addition, postsynaptic currents from model synapses can be included in the modeled elements. The most prevalent package for this is Neuron [39]. This form of model is generally slow, though this depends on the number of compartments being modeled. However, even although hardware implementation would clearly be faster, it is rarely attempted, primarily because such simulations are carried out with a view to understanding detailed neuron operation (for example, the effects of synapses on distal and proximal dendrites, and the effects of branching both in dendrites and axons) rather than actual information processing.

There has been more interest in hardware implementation of dendrites. Extending the dendrite beyond a single point means that the activity of the neuron is no longer a single value but is a function of location as well as time. Further, the precise time ordering of presynaptic signals will have an effect on this activity. Mel [8] has provided a major review of information processing on the dendrite, concluding that dendrites from single neurons could perform logical operations or discriminate between images. Elias [112] and Northmore and Elias [113] have developed an analogue VLSI dendrite implementation which can process spike trains. In [114], they have used switched capacitor techniques to achieve the range of membrane resistances required. Simple dendritic processing has been used to design an aVLSI chip that is sensitive to the direction of motion [115]. In [116], learning in dendritic systems is emulated. There is current interest in combining model dendrites with temporal Hebbian learning: recent research suggests that the precise timing of presynaptic and postsynaptic signals [40], and the location of the synapse on the dendrite [117], can affect the way in which weights characterizing synapses alter. Dendritic models are usually combined with spike-generating entities, and sometimes with models of delay in the axon, due to axon diameter (wide axons conduct faster) and even myelinization[2] to produce models of whole neurons in which precise spike timing can be modeled.

[2]Myelin is a protein produced by glial brain cells. It is often wrapped around axons, reducing both their leakage and their capacitance, and allowing much faster transfer of action potentials (see [5], chapter 4).

### 5.5.2 Implementing models of excitable membranes

The lowest level of neural modeling currently attempted is modeling of excitable membranes. The impetus for producing such models is clear: as discussed in Section 2, ion channels embedded in the membrane are the primary mechanism whereby the potential of the neuron is modified or altered. The aim of this work has generally been "explanatory neuroscience" [118], rather than biologically inspired computing. It is not possible to emulate multiple different yet interacting ion species directly in electronics. Electronic systems have only one charge carrier, the electron. Similarly, one cannot model multiple varieties of voltage-sensitive (and ion-type-sensitive) ion channels. These can be modeled in software, but such models are slow and complex.

The idea of using subthreshold FETs to emulate the exponential conductance properties of ion channels is discussed at length in Mead's book [62], where he calls it *eclectronics*. A highly influential implementation of the spiking characteristics of the soma and axons was produced by Mahowald and Douglas [119]. This aVLSI implementation implements bulked ion channels (rather than individual ones) and is essentially a silicon compartment model. It was the first to achieve this goal in hardware and thus to operate in real time. A more detailed discussion of the elements of this chip can be found in [120]. A number of other authors have followed this early start: Rasche, Douglas, and Mahowald [121] added extra conductances, and Rasche and Douglas [122] have developed this concept and have produced a more robust chip. Both [119] and [122] implement these ion channels as a circuit, rather than as a single transistor, as implied by Mead. Implementing ion channels as single transistors was attempted in [123]: however, it proved difficult to get the range of behaviors one would want from a range of different types of ion channels. Shin and Koch [124] provide an aVLSI implementation of an adaptive algorithm that permits an electronic neuron to enable it to adapt its current threshold to the mean of the input current. Rasche [125] has produced aVLSI adaptive dendrite that can operate in widely varying levels of overall neural activity. This form of adaptation allows the dendrite to signal changes from the short-term mean of their input. Rasche and Douglas [126] have developed the silicon axon so that it can support both forward and backward propagation of spikes. Minch et al. [127] have produced a silicon axon that recreates a pulse along its length.

Real synapses, of course, are not simple mulipliers. One form of synapse (a chemical synapse: see [5] chapter 10) consists in essence of a set of ion channels on the postsynaptic membrane that are opened when neurotransmitter is released presynaptically. This occurs in response to presynaptic action potentials. Such a synapse has been implemented in aVLSI by Rasche and Douglas [128], where they provide equivalent circuits for (bulk) AMPA and NMDA conductances.

