2

Threshold Logic

2.1 Networks of functions

We deal in this chapter with the simplest kind of computing units used to build artificial neural networks. These computing elements are a generalization of the common logic gates used in conventional computing and, since they operate by comparing their total input with a threshold, this field of research is known as *threshold logic*.

2.1.1 Feed-forward and recurrent networks

Our review in the previous chapter of the characteristics and structure of biological neural networks provides us with the initial motivation for a deeper inquiry into the properties of networks of abstract neurons. From the viewpoint of the engineer, it is important to define how a network should behave, without having to specify completely all of its parameters, which are to be found in a learning process. Artificial neural networks are used in many cases as a *black box*: a certain input should produce a desired output, but how the network achieves this result is left to a self-organizing process.

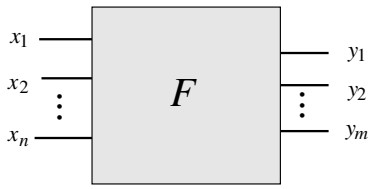

Fig. 2.1. A neural network as a black box

In general we are interested in mapping an n-dimensional real input $(x_1, x_2, \ldots, x_n)$ to an m-dimensional real output $(y_1, y_2, \ldots, y_m)$. A neural

network thus behaves as a "mapping machine", capable of modeling a function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$. If we look at the structure of the network being used, some aspects of its dynamics must be defined more precisely. When the function is evaluated with a network of primitive functions, information flows through the directed edges of the network. Some nodes compute values which are then transmitted as arguments for new computations. If there are no cycles in the network, the result of the whole computation is well-defined and we do not have to deal with the task of synchronizing the computing units. We just assume that the computations take place without delay.

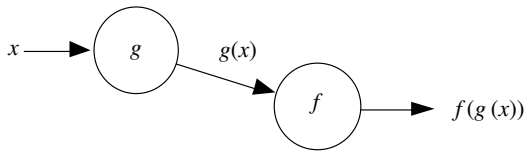

Fig. 2.2. Function composition

If the network contains cycles, however, the computation is not uniquely defined by the interconnection pattern and the temporal dimension must be considered. When the output of a unit is fed back to the same unit, we are dealing with a recursive computation without an explicit halting condition. We must define what we expect from the network: is the fixed point of the recursive evaluation the desired result or one of the intermediate computations? To solve this problem we assume that every computation takes a certain amount of time at each node (for example a time unit). If the arguments for a unit have been transmitted at time t, its output will be produced at time $t + 1$. A recursive computation can be stopped after a certain number of steps and the last computed output taken as the result of the recursive computation.

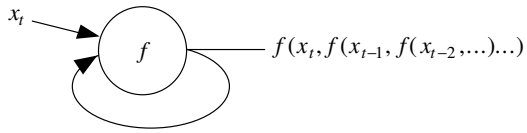

Fig. 2.3. Recursive evaluation

In this chapter we deal first with networks without cycles, in which the time dimension can be disregarded. Then we deal with recurrent networks and their temporal coordination. The first model we consider was proposed in 1943 by Warren McCulloch and Walter Pitts. Inspired by neurobiology they put forward a model of computation oriented towards the computational capabilities of real neurons and studied the question of abstracting universal concepts from specific perceptions [299].

We will avoid giving a general definition of a *neural network* at this point. So many models have been proposed which differ in so many respects that any definition trying to encompass this variety would be unnecessarily clumsy. As we show in this chapter, it is not necessary to start building neural networks with "high powered" computing units, as some authors do [384]. We will start our investigations with the general notion that a neural network is a *network of functions* in which synchronization can be considered explicitly or not.

2.1.2 The computing units

The nodes of the networks we consider will be called *computing elements* or simply *units*. We assume that the edges of the network transmit information in a predetermined direction and the number of incoming edges into a node is not restricted by some upper bound. This is called the *unlimited fan-in* property of our computing units.

