# 18

# Hardware for Neural Networks

## 18.1 Taxonomy of neural hardware

This chapter concludes our analysis of neural network models with an overview of some hardware implementations proposed in recent years. In the first chapter we discussed how biological organisms process information. We are now interested in finding out how best to process information using electronic devices which in some way emulate the massive parallelism of the biological world. We show that neural networks are an attractive option for engineering applications if the *implicit* parallelism they offer can be made *explicit* with appropriate hardware. The important point in any parallel implementation of neural networks is to restrict communication to local data exchanges. The structure of some of those architectures, such as systolic arrays, resembles cellular automata.

There are two fundamentally different alternatives for the implementation of neural networks: a software simulation in conventional computers or a special hardware solution capable of dramatically decreasing execution time. A software simulation can be useful to develop and debug new algorithms, as well as to benchmark them using small networks. However, if large networks are to be used, a software simulation is not enough. The problem is the time required for the learning process, which can increase exponentially with the size of the network. Neural networks without learning, however, are rather uninteresting. If the weights of a network were fixed from the beginning and were not to change, neural networks could be implemented using any programming language in conventional computers. But the main objective of building special hardware is to provide a platform for efficient adaptive systems, capable of updating their parameters in the course of time. New hardware solutions are therefore necessary [54].

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

### 18.1.1 Performance requirements

Neural networks are being used for many applications in which they are more effective than conventional methods, or at least equally so. They have been introduced in the fields of computer vision, robot kinematics, pattern recognition, signal processing, speech recognition, data compression, statistical analysis, and function optimization. Yet the first and most relevant issue to be decided before we develop physical implementations is the size and computing power required for the networks.

The capacity of neural networks can be estimated by the number of weights used. Using this parameter, the complexity of the final implementation can be estimated more precisely than by the number of computing units. The number of weights also gives an indication of how difficult it will be to train the network. The performance of the implementation is measured in *connections per second* (cps), that is, by the number of data chunks transported through all edges of the network each second. Computing a connection requires the transfer of the data from one unit to another, multiplying the data by the edge's weight in the process. The performance of the learning algorithm is measured in *connection updates per second* (cups). Figure 18.1 shows the number of connections and connection updates per second required for some computationally intensive applications. The numbered dots represent the performance and capacity achieved by some computers when executing some of the reported neural networks applications.

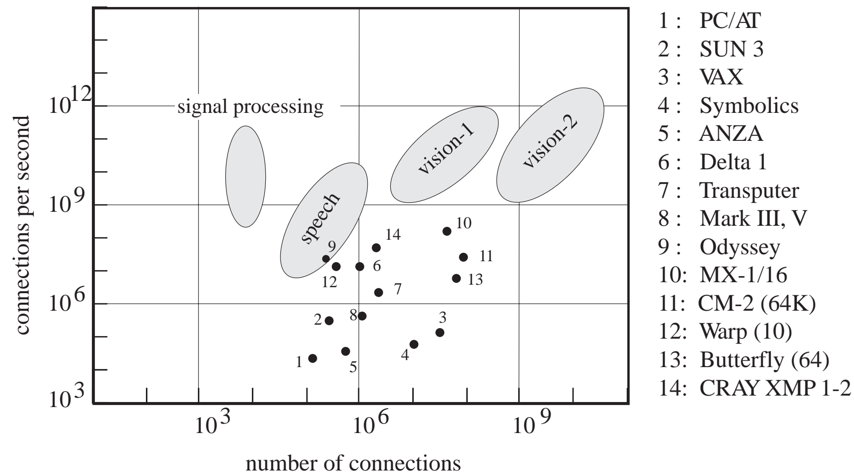



**Fig. 18.1.** Performance and capacity of different implementations of neural networks and performance requirements for some applications [Ramacher 1991]

Figure 18.1 shows that an old personal computer is capable of achieving up to 10Kcps, whereas a massively parallel CM-2 with 64K processors is

capable of providing up to 10 Mcps of computing power. With such performance we can deal with speech recognition in its simpler form, but not with complex computer vision problems or more sophisticated speech recognition models. The performance necessary for this kind of application is of the order of Giga-cps. Conventional computers cannot offer such computing power with an affordable price-performance ratio. Neural network applications in which a person interacts with a computer require compact solutions, and this has been the motivation behind the development of many special coprocessors. Some of them are listed in Figure 18.1, such as the Mark III or ANZA boards [409].

A pure software solution is therefore not viable, yet it remains open whether an analog or a digital solution should be preferred. The next sections deal with this issue and discuss some of the systems that have been built, both in the analog and the digital world.

### 18.1.2 Types of neurocomputers

To begin with, we can classify the kinds of hardware solution that have been proposed by adopting a simple taxonomy of hardware models. This will simplify the discussion and help to get a deeper insight into the properties of each device. Defining a taxonomy of neurocomputers requires consideration of three important factors:

- the kind of signals used in the network,
- the implementation of the weights,
- the integration and output functions of the units.

The signals transmitted through the network can be coded using an analog or a digital model. In the analog approach, a signal is represented by the magnitude of a current, or a voltage difference. In the digital approach, discrete values are stored and transmitted. If the signals are represented by currents or voltages, it is straightforward to implement the weights using resistances or transistors with a linear response function for a certain range of values. In the case of a digital implementation, each transmission through one of the network's edges requires a digital multiplication.

These two kinds of network implementations indicate that we must differentiate between analog and digital neurocomputers. Figure 18.2 shows this first level of classification and some successive refinements of the taxonomy. Hybrid neurocomputers are built combining analog and digital circuits.

The analog approach offers two further alternatives: the circuits can be built using electronic or optical components. The latter alternative has been studied and has led to some working prototypes but not to commercial products, where electronic neurocomputers still dominate the scene.

In the case of a digital implementation, the first two subdivisions in Figure 18.2 refer to conventional parallel or pipelined architectures. The networks can be distributed in multiprocessor systems, or it can be arranged for the

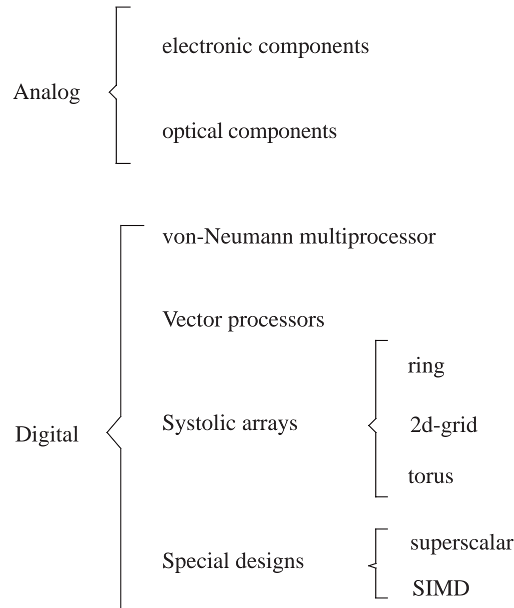



**Fig. 18.2.** Taxonomy of neurosystems

training set to be allocated so that each processor works with a fraction of the data. Neural networks can be efficiently implemented in vector processors, since most of the necessary operations are computations with matrices. Vector processors have been optimized for exactly such operations.