### 5.5.3 Applications of hardware spiking neurons

What evidence is there that more sophisticated neural hardware, such as that of point neurons, might have application, when those for time-free neurons (discussed in Section 5.3.2) have proven largely a graveyard for silicon implementations? Firstly, these chips can process time-varying signals directly. They do not

require the signal to be sampled initially. Thus a minimum of extra hardware is required (bringing the signal into the desired voltage/current range, or pulse coding it, for example), greatly simplifying the direct interfacing of the neural network system with the devices providing input and accepting output from the network. If interfacing the chip does not entail using a PC, then there is more advantage to be gained from direct hardware implementation.

Although such silicon neural systems have not yet found industrial applications, there have been applications for this type of technology in the neuromorphic field. These applications vary from line following in a robot [129] to sound direction finding [130, 131], including sonar [132], to real-time image analysis [133, 134] to motor control [135]. They have been applied particularly in autonomous systems, where the simplicity of interfacing the implemented neuron to the rest of the system has been important. Even where digital computers are part of the overall system, there are still advantages in using hardware-implemented neural systems, particularly at the sensor-processing end of the system. Their explicit parallelism can permit effective real-time exploitation of the signals being interpreted, distributing the processing in an effective way.

The other application area for hardware neural implementations is in modeling and interfacing to real neural systems. One interesting example of modeling neural systems is Tobi Delbruck's "Physiologist's Friend" chip [136], a model of a visual cortical neuron with retinal sensors that can model the receptive field of a visual cortical neuron well enough to be used instead of a live animal for training psychology or physiology students. In addition, spiking silicon neurons are one of the underlying technologies that may permit effective sensory implants [137], both auditory [138] and visual [139]. These prosthetic applications may prove to be an important growth area for this type of technology, where small size and ultra-low power consumption are critical.

There is also rather less disagreement about the most appropriate technologies to use for implementing these systems. Most implementations are either analogue or hybrid, using aVLSI (often largely subthreshold, partly because of its low power consumption and partly to take advantage of its nonlinear circuit elements) and sometimes combining this with pulse techniques. One recent paper [140] uses a mixture of excitatory and inhibitory neurons, implemented in subthreshold aVLSI, with separate dendrites for different types of input. The analogue circuitry produces an essentially digital output, using strong positive feedback to provide a robust selection output—robust against the actual level of the input. This mixture of analogue and digital, inspired by biology yet not constrained to follow it exactly, is conceptually reasonably simple (and thus efficiently implementable) and able to implement an algorithm. This approach may represent a direction that could lead to a greater range of applications.

## 6 CONCLUSION

Modeling neurons at a number of different levels has uncovered a number of what appear to be computational principles of the brain. These have then been used in electronic systems or in software and where appropriate in hardware as well. Neural network technology is now well established. Whether the novel com-

putational paradigms from more sophisticated model neurons will prove useful remains to be seen. Initial applications seem to suggest that the first areas of application will be in what is currently the niche area of autonomous systems. Other research areas (with titles like "the disappearing computer" or "the ubiquitous computer") suggest that greater autonomy for computer-based systems will be required, so this niche area may well come to be more important.

It is, however, still the case that brains can do many things that are not possible in current electronic systems. Neuromorphic systems have been proposed as one set of techniques for capturing some of these capabilities. They have indeed helped to explain some of the brain's sensory capabilities, particularly in vision and in motor control. Yet the deeper, less peripheral capabilities of brains remain essentially untouched. It is an open question as to which, if any, of the other aspects of neural systems apart from those already modeled might provide a clue as to the nature of these capabilities. Currently, spiking systems are being investigated by many laboratories. These certainly show promise for parallel processing of time-varying signals. However, so far, investigation of spiking systems has thrown no light on awareness, self-consciousness, or indeed, consciousness. Even planning is still entirely in the domain of old-fashioned software.

There are a number of candidate "biotechnologies" for possible further investigation. These range from the interactions between the different ion types gated by the zoo of ion channels found in neurons to interactions between elements of neurons at the quantum level (as suggested by Hammeroff and Penrose). Modeling these systems in software or hardware presents one way of investigating these possibilities. There are other possibilities as well, such as producing hybrid machines, part electronic and part neural [141].