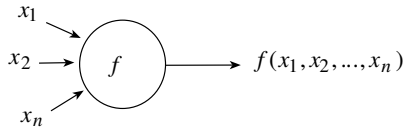

Fig. 2.4. Evaluation of a function of n arguments

The primitive function computed at each node is in general a function of n arguments. Normally, however, we try to use very simple primitive functions of one argument at the nodes. This means that the incoming n arguments have to be reduced to a single numerical value. Therefore computing units are split into two functional parts: an integration function g reduces the n arguments to a single value and the output or activation function f produces the output of this node taking that single value as its argument. Figure 2.5 shows this general structure of the computing units. Usually the integration function g is the addition function.

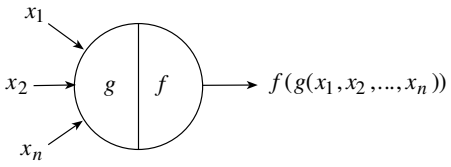

Fig. 2.5. Generic computing unit

McCulloch–Pitts networks are even simpler than this, because they use solely binary signals, i.e., ones or zeros. The nodes produce only binary results

and the edges transmit exclusively ones or zeros. The networks are composed of directed unweighted edges of *excitatory* or of *inhibitory* type. The latter are marked in diagrams using a small circle attached to the end of the edge. Each McCulloch–Pitts unit is also provided with a certain threshold value θ.

At first sight the McCulloch–Pitts model seems very limited, since only binary information can be produced and transmitted, but it already contains all necessary features to implement the more complex models. Figure 2.6 shows an abstract McCulloch–Pitts computing unit. Following Minsky [311] it will be represented as a circle with a black half. Incoming edges arrive at the white half, outgoing edges leave from the black half. Outgoing edges can fan out any number of times.

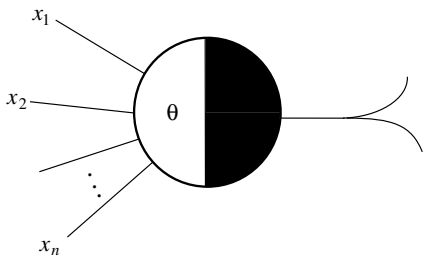

Fig. 2.6. Diagram of a McCulloch–Pitts unit

The rule for evaluating the input to a McCulloch–Pitts unit is the following:

- Assume that a McCulloch–Pitts unit gets an input $x_1, x_2, \ldots, x_n$ through n excitatory edges and an input $y_1, y_2, \ldots, y_m$ through m inhibitory edges.
- If $m \geq 1$ and at least one of the signals $y_1, y_2, \ldots, y_m$ is 1, the unit is inhibited and the result of the computation is 0.
- Otherwise the total excitation $x = x_1 + x_2 + \cdots + x_n$ is computed and compared with the threshold θ of the unit (if $n = 0$ then $x = 0$). If $x \geq \theta$ the unit *fires* a 1, if $x < \theta$ the result of the computation is 0.

This rule implies that a McCulloch–Pitts unit can be inactivated by a single inhibitory signal, as is the case with some real neurons. When no inhibitory signals are present, the units act as a *threshold gate* capable of implementing many other logical functions of n arguments.

Figure 2.7 shows the activation function of a unit, the so-called step function. This function changes discontinuously from zero to one at θ. When θ is zero and no inhibitory signals are present, we have the case of a unit producing the constant output one. If θ is greater than the number of incoming excitatory edges, the unit will never fire.

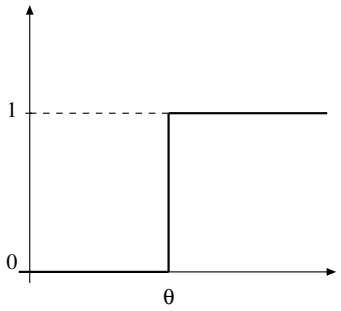

Fig. 2.7. The step function with threshold θ

In the following subsection we assume provisionally that there is no delay in the computation of the output.

2.2 Synthesis of Boolean functions

The power of threshold gates of the McCulloch–Pitts type can be illustrated by showing how to synthesize any given logical function of n arguments. We deal firstly with the more simple kind of logic gates.