A third digital model is that of systolic arrays. They consist of regular arrays of processing units, which communicate only with their immediate neighbors. Input and output takes place at the boundaries of the array [207]. They were proposed to perform faster matrix-matrix multiplications, the kind of operation in which we are also interested for multilayered networks.

The fourth and last type of digital system consists of special chips of the superscalar type or systems containing many identical processors to be used in a SIMD (single instruction, multiple data) fashion. This means that all processors execute the same instruction but on different parts of the data.

Analog systems can offer a higher implementation density on silicon and require less power. But digital systems offer greater precision, programming flexibility, and the possibility of working with *virtual networks*, that is, networks which are not physically mapped to the hardware, making it possible to deal with more units. The hardware is loaded successively with different portions of the virtual network, or, if the networks are small, several networks can be processed simultaneously using some kind of multitasking.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

## 18.2 Analog neural networks

In the analog implementation of neural networks a coding method is used in which signals are represented by currents or voltages. This allows us to think of these systems as operating with real numbers during the neural network simulation.

### 18.2.1 Coding

When signals are represented by currents, it is easy to compute their addition. It is only necessary to make them meet at a common point. One of the Kirchhoff laws states that the sum of all currents meeting at a point is zero (outgoing currents are assigned a negative sign). The simple circuit of Figure 18.3 can be used to add $I_1$ and $I_2$, and the result is the current $I_3$.

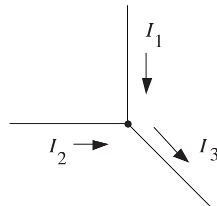



**Fig. 18.3.** Addition of electric current

The addition of voltage differences is somewhat more complicated. If two voltage differences $V_1$ and $V_2$ have to be added, the output line with voltage $V_1$ must be used as the reference line for voltage $V_2$. This simple principle cannot easily be implemented in real circuits using several different potentials $V_1, V_2, \ldots, V_n$. The representation of signals using currents is more advantageous in this case. The integration function of each unit can be implemented by connecting all incoming edges to a common point. The sum of the currents can be further processed by each unit.

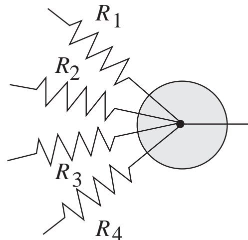



**Fig. 18.4.** Network of resistances

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

The weighting of the signal can be implemented using variable resistances. Rosenblatt used this approach in his first perceptron designs [185]. If the resistance is $R$ and the current $I$, the potential difference $V$ is given by Ohm's law $V = RI$. A network of resistances can simulate the necessary network connections and the resistors are the adaptive weights we need for learning (Figure 18.4).

Several analog designs for neural networks were developed in the 1970s and 1980s. Carver Mead's group at Caltech has studied different alternatives with which the size of the network and power consumption can be minimized [303]. In Mead's designs transistors play a privileged role, especially for the realization of so-called transconductance amplifiers. Some of the chips developed by Mead have become familiar names, like the *silicon retina* and the *silicon cochlea*.

Karl Steinbuch proposed at the end of the 1950s a model for associative learning which he called the *Lernmatrix* [414]. Figure 18.5 shows one of the columns of his learning matrix. It operated based on the principles discussed above.

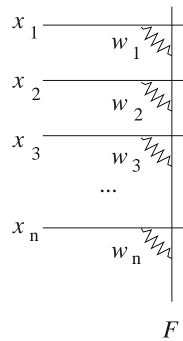



**Fig. 18.5.** A column of the *Lernmatrix*

The input $x_1, x_2, \ldots, x_n$ is transmitted as a voltage through each of the input lines. The resistances $w_1, w_2, \ldots, w_n$ transform the voltage into a weighted current. Several of these columns allow a vector-matrix product to be computed in parallel. Using such an array (and a special kind of nonlinearity) Steinbuch was able to build the first associative memory in hardware.

### 18.2.2 VLSI transistor circuits

Some VLSI circuits work with *field effect transistors* made of semiconductors. These are materials with a nonlinear voltage-current response curve. The nonlinear behavior makes them especially suitable for the implementation of digital switches.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

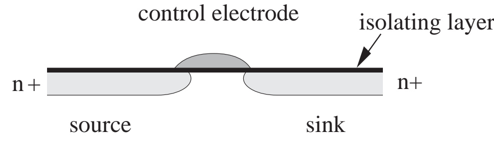



**Fig. 18.6.** Diagram of a field effect transistor (FET)

Figure 18.6 shows the structure of a typical field effect transistor. The source and the sink are made of n+ semiconductors, which contain a surplus of positive charge produced by carefully implanted impurities in the semiconductor crystal. A current can flow between source and sink only when the control electrode is positively charged above a certain threshold. Electrons are attracted from the source to the control electrode, but they also reach the sink by diffusing through the gap between source and sink. This closes the circuit and a current flows through the transistor. A potential difference is thus transformed into a current. The voltage difference is applied between source and control electrode, but the current that represents the signal flows between source and sink.

Let $V_g$ be the potential at the control electrode, $V_s$ the potential at the sink and $V_{gs}$ the potential difference between control electrode and source. Assume further that $V_{ds}$ is the potential difference between source and sink. It can be shown [303] that in this case the current $I$ through the transistor is given by

$$I = I_0 \, e^{cV_g - V_s}(1 - e^{V_{ds}}),$$

where $I_0$ and $c$ are constants. The approximation $I = I_0 \, e^{cV_g - V_s}$ is valid for large enough negative $V_{ds}$. In this case the output current depends only on $V_g$ and $V_s$.

It is possible to design a small circuit made of FETs capable of multiplying the numerical value of a voltage by the numerical value of a potential difference (Figure 18.7).

The three transistors $T_1$, $T_2$ and $T_b$ are interconnected in such a way that the currents flowing through all of them meet at a common point. This means that $I_1 + I_2 = I_b$. But since

$$I_1 = I_0 e^{cV_1 - V} \qquad \text{and} \qquad I_2 = I_0 e^{cV_2 - V},$$

it is true that

$$I_b = I_1 + I_2 = I_0 e^{-V}(e^{cV_1} + e^{cV2}).$$

Some additional algebraic steps lead to the following equation

$$I_1 - I_2 = I_b \, \frac{e^{cV_1} - e^{cV_2}}{e^{cV_1} + e^{cV_2}} = I_b \tanh \frac{c(V_1 - V_2)}{2}.$$

This equation shows that a potential difference $V_1 - V_2$ produces a nonlinear result $I_1 - I_2$. Using the circuit in Figure 18.7 we can implement the output

**Fig. 18.7.** Transconductance amplifier (Mead 1989)

function of a unit. Remember that the hyperbolic tangent is just the symmetric sigmoid used in backpropagation networks. For small values of $x$ it holds that $\tanh(x) \approx x$. In this case for small values of $V_1 - V_2$ we get

$$I_1 - I_2 = \frac{cI_b}{2}(V_1 - V_2).$$

The circuit operates as a multiplier. The magnitude of the current $I_b$ is multiplied by the magnitude of the potential difference $V_1 - V_2$, where we only require $I_b > 0$. An analog multiplier for two numbers with arbitrary sign requires more components and a more sophisticated design, but the basic construction principle is the same. An example is the Gilbert multiplier used in many VLSI circuits.

The above discussion illustrates how FETs can be used, and are in fact being used, to implement the main computations for hardware models of neural networks, that is, the weighting of signals, their integration, and the computation of the nonlinear output.