There are difficulties in producing simulations of interacting ions or of systems at the quantum level on normal computers. Such computers are inherently deterministic, and this makes the modeling of stochastic or quantum systems slow and cumbrous. It is possible that Moore's Law will come to the rescue: as feature sizes decrease, gates and transistors become more noisy do to various noise effects, making the emulation of stochastic systems in hardware much simpler (even if it does make building deterministic systems that much harder). It may yet be that there are general principles of another sort of computation grounded in this stochasticity, and that understanding these using modeling will provide some other general principles, perhaps even shedding light on some of the brain's deeper capabilities.

## ACKNOWLEDGMENTS

The support of the UK EPSRC (grant number GR/R64654) is acknowledged.

## REFERENCES

[1] P. Hammarlund and O. Ekeberg (1998): Large neural network simulations on multiple hardware platforms. *Journal of Computational Neuroscience 5*, 443–459.

[2] E. Claverol, A. Brown, and J. Chad (2001): Scalable cortical simulations on Beowulf architectures. *Neurocomput. 43*, 307–315.
[3] D. Hammerstrom (2001): Biologically inspired computing. [Online]. Available: http://www.ogi.ece.edu/strom
[4] Neural network hardware. [Online]. (1998): Available: http://neuralnets.web.cern.ch/NeuralNets/nnwlnHepHard.html
[5] E. Kandel, J. Schwartz, and T. Jessell (2000): *Principles of Neural Sci*. (4th Ed.) McGraw Hill.
[6] C. Koch (1999): *Biophysics of Computation*. Oxford.
[7] T. Bell (1991): A channel space theory of dendritic self-organisation. AI Laboratory, Free University of Brussels, Tech. Rep. 91–4.
[8] B. Mel (1994): Information processing in dendritic trees. *Neural Comput. 6*, 1031–1085.
[9] D. Aidley (1999): *The Physiology of Excitable Cells*. (4th Ed.) Cambridge University Press.
[10] S. Hammeroff (1999): The neuron doctrine is an insult to neurons. *Behavioural and Brain Sciences, 22*, 838–839.
[11] W. McCulloch and W. Pitts (1943): A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics, 5*, reprinted in [142].
[12] D. Hebb (1949): *The Organization of Behavior*. Wiley, New York. partially reprinted in [142].
[13] J. Anderson (1995): *An Introduction to Neural Networks*. Cambridge, MA: MIT Press.
[14] F. Rosenblatt (1962): *Principles of Neurodynamics*. Spartan, New York.
[15] J. Hertz, A. Krogh, and R. Palmer (1991): *Introduction to the Theory of Neural Computation*. Addison Wesley.
[16] S. Haykin (1999): *Neural Networks: A Comprehensive Foundation*. (2nd Ed.) Macmillan.
[17] B. Widrow and M. Hoff (1960): Adaptive switching circuits, In *1960 IRE WESCON Convention Record*. New York: IRE, *4*, 96–104.
[18] R. Rescorla and A. Wagner (1972): A theory of pavlovian conditioning: The effectiveness of reinforcement and nonreinforcement. In *Classical Conditioning II: Current Research and Theory* (A. Black and W. Prokasy, eds) Appleton-Century-Crofts, New York: 64–69.
[19] M. Minsky and S. Papert (1969): *Perceptrons*. MIT Press, Cambridge partially reprinted in [142].
[20] J. Hopfield (1982): Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences. USA, 79*, 1982, reprinted in [142].
[21] D. Ackley, G. Hinton, and T. Sejnowski (1985): A learning algorithm for boltzmann machines. *Cognitive Science, 9*, reprinted in [142].
[22] A. Bryson and Y.-C. Ho (1969): *Applied Optimal Control*. Blaisdell, New York.
[23] P. Werbos (1974): Beyond regression: New tools for prediction and analysis in the behavioral sciences. Ph.D. dissertation, Harvard University.
[24] D. Parker (1985): Learning logic. Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, MA, Tech. Rep. TR–47.

[25] Y. Le Cun (1985): Une procédure d'apprentissage pour réseau à seuil assymétrique. In *Cognitiva 85: A la Frontière de l'Intelligence Artificielle des Sciences de la Connaissance des Neurosciences*, (Paris 1985). CESTA, Paris: 599–604.

[26] D. Rumelhart, G. Hinton, and R. Williams (1986): Learning representations by back-propagating errors. *Nature, 323*, 533–536, reprinted in [142].

[27] J. Moody and C. Darken (1988): Learning with localized receptive fields. In *Proceedings of the 1988 Connectionist Models Summer School*, (D. Touretzky, G. Hinton, and T. Sejnowski, eds) (Pittsburg). Morgan Kaufmann, San Mateo 133–143.