2.2.1 Conjunction, disjunction, negation

Mappings from $\{0,1\}^n$ onto $\{0,1\}$ are called logical or Boolean functions. Simple logical functions can be implemented directly with a single McCulloch–Pitts unit. The output value 1 can be associated with the logical value *true* and 0 with the logical value *false*. It is straightforward to verify that the two units of Figure 2.8 compute the functions AND and OR respectively.

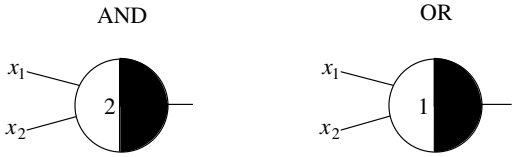

Fig. 2.8. Implementation of AND and OR gates

A single unit can compute the disjunction or the conjunction of n arguments as is shown in Figure 2.9, where the conjunction of three and four arguments is computed by two units. The same kind of computation requires several conventional logic gates with two inputs. It should be clear from this simple example that threshold logic elements can reduce the complexity of the circuit used to implement a given logical function.

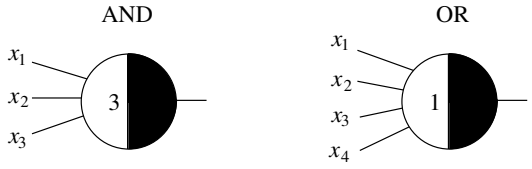

Fig. 2.9. Generalized AND and OR gates

As is well known, AND and OR gates alone cannot be combined to produce all logical functions of n variables. Since uninhibited threshold logic elements are capable of implementing more general functions than conventional AND or OR gates, the question of whether they can be combined to produce all logical functions arises. Stated another way: is inhibition of McCulloch–Pitts units necessary or can it be dispensed with? The following proposition shows that it is necessary. A monotonic logical function f of n arguments is one whose value at two given n-dimensional points $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ is such that $f(x) \geq f(y)$ whenever the number of ones in the input y is a subset of the ones in the input x. An example of a non-monotonic logical function of one argument is logical negation.

Proposition 1. *Uninhibited threshold logic elements of the McCulloch–Pitts type can only implement monotonic logical functions.*

Proof. An example shows the kind of argumentation needed. Assume that the input vector $(1, 1, \ldots, 1)$ is assigned the function value 0. Since no other vector can set more edges in the network to 1 than this vector does, any other input vector can also only be evaluated to 0. In general, if the ones in the input vector y are a subset of the ones in the input vector x, then the first cannot set more edges to 1 than x does. This implies that $f(x) \geq f(y)$, as had to be shown. □

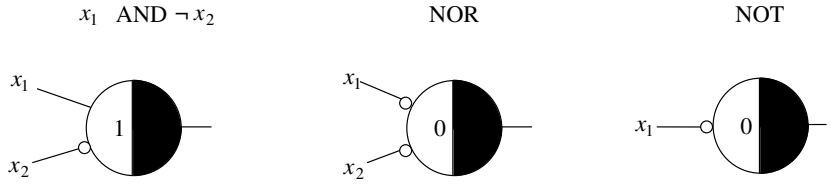

Fig. 2.10. Logical functions and their realization

The units of Figure 2.10 show the implementation of some non-monotonic logical functions requiring inhibitory connections. Logical negation, for example, can be computed using a McCulloch–Pitts unit with threshold 0 and an inhibitory edge. The other two functions can be verified by the reader.