### 18.2.3 Transistors with stored charge

A possible alternative to the kind of components just discussed is to use *floating gate transistors* for an analog implementation of the multiplication operation. A problem with the circuits discussed above is that the numbers to be multiplied must be encoded as voltages and currents. The numerical value of the weights must be transformed into a potential difference, which must be kept constant as long as no weight update is performed. It would be simpler

to store the weights using an analog method in such a way that they could be used when needed. This is exactly what floating gate transistors can achieve.

Figure 18.8 shows the scheme of such a transistor. The structure resembles that of a conventional field effect transistor. The main visible difference is the presence of a metallic layer between the control electrode and the silicon base of the transistor. This metallic layer is isolated from the rest of the transistor in such a way that it can store a charge just as a capacitor. The metallic layer can be charged by raising the potential difference between the control electrode and the source by the right amount. The charge stored in the metallic layer decays only slowly and can be stored for a relatively long time [185].

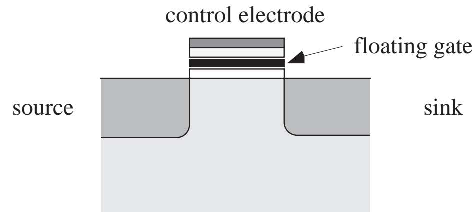



**Fig. 18.8.** Diagram of a floating gate transistor

A floating gate transistor can be considered as one which includes an analog memory. The stored charge modifies the effectiveness of the control electrode. The current which flows through source and sink depends linearly (in a certain interval) on the potential difference between them. The proportionality constant is determined by the stored charge. The potential difference is therefore weighted before being transformed into a current. Learning is very simple to implement: weight updates correspond to changes in the amount of charge stored. Even if the power source is disconnected the magnitudes of the weights remain unchanged.

### 18.2.4 CCD components

Floating gate transistors represent the weights by statically stored charges. A different approach consists in storing charges *dynamically*, as is done with the help of *charge coupled devices* (CCDs). Computation of the scalar product of a weight and an input vector can be arranged in such a way that the respective components are used sequentially. The weights, which are stored dynamically, can be kept moving in a closed loop. At a certain point in the loop, the magnitudes of the weights are read and multiplied by the input. The accumulated sum is the scalar product needed.

Figure 18.9 shows a CCD circuit capable of transporting charges. A chain of electrodes is built on top of a semiconductor base. Each electrode receives the signal of a clock. The clock pulses are synchronized in such a way that at

every discrete time $t$ just one of the electrodes reaches its maximum potential. At the next time step $t+1$ only the neighbor to the right reaches the maximum potential and so on. A charge $Q$ is stored in the semiconductor material, which has been treated in order to create on its surface potential wells aligned with the electrodes. The charge stored in a potential well diffuses gradually (as shown for $t = 1.5$), but if the next electrode reaches its maximum before the charge has completely diffused, it is again collected as a charge packet in its potential well. One or more stored charge packets can move from left to right in this CCD circuit. If the chain of electrodes is arranged in a closed loop, the stored charge can circulate indefinitely.

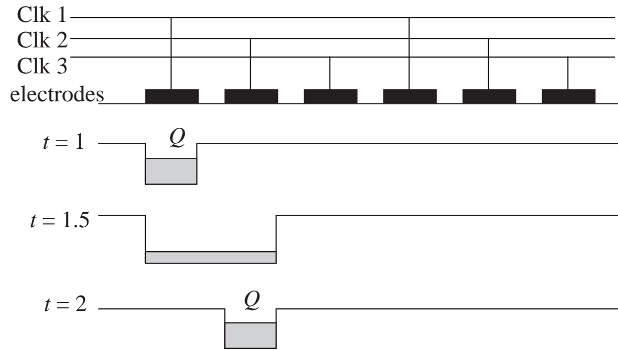



**Fig. 18.9.** CCD transport band

Figure 18.10 shows a very interesting design for an associative memory with discrete weights. The matrix of weights is stored in the $n$ CCD linear transport arrays shown on the left. Each row of the weight matrix is stored in one of the arrays. At the end of the line a weight is read in every cycle and is transformed into a digital number using an A/D converter. The transport chain to the right stores the momentary states of the units (0 or 1). In each cycle a state is read and it is multiplied by each of the $n$ weights from the CCD arrays to the left. Since the states and weights are single bits, an AND gate is enough. The partial products are kept in the accumulators for $n$ cycles. At the end of this period of time the circuit has computed $n$ scalar products which are stored in the $n$ accumulators. A threshold function now converts the scalar products in the new unit states, which are then stored on the vertical CCD array.

Very compact arithmetic and logical units can be built using CCD technology. The circuit shown in Figure 18.10 is actually an example of a hybrid approach, since the information is handled using analog and digital coding. Although much of the computation is done serially, CCD components are fast enough. The time lost in the sequential computation is traded off against the

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

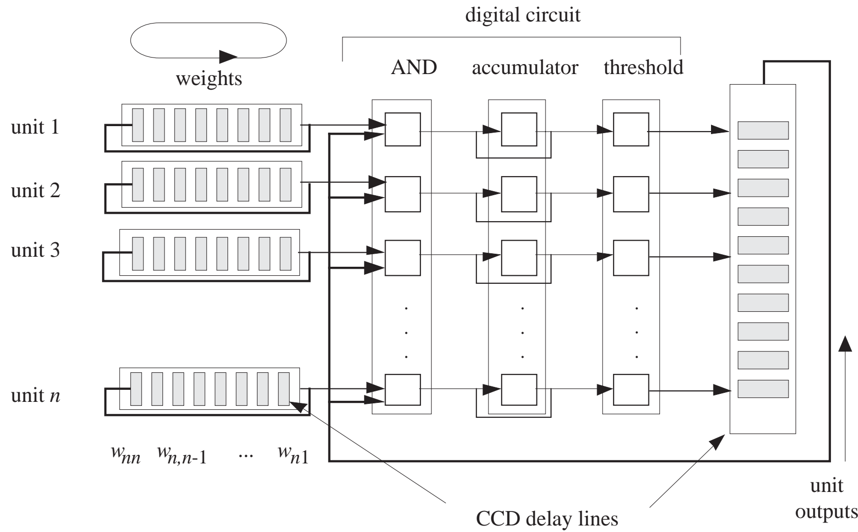



**Fig. 18.10.** Associative memory in CCD technology

simplicity of the design. The circuit works synchronously, because information is transported at discrete points in time.

## 18.3 Digital networks

In digital neurocomputers signals and network parameters are encoded and processed digitally. The circuits can be made to work with arbitrary precision just by increasing the word length of the machine. Analog neurocomputers are affected by the strong variation of the electrical characteristics of the transistors, even when they have been etched on the same chip [303]. According to the kind of analog component, the arithmetic precision is typically limited to 8 or 9 bits [185]. Analog designs can be used in all applications that can tolerate statistical deviations in the computation and that do not require more than the precision just mentioned, as is the case, for example, in computer vision. If we want to work with learning algorithms based on gradient descent, more precision is required.

There is another reason for the popularity of digital solutions in the neurocomputing field: our present computers work digitally and in many cases the neural network is just one of the pieces of a whole application. Having everything in the same computer makes the integration of the software easier. This has led to the development of several boards and small machines connected to a host as a neural coprocessor.