[28] C. Bishop (1995): *Neural networks for Pattern Recognition*. Clarendon Press, Oxford.

[29] J. Elman (1990): Finding structure in time. *Cognitive Science. 14*, 179–211.

[30] H. Barlow (1959): Sensory mechanisms, the reduction of redundancy and intelligence. *The Mechanisation of Thought Processes*: *NPL Symposium, 10*.

[31] T. Kohonen, T. Huang, and M. Schroeder (2000): *Self-organizing Maps*. (3rd ed.) Springer-Verlag.

[32] L. Lapique (1907): Sur l'excitation electrique des nerfs. *J. Physiology. Paris*, 620–635.

[33] W. Gerstner (1995): Time structure of the activity in neural network models. *Physical Reviews E. 51*, 738–758.

[34] W. Gerstner and W. Kistler (2002): *Spiking Neural Models*. Cambridge.

[35] J. Feng and D. Brown (2000): Integrate-and-fire models with nonlinear leakage. *Bulletin of Mathematical Biology. 62*, 467–481.

[36] J. Feng and G. Wei (2001): Increasing inhibitory input increases neuronal firing rate: when and why? Diffusion process cases. *J. Phys. A. 34*, 7493–7509.

[37] E. Izhikevich. Which model to use for cortical spiking neurons? submitted to *IEEE Transactions of Neural Networks*.

[38] —— , Simple model of spiking neurons, accepted for publication in *IEEE Transactions of Neural Networks*.

[39] M. Hines and N. Carnevale (1997): The NEURON simulation environment. *Neural Computation. 9*, 1179–1209.

[40] G. Bi and M. Poo (2001): Synaptic modification by correlated activity: Hebb's postulate revisited. *Annual Review of Neuroscience. 24*, 139–166.

[41] L. Smith (2002): Using Beowulf clusters to speed up neural simulations. *Trends in the Cognitive Science. 6*, 231–232.

[42] R. Fitzhugh (1966): An electronic model of the nerve membrane for demonstration purposes. *J. Appl. Physiology. 21*, 305–308.

[43] R. Johnson and G. Hanna (1969): Membrane model: a single transistor analog of excitable membrane. *J. Theoretical Biology. 22*, 401–411.

[44] E. R. Lewis (1968): An electronic model of the neuroelectric point process. *Kybernetik. 5*, 30–46.

[45] G. Roy (1972): A simple electronic analog of the squid axonmembrane: the neuro FET. *IEEE Transactions on Biomedical Engineering*. BME-18, 60–63.