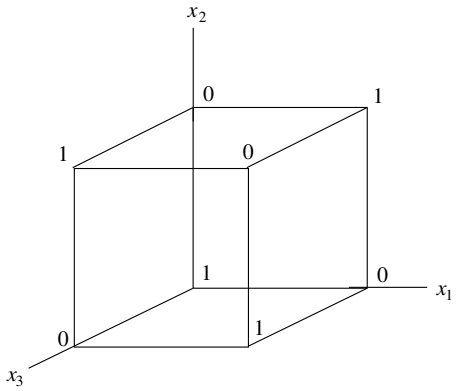

Fig. 2.11. Function values of a logical function of three variables

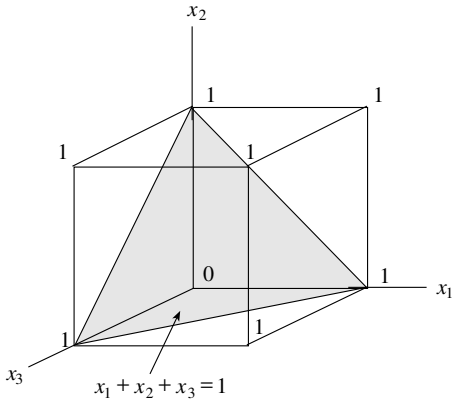

Fig. 2.12. Separation of the input space for the OR function

2.2.2 Geometric interpretation

It is very instructive to visualize the kind of functions that can be computed with McCulloch–Pitts cells by using a diagram. Figure 2.11 shows the eight vertices of a three-dimensional unit cube. Each of the three logical variables x_1, x_2 and x_3 can assume one of two possible binary values. There are eight possible combinations, represented by the vertices of the cube. A logical function is just an assignment of a 0 or a 1 to each of the vertices. The figure shows one of these assignments. In the case of n variables, the cube consists of 2^n vertices and admits 2^{2^n} different binary assignments.

McCulloch–Pitts units divide the input space into two half-spaces. For a given input (x_1, x_2, x_3) and a threshold θ the condition $x_1 + x_2 + x_3 \geq \theta$ is tested, which is true for all points to one side of the plane with the equation $x_1 + x_2 + x_3 = \theta$ and false for all points to the other side (without including the plane itself in this case). Figure 2.12 shows this separation for the case in

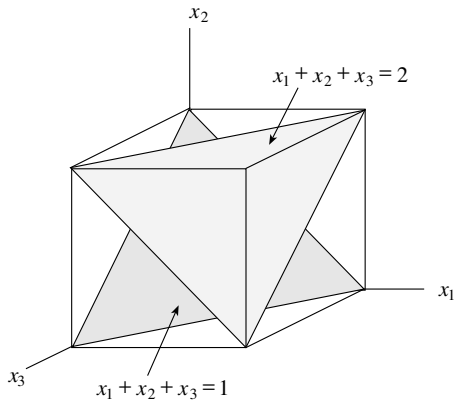

Fig. 2.13. Separating planes of the OR and majority functions

which $\theta = 1$, i.e., for the OR function. Only those vertices above the separating plane are labeled 1.

The majority function of three variables divides input space in a similar manner, but the separating plane is given by the equation $x_1 + x_2 + x_3 = 2$. Figure 2.13 shows the additional plane. The planes are always parallel in the case of McCulloch–Pitts units. Non-parallel separating planes can only be produced using weighted edges.

2.2.3 Constructive synthesis

Every logical function of n variables can be written in tabular form. The value of the function is written down for every one of the possible binary combinations of the n inputs. If we want to build a network to compute this function, it should have n inputs and one output. The network must associate each input vector with the correct output value. If the number of computing units is not limited in some way, it is always possible to build or synthesize a network which computes this function. The constructive proof of this proposition profits from the fact that McCulloch–Pitts units can be used as binary decoders.

Consider for example the vector $(1, 0, 1)$. It is the only one which fulfills the condition $x_1 \wedge \neg x_2 \wedge x_3$. This condition can be tested by a single computing unit (Figure 2.14). Since only the vector $(1, 0, 1)$ makes this unit fire, the unit is a decoder for this input.

Assume that a function F of three arguments has been defined according to the following table:

To compute this function it is only necessary to decode all those vectors for which the function's value is 1. Figure 2.15 shows a network capable of computing the function F.