### 18.3.1 Numerical representation of weights and signals

If a neural network is simulated in a digital computer, the first issue to be decided is how many bits should be used for storage of the weights and signals. Pure software implementations use some kind of floating-point representation, such as the IEEE format, which provides very good numerical resolution at the cost of a high investment in hardware (for example for the 64-bit representations). However, floating-point operations require more cycles to be computed than their integer counterparts (unless very complex designs are used). This has led most neurocomputer designers to consider fixed-point representations in which only integers are used and the position of the decimal point is managed by the software or simple additional circuits. If such a representation is used, the appropriate word length must be found. It must not affect the convergence of the learning algorithms and must provide enough resolution during normal operation. The classification capabilities of the trained networks depend on the length of the bit representation [223].

Some researchers have conducted series of experiments to find the appropriate word length and have found that 16 bits are needed to represent the weights and 8 to represent the signals. This choice does not affect the convergence of the backpropagation algorithm [31, 197]. Based on these results, the design group at Oregon decided to build the CNAPS neurocomputer [175] using word lengths of 8 and 16 bits (see Sect. 8.2.3).

Experience with some of the neurocomputers commercially available shows that the 8- and 16-bit representations provide enough resolution in most, but not in all cases. Furthermore, some complex learning algorithms, such as variants of the conjugate gradient methods, require high accuracy for some of the intermediate steps and cannot be implemented using just 16 bits [341]. The solution found by the commercial suppliers of neurocomputers should thus be interpreted as a compromise, not as the universal solution for the encoding problem in the field of neurocomputing.

### 18.3.2 Vector and signal processors

Almost all models of neural networks discussed in the previous chapters require computation of the scalar product of a weight and an input vector. Vector and signal processors have been built with this type of application in mind and are therefore also applicable for neurocomputing.

A vector processor, such as a CRAY, contains not only scalar registers but also vector registers, in which complete vectors can be stored. To perform operations on them, they are read sequentially from the vector registers and their components are fed one after the other to the arithmetic units. It is characteristic for vector processors that the arithmetic units consist of several stages connected in a pipeline. The multiplier in Figure 18.11, for example, performs a single multiplication in 10 cycles. If the operand pairs are fed sequentially into the multiplier, one pair in each cycle, at the end of cycle 10

one result has been computed, another at the end of cycle 11, etc. After a start time of 9 cycles a result is produced in every cycle. The partial results can be accumulated with an adder. After $9 + n$ cycles the scalar product has been computed.

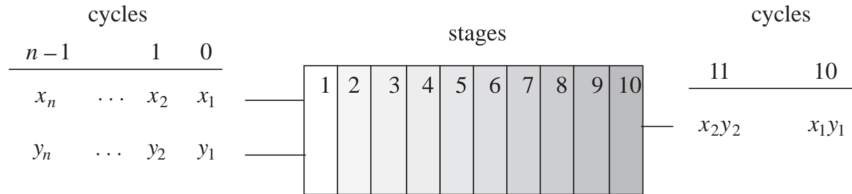



**Fig. 18.11.** Multiplier with 10 pipeline sections

The principle of vector processors has been adopted in signal processors, which always include fast pipelined multipliers and adders. A multiplication can be computed simultaneously with an addition. They differ from vector processors in that no vector registers are available and the vectors must be transmitted from external memory modules. The bottleneck is the time needed for each memory access, since processors have become extremely fast compared to RAM chips [187]. Some commercial neurocomputers are based on fast signal processors coupled to fast memory components.

### 18.3.3 Systolic arrays

An even greater speedup of the linear algebraic operations can be achieved with *systolic arrays*. These are regular structures of VLSI units, mainly one- or two-dimensional, which can communicate only locally. Information is fed at the boundaries of the array and is transported synchronously from one stage to the next. Kung gave these structures the name systolic arrays because of their similarity to the blood flow in the human heart. Systolic arrays can compute the vector-matrix multiplication using fewer cycles than a vector processor. The product of an $n$-dimensional vector and an $n \times n$ matrix can be computed in $2n$ cycles. A vector processor would require in the order of $n^2$ cycles for the same computation.

Figure 18.12 shows an example of a two-dimensional systolic array capable of computing a vector-matrix product. The rows of the matrix

$$\mathbf{W} = \begin{pmatrix} c & d \\ e & f \end{pmatrix}$$

are used as the input from the top into the array. In the first cycle $c$ and $d$ constitute the input. In the second cycle $e$ and $f$ are fed into the array, but displaced by one unit to the right. We want to multiply the vector $(a, b)$

with the matrix $\mathbf{W}$. The vector $(a, b)$ is fed into the array from the left. Each unit in the systolic array multiplies the data received from the left and from above and the result is added to the data transmitted through the diagonal link from the neighbor located to the upper left. Figure 18.12 shows which data is transmitted through which links (for different cycles). After two cycles, the result of the vector-matrix multiplication is stored in the two lower units to the right. Exactly the same approach is used in larger systolic arrays to multiply $n$-dimensional vectors and $n \times n$ matrices.

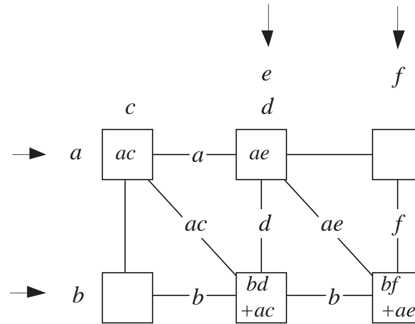



**Fig. 18.12.** Planar systolic array for vector-matrix multiplication

The systolic array used in the above example can be improved by connecting the borders of the array to form a torus. The previous vector-matrix multiplication is computed in this case, as shown in Figure 18.13. In general it holds that in order to multiply an $n$-dimensional vector with an $n \times n$ matrix, a systolic array with $n(2n-1)$ elements is needed. A torus with $n \times n$ elements can perform the same computation. Therefore when systolic arrays are used for neurocomputers, a toroidal architecture is frequently selected.

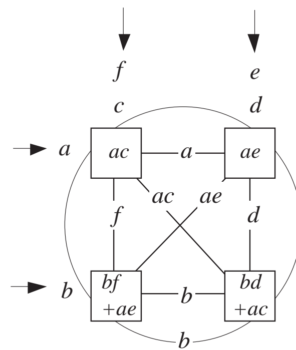



**Fig. 18.13.** Systolic array with toroidal topology

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

An interesting systolic design is the one developed by Siemens (Figure 18.14). It consists of a two-dimensional array of multipliers. The links for the transmission of weights and data are 16 bits wide. The array shown in the figure can compute the product of four weights and four inputs. Each multiplier gets two arguments which are then multiplied. The result is transmitted to an adder further below in the chain. The last adder eventually contains the result of the product of the four-dimensional input with the four-dimensional weight vector. The hardware implementation does not exactly correspond to our diagram, since four links for the inputs and four links for the weights are not provided. Just one link is available for the inputs and one for the weights, but both links operate by multiplexing the input data and the weights, that is, they are transmitted sequentially in four time frames. Additional hardware in the chip stores each weight as needed for the multiplication and the same is done for the inputs. In this way the number of pins in the final chip could be dramatically reduced without affecting performance [354]. This kind of multiplexed architecture is relatively common in neurocomputers [38].