[46] W. Brockman (1979): A simple electronic neuron model incorporating both active and passive responses. *IEEE Transactions on Biomedical Engineering*. BME-26, 635–639.
[47] F. Rosenblatt (1958): The perceptron: a probabilistic mode for information storage and processing in the brain. *Psychological Rev. 65*, 386–408.
[48] B. Widrow (1962): Generalization and information storage in networks of ADALINE neurons. In *Self-Organizing Systems* (G. Yovitts, ed) Spartan Books.
[49] R. Runge, M. Uemura, and S. Viglione (1968): Electronic synthesis of the avian retina. *IEEE Transactions on Biomedical Eng*., BME-15, 138–151.
[50] L. Smith (1989): Implementing neural networks. In *New Developments in Neural Computing* (J. Taylor and C. Mannion, eds) Adam Hilger, 53–70.
[51] I. Aybay, S. Cetinkaya, and U. Halici (1996): Classification of neural network hardware. *Neural Network World. 6*(1), 11–29.
[52] "AN220E04 datasheet: Dynamically reconfigurable FPAA," Anadigm, 2003.
[53] R. Hecht-Nielsen, *Neurocomputing*. Addison-Wesley, 1990.
[54] E. Vittoz, H. Oguey, M. Maher, O. Nys, E. Dijkstra, and M. Cehvroulet (1991): Analog storage of adjustable synaptic weights. In *VLSI Design of Neural Networks*. (U. Ramacher and E. Rueckert, eds) Kluwer Academic.
[55] "80170nx electrically trainable analog neural network," Intel Corporation, 1991.
[56] J. Meador, A. Wu, C. Cole, N. Nintunze, and P. Chintrakulchai (1991): Programmable impulse neural circuits. *IEEE Transactions on Neural Networks. 2*(1), 101–109.
[57] C. Diorio, P. Hasler, B. Minch, and C. Mead (1996): A single-transistor silicon synapse. *IEEE Transactions on Electron Devices. 43*(11), 1982–1980.
[58] L. Smith, B. Eriksson, A. Hamilton, and M. Glover (1999): SPIKEII: an integrate-and-fire aVLSI chip. *Int. J. Neural Syst. 9*(5), 479–484.
[59] D. Hsu, M. Figueroa, and C. Diorio (2002): Competitive learning with floating-gate circuits. *IEEE Transactions on Neural Networks. 13*, 732–744.
[60] T. Morie, T. Matsuura, M. Nagata, and A. Iwata (2003) A multinanodot floating-gate mosfet circuit for spiking neuron models. *IEEE Transactions on Nanotechnology. 2*, 158–164.
[61] D. Green (1999) *Digital Electronics* (5th ed.) Prentice Hall.
[62] C. Mead (1989): *Analog VLSI and Neural Systems*. Addison-Wesley.
[63] S.-C. Liu, J. Kramer, G. Indiveri, T. Delbruck, and R. Douglas (2002): *Analog VLSI: Circuits and Principles*. MIT Press.
[64] E. Ifeachor and B. Jervis (2002): *Digital Signal Processing: A Practical Approach* (2nd ed.) Prentice Hall.
[65] M. Hohfield and S. Fahlman (1997): Probabilistic rounding in neural network learning with limited precision. *Neurocomputing. 4*, 291–299.
[66] E. Sackinger (1997): Measurement of finite precision effects in handwriting and speech recognition algorithms. In *ICANN 97: LNCS 1327* (W. Gerstner, A. Germond, M. Hasler, and J.-D. Nicoud, eds), Springer Verlag, 1223–1228.
[67] P. Moerland and E. Fiesler (1997): Neural network adaptations to hardware implementations. In *Handbook of Neural Computation* (E. Fiesler and R. Beale, eds) IOP Publishing.

[68] S. Draghici (2002): On the capabilities of neural networks using limited precision weights. *Neural Networks. 15*, 395–414.
[69] I. Corporation (1990): 80170NN electrically trainable analog neural network. *Datasheet*.
[70] C. S. Lindsey, B. Denby, and T. Lindblad. Neural network hardware. [Online]. Available: http://www.avaye.com/ai/nn/hardware/index.html
[71] A. Eide (1994): An implementation of the zero instruction set computer (zisc036) on a pc/isa-bus card, [Online]. Available: citeseer.nj.nec.com/eide94implementation.html
[72] H. McCartor (1991): Back propagation implementation on the adaptive solutions cnaps neurocomputer chip. In *Advances in Neural Information Processing Systems 3*, (R. Lippmann, J. Moody, and D. Touretzky, eds), Morgan Kaufmann pp. 1028–1031.
[73] N. Mauduit, M. Duranton, and J. Gobert (1992): Lneuro 1.0: A piece of hardware LEGO for building neural network systems. *IEEE Transactions on Neural Networks. 3*(3).
[74] Y. Deville (1995) Digital VLSI neural networks: from versatile neural processors to application-specific chips. *Proc. of the International Conference on Artificial Neural Networks ICANN'95*, Paris, France, Industrial Conference, Session 9, VLSI and Dedicated Hardware.
[75] U. Ramacher, J. Beichter, W. Raab, J. Anlauf, N. Bruels, U. Hachmann, and M. Weseling (1991): Design of a 1st generation neurocomputer. In *VLSI Design of Neural Networks*, (U. Ramacher and E. Rueckert, eds), Kluwer Academic.
[76] U. Ramacher, W. Raab, J. Anlauf, U. Hachmann, J. Beichter, N. Bruls, R. Manner, J. Glas, and A. Wurz (1993): Multiprocessor and memory architecture of the neurocomputer SYNAPSE-1. *Proc. International Conference on Microelectronics for Neural Networks*. Edinburgh, pp. 227–232.
[77] H. Chen and A. Murray (2002): A continuous restricted Boltzmann machine with a hardware amenable training algorithm. In *Proceedings of ICANN 2002*, pp. 426–431.
[78] — , A continuous restricted Boltzmann machine with an implementable training algorithm. In *IEEE Proceedings on Vision Image and Signal Processing*.
[79] G. Hinton, B. Sallans, and Z. Ghahramani (1999): A hierarchical community of experts. In *Learning in Graphical Models* (M. Jordan, ed) MIT Press pp. 479–494.
[80] P. Fleury and A. Murray (2003): Mixed-signal VLSI implementation of the products of experts' contrastive divergence learning scheme. In *Proceedings of ISCAS 2003. 5*, pp. 653–656.
[81] A. Murray, L. Tarassenko, H. Reekie, A. Hamilton, M. Brownlow, D. Baxter, and S. Churcher (1991): Pulsed silicon neural nets—following the biological leader. In *Introduction to VLSI Design of Neural Networks* (U. Ramacher, ed), Kluwer pp. 103–123.
[82] A. Murray, S. Churcher, A. Hamilton, A. Holmes, G. Jackson, R. Woodburn, and H. Reekie (1994) Pulse-stream VLSI neural networks. *IEEE MICRO*, pp. 29–39.