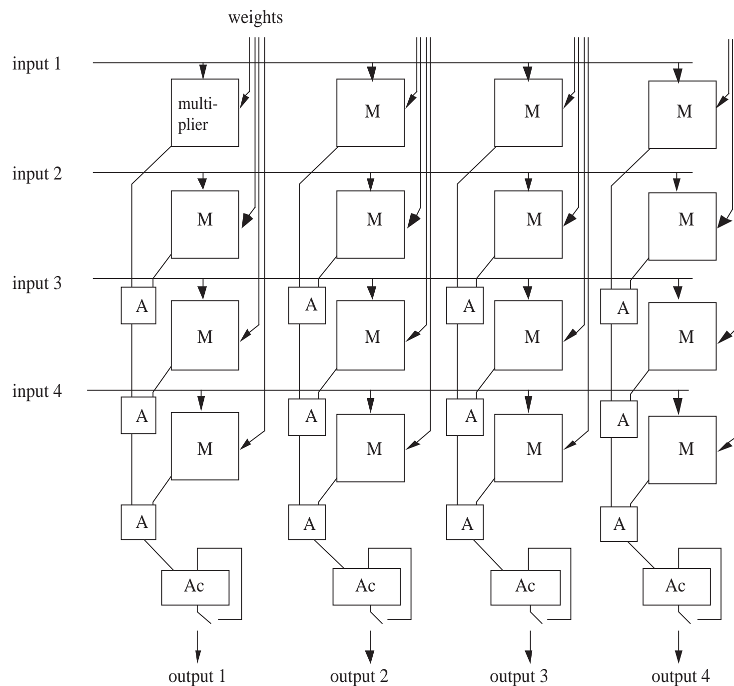



**Fig. 18.14.** Systolic array for 4 scalar product chains

The systolic chips manufactured by Siemens (MA16) can be arranged in even larger two-dimensional arrays. Each MA16 chip is provided with an external memory in order to store some of the network's weights. The performance

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

promised by the creators of the MA16 should reach 128 Gcps for a $16 \times 16$ systolic array [354]. This kind of performance could be used, for example, in computer vision tasks. Siemens started marketing prototypes of the *Synapse* neurocomputer in 1995.

### 18.3.4 One-dimensional structures

The hardware invested in two-dimensional systolic arrays is in many cases excessive for the vector-matrix multiplication. This is why some researchers have proposed using one-dimensional systolic arrays in order to reduce the complexity of the hardware without losing too much performance [438, 263].

In a systolic ring information is transmitted from processor to processor in a closed loop. Each processor in the ring simulates one of the units of the neural network. Figure 18.15 shows a ring with four processors. Assume that we want to compute the product of the $4 \times 4$ weight matrix $\mathbf{W} = \{w_{ij}\}$ with the vector $x^{\mathrm{T}} = (x_1, x_2, \ldots, x_n)$, that is, the vector $\mathbf{Wx}$. The vector $\mathbf{x}$ is loaded in the ring as shown in the figure. In each cycle each processor accesses a weight from its local memory (in the order shown in Figure 18.15) and gets a number from its left neighbor. The weight is multiplied by $x_i$ and the product is added to the number received from the left neighbor. The number from the left neighbor is the result of the previous multiply-accumulate operation (the ring is initialized with zeros). The reader can readily verify that after four cycles (a complete loop) the four small processors contain the four components of the vector $\mathbf{Wx}$.

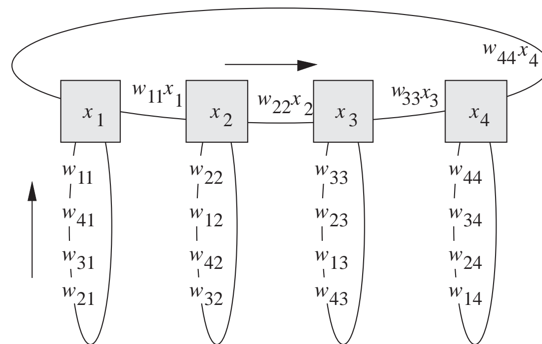



**Fig. 18.15.** Systolic array computing the product of $\mathbf{W}$ and $\mathbf{x}$

It is not necessary for the units in the ring to be fully-fledged processors. They can be arithmetic units capable of multiplying and adding and which contain some extra registers. Such a ring can be expanded to work with larger matrices just by including additional nodes.

For the backpropagation algorithm it is also necessary to multiply vectors with the transpose of the weight matrix. This can be implemented in the

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

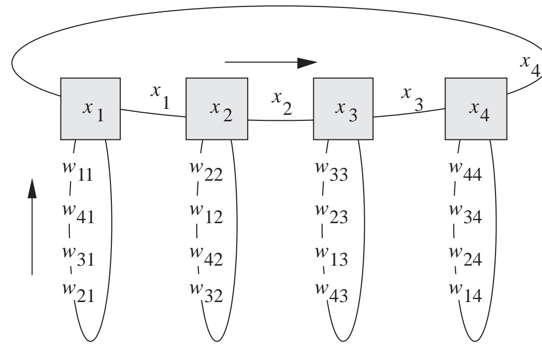



**Fig. 18.16.** Systolic array computing the product of the transpose **W** and **x**

same ring by making some changes. In each cycle the number transmitted to the right is $x_i$ and the results of the multiplication are accumulated at each processor. After four cycles the components of the product $\mathbf{W}^{\mathrm{T}}\mathbf{x}$ are stored at each of the four units.

A similar architecture was used in the Warp machine designed by Pomerleau and others at Princeton [347]. The performance reported was 17 Mcps. Jones et al. have studied similar systolic arrays [226].

Another machine of a systolic type is the RAP (Ring Array Processor) developed at the International Computer Science Institute in Berkeley. The processing units consist of signal processors with a large local memory [321]. The weights of the network are stored at the nodes in the local memories. The performance reported is 200–570 Mcps using 10 processor nodes in a ring. It is significant that the RAP has been in constant use for the last five years at the time of this writing. It seems to be the neural computer with the largest accumulated running time.

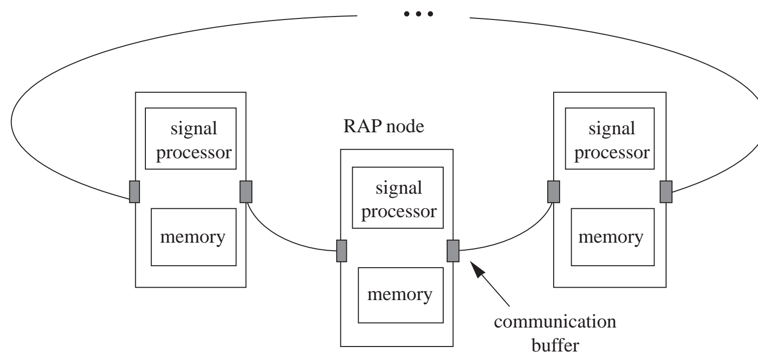



**Fig. 18.17.** Architecture of the Ring Array Processor

Figure 18.17 shows the architecture of the RAP. The local memory of each node is used to store a row of the weight matrix $\mathbf{W}$, so that each processor can compute the scalar product of a row with the vector $(x_1, x_2, \ldots, x_n)^{\mathrm{T}}$. The input vector is transmitted from node to node. To compute the product $\mathbf{W}^{\mathrm{T}}\mathbf{x}$ partial products are transmitted from node to node in systolic fashion. The weight updates can be made locally at each node. If the number of processors is lower than the dimension of the vector and matrix rows, then each processor stores two or more rows of the weight matrix and computes as much as needed. The advantage of this architecture compared to a pure systolic array is that it can be implemented using off-the-shelf components. The computing nodes of the RAP can also be used to implement many other non-neural operations needed in most applications.