[83] A. Hamilton, S. Churcher, P. Edwards, G. B. Jackson, A. Murray, and H. Reekie (1994): Pulse-stream VLSI circuits and systems: the EPSILON neural network chipset. *Int. J. Neural Sys. 4*(4), 395–405.

[84] P. Richert, L. Spaanenburg, M. Kespert, J. Nijhuis, M. Schwarz, and A. Siggelkow (1991): ASICs for proto-typing pulse-density modulated neural networks. In *Introduction to VLSI Design of Neural Networks* (U. Ramacher, ed), Kluwer pp. 125–151.

[85] T. Lehmann (1997): Classical conditioning with pulsed integrated neural networks: Circuits and system. pt. II, *IEEE Transactions on Circuits and Systems, 45(6)*, 720–728.

[86] T. Lehmann and R. Woodburn (1999): Biologically-inspired learning in pulsed neural networks. In *Learning on Silicon: Adaptive VLSI Neural Systems* (G. Cauwenberghs and M. Bayoumi, eds) Kluwer, pp. 105–130.

[87] L. Watts (1993): Event driven simulation of networks of spiking neurons. In *Advances in Neural Information Processing Systems 6* (J. Alspector, J. Cowan, and G. Tesauro, eds), pp. 927–934.

[88] A. Nishwitz and H. Glünder (1995): Local lateral inhibition—a key to spike synchronization. *Biological Cybernetics. 73*(5), 389–400.

[89] L. Smith, B. Eriksson, A. Hamilton, and M. Glover (1999): Fast digital simulation of spiking neural networks and neuromorphic integration with SPIKELAB. *Int. J. Neural Sys. 9*(5), 473–478.

[90] S. Lim, A. Temple, S. Jones, and R. Meddis (1998): Digital hardware implementation of a neuromorphic pitch extraction system. In *Neuromorphic Systems: Engineering Silicon from Neurobiology* (L. Smith and A. Hamilton, eds), World Scientific.

[91] N. Mtetwa, L. Smith, and A. Hussain (2000): Stochastic resonance and finite resolution in a network of leaky integrate-and-fire neurons. In *Artificial neural networks—ICANN 2002*. Springer, Madrid, Spain pp. 117–122.

[92] S. Wolpert and E. Micheli-Tzanakou (1996): A neuromime in VLSI. *IEEE Transactions on Neural Networks, 7*(2), 300–306.

[93] M. Glover, A. Hamilton, and L. Smith (1998): Analogue VLSI integrate and fire neural network for clustering onset and offset signals in a sound segmentation system. In *Neuromorphic Systems: Engineering Silicon from Neurobiology* (L. Smith and A. Hamilton, eds), pp. 238–250.

[94] S.-C. Liu and B. A. Minch (2001): Homeostasis in a silicon integrate and fire neuron. In *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA* (T. K. Leen, T. G. Dietterich, and V. Tresp, eds), MIT Press, pp. 727–733.

[95] E. Chicca, D. Badoni, V. Dante, M. D'Andreagiovanni, G. Salina, L. Carota, S. Fusi, and P.D. Giudice (2003): A vlsi recurrent network of integrate-and-fire neurons connected by plastic synapses with long term memory. *IEEE Transactions on Neural Network. 14*(5), 1409–1416.

[96] J. Mavor, M. Jack, and P. Denyer (1983): *Introduction to MOS LSI Design*. Addison Wesley.

[97] B. Eriksson (2002): A critical study of a hardware integrate-and-fire neural network. Master's thesis, University of Stirling, Department of Computing Science and Mathematics.

[98] G. Indiveri (2003): A low-power adaptive integrate-and-fire neuron circuit. In *Proc. IEEE International Symposium on Circuits and Systems*. May 2003.
[99] G. Patel and S. P. DeWeerth (1997): Analog VLSI Morris-Lecar neuron. *Electronics Letters, 33*, 997–998.
[100] C. Morris and H. Lecar (1981): Voltage oscillations in the barnacle giant muscle fiber. *Biophysics J. 35*, 193–213.
[101] C. Rasche and R. Hahnloser (2001): Silicon synaptic depression. *Biological Cybernetics. 84*, 57–62.
[102] W. Maass (1997): Networks of spiking neurons: The third generation of neural network models. *Neural Networks. 10* (9), 1659–1671.
[103] A. Bofill, R. Woodburn, and A. Murray (2001): Circuits for VLSI implementation of temporally-asymmetric Hebbian learning. In *Neural Information Processing Systems*. Vancouver.
[104] A. Bofill-i-Petit and A. Murray (2003): Synchrony detection by analogue VLSI neurons with bimodal STDP synapses. accepted for NIPS 2003.
[105] E. Chicca, G. Indiveri, and R. Douglas (2003): An adaptive silicon synapse. In *Proc. IEEE International Symposium on Circuits and Systems*. May.
[106] M. Hewitt and R. Meddis (1991): An evaluation of eight computer models of mammalian inner hair-cell function. *J. Acoustical Soc. Am. 90*(2), 904–917.
[107] J. Lazzaro and C. Mead (1989): Circuit models of sensory transduction in the cochlea. In *Analog VLSI Implementations of Neural Networks*. Kluwer pp. 85–101.
[108] I. Grech, J. Micallef, and T. Vladimirova (1999): Silicon cochlea and its adaptation to spatial localisation. *IEE Proceedings—Circuits Devices and Systems. 146*(2), 70–76.
[109] A. van Schaik and A. McEwan (2003): An analog VLSI implementation of the meddis inner hair cell model. *EURASIP J. Applied Signal Processing*.
[110] K. Boahen, Point-to-point connectivity between neuromorphic chips using address-events. *IEEE Transactions on Circuits and Systems II*. 47(*5*), 416–434.
[111] I. Segev, M. Rapp, Y. Manor, and Y. Yarom (1992): Analog and digital processing in single nerve cells: dendritic integration and exonal propagation. In *Single Neuron Computation* (T. McKenna, J. Davis, and S. Zornetzer, eds) pp. 173–198.
[112] J. Elias (1993): Artificial dendritic trees. *Neural Comput. 5*(4), 648–664.
[113] D. Northmore and J. Elias (1996): Spike train processing by a silicon neuromorph: The role of sublinear summation in dendrites. *Neural Comput. 8*(6), 1245–1265.
[114] J. Elias and D. Northmore (1995): Switched-capacitor neuromorphs with wide-range variable dynamics. *IEEE Transactions on Neural Networks. 6*(6), 1542–1548.
[115] M. Ohtani, H. Yamada, K. Nishio, H. Yonezu, and Y. Furukawa (2002) Analog LSI implementation of biological direction-sensitive neurons. part 1 *Japanese Journal of Applied Physics, 41*, 1409–1416.
[116] W. Westerman, D. P. Northmore, and J. G. Elias (1998): A hybrid (hardware/software) approach towards implementing hebbian learning in silicon neurons with passive dendrites. In *Neuromorphic Systems: Engineering Silicon from Neurobiology*. (L. Smith and A. Hamilton, eds), World Scientific.

[117] A. Saurdagiene, B. Porr, and F. Woergoetter (2004): How the shape of pre- and post-synaptic signals can influence STDP: A biophysical model, accepted for *Neural Comput*.

[118] R. Douglas, M. Mahowald, and K. Martin (1996): Neuroinformatics as explanatory neuroscience. *Neuroimage*. S25–S27.

[119] M. Mahowald and R. Douglas (1991): A silicon neuron. *Nature, 354* (6354), 515–518.

[120] R. Douglas and M. Mahowald (1995): A construction set for silicon neurons. In *An Introduction to Neural and Electronic Networks* (S. Zornetzer, J. L. Davis, C. Lau, and T. McKenna, eds) Academic Press pp. 277–296.