## 18.4 Innovative computer architectures

The experience gained from implementations of neural networks in conventional von Neumann machines has led to the development of complete microprocessors especially designed for neural networks applications. In some cases the SIMD computing model has been adopted, since it is relatively easy to accelerate vector-matrix multiplications significantly with a low hardware investment. Many authors have studied optimal mappings of neural network models onto SIMD machines [275].

### 18.4.1 VLSI microprocessors for neural networks

In SIMD machines the same instruction is executed by many processors using different data. Figure 18.18 shows a SIMD implementation of a matrix-vector product. A chain of $n$ multipliers is coupled to $n$ adders. The components of a vector $x_1, x_2, \ldots, x_n$ are transmitted sequentially to all the processors. At each cycle one of the columns of the matrix $\mathbf{W}$ is transmitted to the processors. The results of the $n$ multiplications are accumulated in the adders and at the same time the multipliers begin to compute the next products. For such a computation model to work efficiently it is necessary to transmit $n$ arguments from the memory to the arithmetic units at each cycle. This requires a very wide instruction format and a wide bus.

The Torrent chip completed and tested at Berkeley in the course of 1995 has a SIMD architecture in each of its vector pipelines [440] and was the first vector processor on a single chip. The chip was designed for neural network but also for other digital signal processing tasks. It consists of a scalar unit compatible with a MIPS-II 32-bit integer CPU. A fixed point coprocessor is also provided.

Figure 18.19 shows a block diagram of the chip. It contains a register file for 16 vectors, each capable of holding 32 elements, each of 32 bits. The pipeline of the chip is superscalar. An instruction can be assigned to the CPU, the

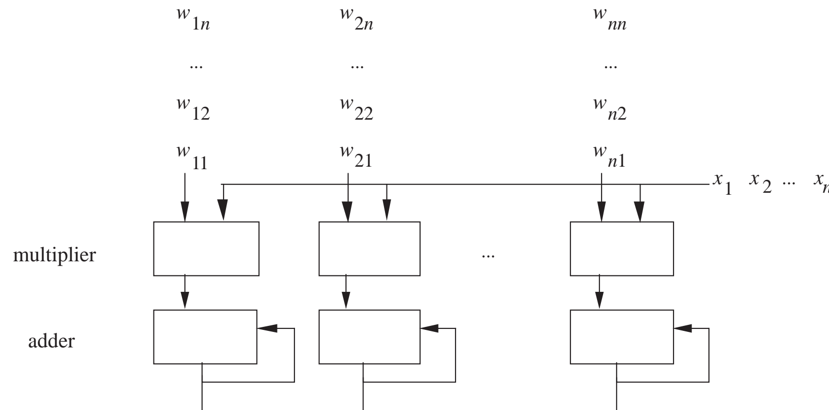



**Fig. 18.18.** SIMD model for vector-matrix multiplication

Vector memory unit (VMP) or any of two vector processing units (VP0 and VP1). The vector units can add, shift, and do conditional moves. Only VP0 can multiply. The multipliers in VP0 perform $16 \times 16 \rightarrow 32$ bit multiplications. The vector memory operations provided allow efficient loading and storage of vectors.

Interesting is that all three vector functional units are composed of 8 parallel pipelines. This means that the vector is striped along the 8 pipelines. When all three vector units are saturated, up to 24 operations per cycle are being executed.

Due to its compatibility with the MIPS series, the scalar software can be compiled with the standard available tools. The reported performance with 45 MHz is 720 Mflops [33].

In the Torrent the weight matrix is stored outside the VLSI chip. In the CNAPS (Connected Network of Adaptive Processors) built and marketed by Adaptive Solutions, the weights are stored in local memories located in the main chip [175]. The CNAPS also processes data in SIMD fashion. Each CNAPS chip contains 64 processor nodes together with their local memory.

Figure 18.20 shows the basic structure of the CNAPS chips, which have been built using Ultra-VLSI technology: more than 11 million transistors have been etched on each chip. Every one of them contains 80 processor nodes, but after testing only 64 are retained to be connected in a chain [176]. The 64 nodes receive the same input through two buses: the instruction bus, necessary to coordinate the 64 processing units, and the input bus. This has a width of only 8 bits and transmits the outputs of the simulated units to the other units. The weights are stored locally at each node. The programmer can define any computation strategy he or she likes, but usually what works best is to store all incoming weights in a unit in the local memory, as well as all outgoing weights (necessary for the backpropagation algorithm). The output bus, finally, communicates the output of each unit to the other units.

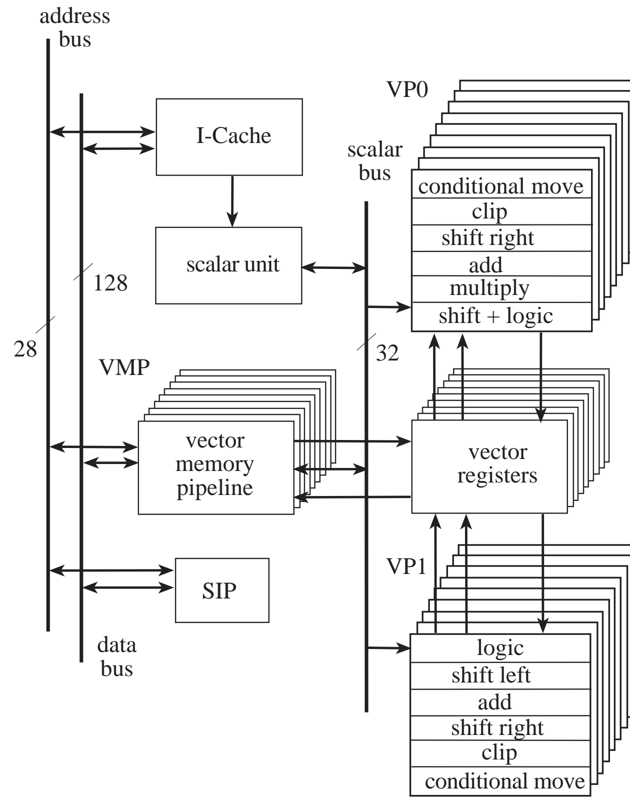



**Fig. 18.19.** Torrent block diagram [Asanovic et al. 95]

Figure 18.21 shows a block diagram of a processor node. A multiplier, an adder, and additional logical functions are provided at each node. The register file consists of 32 16-bit registers. The local memory has a capacity of 4Kb. The structure of each node resembles a RISC processor with additional storage. Using one of the chips with 64 processing nodes it is possible to multiply an 8-bit input with 64 16-bit numbers in parallel. Several CNAPS chips can be connected in a linear array, so that the degree of parallelism can be increased in quanta of 64 units. As in any other SIMD processor it is possible to test status flags and switch-off some of the processors in the chain according to the result of the test. In this way an instruction can be executed by a subset of the processors in the machine. This increases the flexibility of the system. A coprocessor with 4 CNAPS chips is capable of achieving 5 Gcps and 944 Mcups.

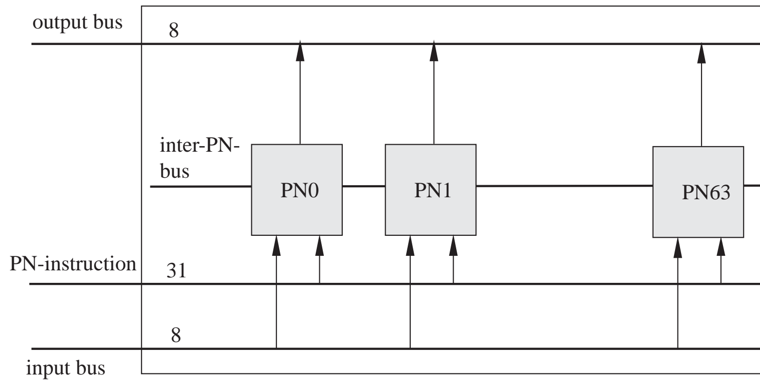



**Fig. 18.20.** CNAPS-Chip with 64 processor nodes (PN)

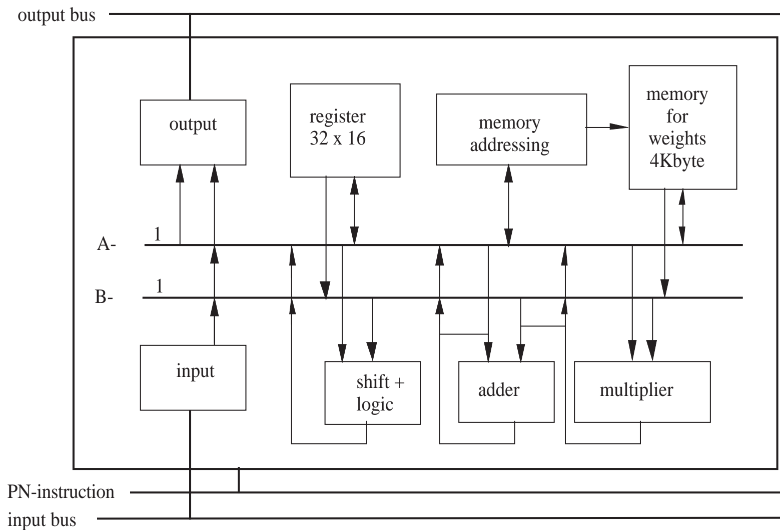



**Fig. 18.21.** Structure of a processor node

### 18.4.2 Optical computers

The main technical difficulty that has to be surmounted in neural networks, and other kinds of massively parallel systems, is the implementation of communication channels. A Hopfield network with 1000 units contains half a million connections. With these large topologies we have no alternative but to work with virtual networks which are partitioned to fit on the hardware at hand. This is the approach followed for the digital implementations discussed in the last section. Optical computers have the great advantage, compared to electronic machines, that the communication channels do not need to be hard-wired. Signals can be transmitted as light waves from one component to

the other. Also, light rays can cross each other and this does not affect the information they are carrying. The energy needed to transmit signals is low, since there is no need to consider the capacity of the transmitting medium, as in the case of metal cables. Switching times of up to 30 GHz can be achieved with optical elements [296].

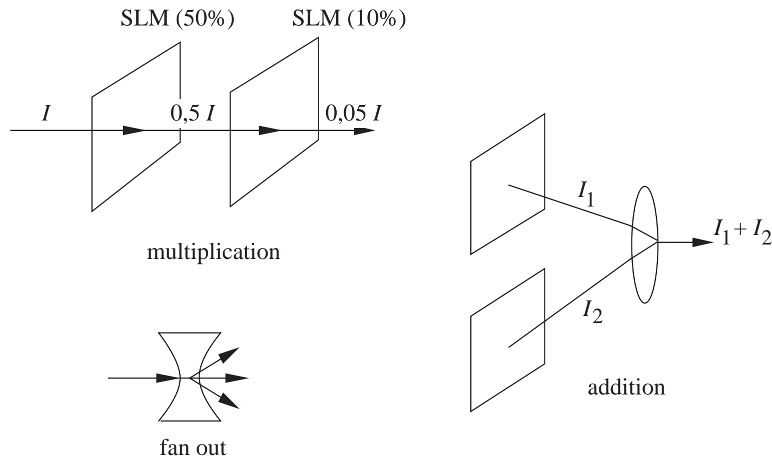



**Fig. 18.22.** Optical implementation of multiplication, addition, and signal splitting

Using optical signals it is possible to implement basic logic operations in a straightforward way. To do this, SLMs (Spatial Light Modulators) are used. These are optical masks which can be controlled using electrodes. According to the voltage applied the SLM becomes darker and reduces the amount of transmitted light when a light ray is projected onto the mask. A light signal projected onto an SLM configured to let only 50% of the light through loses half its intensity. This can be interpreted as multiplication of the number 1 by the number 0.5. A chain of multiplications can be computed almost "instantly" by letting the light ray go through several SLMs arranged in a stack. The light coming out has an intensity proportional to the product of the darkness ratios of the SLMs.

Addition of optical signals can be implemented by reducing two light signals to a single one. This can be done using lenses, prisms, or any other devices capable of dealing with light rays. A signal can be split in two or more using crystals or certain lens types. Figure 18.22 shows possible realizations of some operations necessary to implement optical computers.

Using these techniques all linear algebraic operations can be computed in parallel. To perform a vector-matrix or a matrix-vector multiplication it is only necessary to use an SLM that has been further subdivided into $n \times n$ fields. The darkness of each field must be adjustable, although in some applications it is

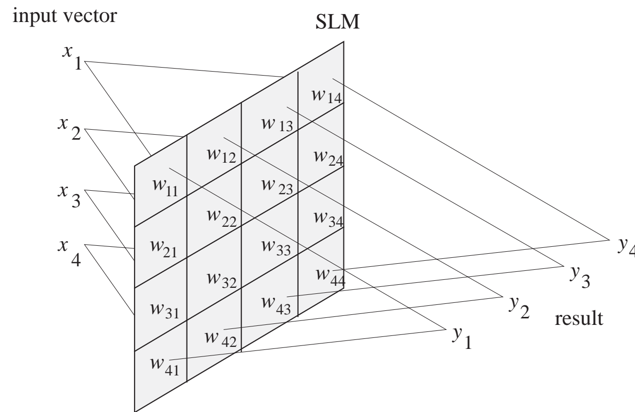



**Fig. 18.23.** Matrix-vector multiplication with an SLM mask

possible to use constant SLM masks. Each field represents one of the elements of the weight matrix and its darkness is adjusted according to its numerical value. The vector to be multiplied with the weight matrix is projected onto the SLM in such a way that the signal $x_1$ is projected onto the first row of the SLM matrix, $x_2$ onto the second row and so on. The outcoming light is collected column by column. The results are the components of the product we wanted to compute.