[121] C. Rasche, R. Douglas, and M. Mahowald (1998): Characterization of a silicon pyramidal neuron. In *Neuromorphic Systems: Engineering Silicon from Neurobiology* (L. Smith and A. Hamilton, eds) World Scientific.

[122] C. Rasche and R. Douglas (2001): An improved silicon neuron. *Analog Integrated Circuits and Signal Processing. 23*(3), 227–236.

[123] C. Breslin and L. Smith (1999): Silicon cellular morphology. *International Journal of Neural Systems. 9*(5), 491–495.

[124] J. Shin and C. Koch (1999): Adaptive neural coding dependent con the time-varying statistics of the somatic input current. *Neural Computation. 11*(8), 1893–1913.

[125] C. Rasche (1999): An aVLSI basis for dendritic adaptation. *IEEE Transactions on Circuits and Systems II. 48*(6), 600–605.

[126] C. Rasche and R. Douglas (2001): Forward- and backpropagation in a silicon dendrite. *IEEE Transactions on Neural Networks. 12*(2).

[127] B. A. Minch, P. Hasler, C. Diorio, and C. Mead (1995): A silicon axon. In *Advances in Neural Information Processing Systems* (G. Tesauro, D. Touretzky, and T. Leen, eds) *7*. The MIT Press, pp. 739–746.

[128] C. Rasche and R. Douglas (1999): Silicon synaptic conductances. *J. Comput. Neuroscience*. 7(1), 33–39.

[129] R. Mudra and G. Indiveri (1999): A modular neuromorphic navigation system applied to line following and obstacle avoidance tasks. In *Experiments with the Mini-Robot Khepera: Proceedings of the 1st International Khepera Workshop* (A. A. Loeffler, F. Mondada, and U. Rueckert, eds), pp. 99–108.

[130] C. Schauer, T. Zahn, P. Paschke, and H. Gross (2000): Binaural sound localization in an artificial neural network. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 865–868.

[131] A. van Schaik and S. Shamma (2003): A neuromorphic sound localizer for a smart mems system. In *IEEE International Symposium on Circuits and Systems*. pp. 864–867.

[132] G. Cauwenberghs, R. Edwards, Y. Deng, R. Genov, and D. Lemonds (2002): Neuromorphic processor for real-time biosonar object detection. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. pp. 3984–3987.

[133] G. Crebbin and M. Fajria (2000): Integrate-and-fire models for image segmentation. In *Visual Communications and Image Processing 2000*, pp. 867–874.

[134] T. Netter and N. Franceschini (2002): A robotic aircraft that follows terrain using a neuromorphic eye. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2002)*, pp. 129–134.

[135] M. Lewis, M. Hartmann, R. Etienne-Cummings, and A. Cohen (2001): Control of a robot leg with an adaptive aVLSI CPG chip. *Neurocomputing. 38*, 1409–1421.

[136] T. Delbrck, S.-C. Liu, E. Chicca, G. M. Ricci, and S. Bovet. (2001): The physiologist's friend chip. [Online]. Available: http://www.ini.unizh.ch/tobi/friend/chip/index.html

[137] T. Berger, M. Baudry, R. Brinton, J. Liaw, V. Marmarelis, A. Park, B. Sheu, and A. Tanguay (2001): Brain-implantable biomimetic electronics as the next era in neural prosthetics. *Proceedings of the IEEE. 89*(7), 993–1012.

[138] T. Lande, J. Marienborg, and Y. Berg (2000): Neuromorphic cochlea implants. In *IEEE International Symposium on Circuits and Sys. (ISCAS 2000)*, pp. 401–404.

[139] E. Maynard (2001): Visual prostheses. *Annual Review of Biomedical Engineering. 3*, 145–168.

[140] R. Hahnloser, R. Sarpeshkar, M. Mahowald, R. Douglas, and H. Seung (2000): Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature. 405*, 947–951.

[141] T. DeMarse, D. Wagenaar, A. Blau, and S. Potter (2001): The neurally controlled animat: Biological brains acting with simulated bodies. *Autonomous Robots. 11*, 305–310.

[142] J. Anderson and E. Rosenfeld (eds) (1988): *Neurocomputing: Foundations of Research*. MIT Press, Cambridge.