**Fig. 18.24.** Optical implementation of the backpropagation algorithm

Many other important operations can be implemented very easily using optical computers [134]. Using special lenses, for example, it is possible to instantly compute the Fourier transform of an image. Some pattern recognition problems can be solved more easily taking the image from the spatial to the frequency domain (see Chap. 12). Another example is that of feedback

systems of the kind used for associative memories. Cellular automata can be also implemented using optical technology. Figure 18.24 shows a diagram of an experiment in which the backpropagation algorithm was implemented with optical components [296].

The diagram makes the main problem of today's optical computers evident: they are still too bulky and are mainly laboratory prototypes still waiting to be miniaturized. This could happen if new VLSI techniques were developed and new materials discovered that could be used to combine optical with electronic elements on the same chips. However, it must be kept in mind that just 40 years ago the first transistor circuits were as bulky as today's optical devices.

### 18.4.3 Pulse coded networks

All of the networks considered above work by transmitting signals encoded analogically or digitally. Another approach is to implement a closer simulation of the biological model by transmitting signals as discrete pulses, as if they were action potentials. Biological systems convey information from one neuron to another by varying the firing rate, that is, by something similar to frequency modulation. A strong signal is represented by pulses produced with a high frequency. A feeble signal is represented by pulses fired with a much lower frequency. It is not difficult to implement such pulse coding systems in analog or digital technology.

Tomlinson et al. developed a system which works with discrete pulses [430]. The neurochip they built makes use of an efficient method of representing weights and signals. Figure 18.25 shows an example of the coding of two signals using asynchronous pulses.

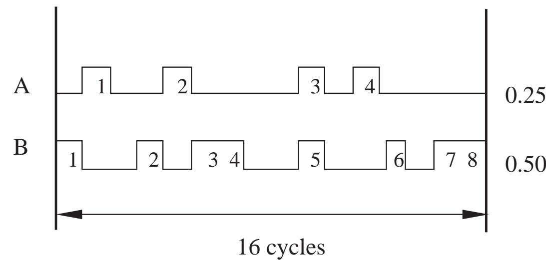



**Fig. 18.25.** Pulse coded representation of signals

Assume that two signals $A$ and $B$ have the respective numerical values 0.25 and 0.5. All signals are represented by pulses in a 16-cycles interval. A generator produces square pulses randomly, but in such a way that for signal $A$, a pulse appears in the transmitted stream a quarter of the time. In the case of $B$, a pulse is produced half of the time. The pulse trains are uncorrelated

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

to each other. This is necessary in order to implement the rest of the logic functions. A decoder can reconstruct the original numbers just by counting the number of pulses in the 16-cycle interval.

The product of the two numbers $A$ and $B$ can be computed using a single AND gate. As shown in Figure 18.26, the two pulse streams are used as the input to the AND gate. The result is a pulse chain containing only $0.25 \times 0.5 \times 16$ pulses. This corresponds to the number $0.25 \times 0.5$, that is, the product of the two arguments $A$ and $B$. This is of course only a statistical result, since the number of pulses can differ from the expected average, but the accuracy of the computation can be increased arbitrarily by extending the length of the coding interval. Tomlinson found in his experiments that 256 cycles were good enough for most of the applications they considered. This corresponds, more or less, to a signal resolution of 8 bits.

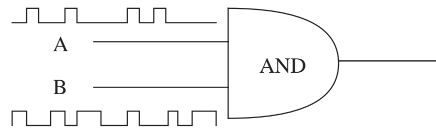



**Fig. 18.26.** Multiplication of two pulse coded numbers

The integration and nonlinear output function of a unit can be computed using an OR gate. If two numbers $A$ and $B$ are coded as indicated and are used as arguments, the result is a train of pulses which corresponds to the number $C = 1 - (1 - A)(1 - B)$. This means that we always get a one as the result, except in the case where both pulse trains contain zeroes. This happens with probability $(1 - A)(1 - B)$. The result $C$ corresponds to a kind of summation with an upper saturation bound which restricts the output to the interval $[0, 1]$. Ten pulse-coded numbers $A_1, A_2, \ldots, A_{10}$ can be integrated using an OR gate. For small values of $A_i$ the result is

$$C = 1 - (1 - A_1) \cdots (1 - A_{10}) \approx 1 - (1 - \sum_{i=1}^{10} A_i) \approx \sum_{i=1}^{10} A_i .$$

It can be shown that for larger magnitudes of the signal and a wide enough coding interval, the approximation

$$C = 1 - \exp(-\sum_{i=1}^{10} A_i)$$

holds. This function has the shape of a squashing function similar to the sigmoid.

The kind of coding used does not allow us to combine negative and positive signals. They must be treated separately. Only when the activation of a unit

has to be computed do we need to combine both types of signals. This requires additional hardware, a problem which also arises in other architectures, for example in optical computers.

How to implement the classical learning algorithms or their variants using pulse coding elements has been intensively studied . Other authors have built analog systems which implement an even closer approximation to the biological model, as done for example in [49].

## 18.5 Historical and bibliographical remarks

The first attempts to build special hardware for artificial neural networks go back to the 1950s in the USA. Marvin Minsky built a system in 1951 that simulated adaptive weights using potentiometers [309]. The perceptron machines built by Rosenblatt are better known. He built them from 1957 to 1958 using Minsky's approach of representing weights by resistances in an electric network.

Rosenblatt's machines could solve simple pattern recognition tasks. They were also the first commercial neurocomputers, as we call such special hardware today. Bernard Widrow and Marcian Hoff developed the first series of adaptive systems specialized for signal processing in 1960 [450]. They used a special kind of vacuum tube which they called a *memistor*. The European pioneers were represented by Karl Steinbuch, who built associative memories using resistance networks [415].

In the 1970s there were no especially important hardware developments for neural networks, but some attempts were made in Japan to actually build Fukushima's cognitron and neocognitron [144, 145].

Much effort was invested in the 1980s to adapt conventional multiprocessor systems to the necessities of neural networks. Hecht-Nielsen built the series of Mark machines, first using conventional microprocessors and later by developing special chips. Some other researchers have done a lot of work simulating neural networks in vector computers or massively parallel systems.

The two main fields of hardware development were clearly defined in the middle of the 1980s. In the analog world the designs of Carver Mead and his group set the stage for further developments. In the digital world many alternatives were blooming at this time. Zurada gives a more extensive description of the different hardware implementations of neural networks [469].

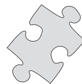
Systolic arrays were developed by H. T. Kung at Carnegie Mellon in the 1970s [263]. The first systolic architecture for neural networks was the Warp machine built at Princeton. The RAP and Synapse machines, which are not purely systolic designs, nevertheless took their inspiration from the systolic model.

But we are still awaiting the greatest breakthrough of all: when will optical computers become a reality? This is the classical case of the application, the

neural networks, waiting for the machine that can transform all their promises into reality.

## Exercises

1. Train a neural network using floating-point numbers with a limited precision for the mantissa, for example 12 bits. This can be implemented easily by truncating the results of arithmetic operations. Does backpropagation converge? What about other fast variations of backpropagation?
2. Show how to multiply two $n \times n$ matrices using a two-dimensional systolic array. How many cycles are needed? How many multipliers?
3. Write the pseudocode for the backpropagation algorithm for the CNAPS. Assume that each processor node is used to compute the output of a single unit. The weights are stored in the local memory of the PNs.
4. Propose an optical system of the type shown in Figure 18.23, capable of multiplying a vector with a matrix $\mathbf{W}$, represented by an SLM, and also with its transpose.



R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996