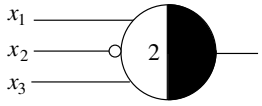

Fig. 2.14. Decoder for the vector $(1, 0, 1)$

input vectors	F
(0,0,1)	1
(0,1,0)	1
all others	0

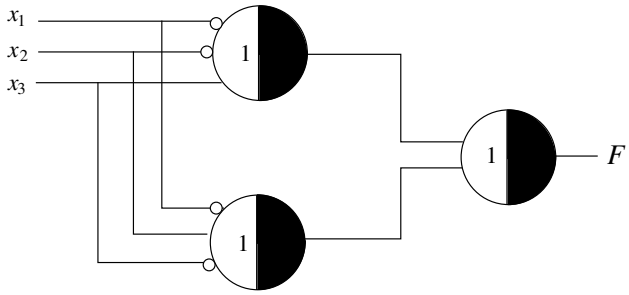

Fig. 2.15. Synthesis of the function F

The individual units in the first layer of the composite network are decoders. For each vector for which F is 1 a decoder is used. In our case we need just two decoders. Components of each vector which must be 0 are transmitted with inhibitory edges, components which must be 1 with excitatory ones. The threshold of each unit is equal to the number of bits equal to 1 that must be present in the desired input vector. The last unit to the right is a disjunction: if any one of the specified vectors can be decoded this unit fires a 1.

It is straightforward to extend this constructive method to other Boolean functions of any other dimension. This leads to the following proposition:

Proposition 2. *Any logical function* $F : \{0,1\}^n \rightarrow \{0,1\}$ *can be computed with a McCulloch–Pitts network of two layers.*

No attempt has been made here to minimize the number of computing units. In fact, we need as many decoders as there are ones in the table of function values. An alternative to this simple constructive method is to use harmonic analysis of logical functions, as will be shown in Sect. 2.5.

We can also consider the minimal possible set of building blocks needed to implement arbitrary logical functions when the fan-in of the units is bounded

in some way. The circuits of Figure 2.14 and Figure 2.15 use decoders of n inputs. These decoders can be built of simpler cells, for example, two units capable of respectively implementing the AND function and negation. Inhibitory connections in the decoders can be replaced with a negation gate. The output of the decoders is collected at a conjunctive unit. The decoder of Figure 2.14 can be implemented as shown in Figure 2.16. The only difference from the previous decoder are the negated inputs and the higher threshold in the AND unit. All decoders for a row of the table of a logical function can be designed in a similar way. This immediately leads to the following proposition:

Proposition 3. *All logical functions can be implemented with a network composed of units which exclusively compute the AND, OR, and NOT functions.*

The three units AND, NOT and OR are called a logical basis because of this property. Since OR can be implemented using AND and NOT units, these two alone constitute a logical basis. The same happens with OR and NOT units. John von Neumann showed that through a redundant coding of the inputs (each variable is transmitted through two lines) AND and OR units alone can constitute a logical basis [326].

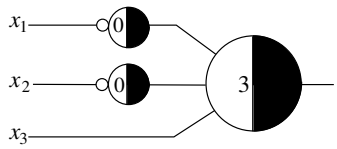

Fig. 2.16. A composite decoder for the vector $(0, 0, 1)$

2.3 Equivalent networks

We can build simpler circuits by using units with more general properties, for example weighted edges and relative inhibition. However, as we show in this section, circuits of McCulloch–Pitts units can emulate circuits built out of high-powered units by exploiting the trade-off between the complexity of the network versus the complexity of the computing units.

2.3.1 Weighted and unweighted networks

Since McCulloch–Pitts networks do not use weighted edges the question of whether weighted networks are more general than unweighted ones must be answered. A simple example shows that both kinds of networks are equivalent.

Assume that three weighted edges converge on the unit shown in Figure 2.17. The unit computes

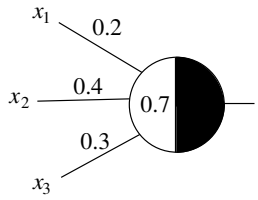

Fig. 2.17. Weighted unit

$$0.2x_1 + 0.4x_2 + 0.3x_3 \geq 0.7.$$

But this is equivalent to

$$2x_1 + 4x_2 + 3x_3 \geq 7,$$

and this computation can be performed with the network of Figure 2.18.

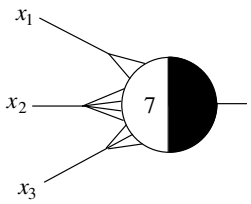

Fig. 2.18. Equivalent computing unit

The figure shows that positive rational weights can be simulated by simply fanning-out the edges of the network the required number of times. This means that we can either use weighted edges or go for a more complex topology of the network, with many redundant edges. The same can be done in the case of irrational weights if the number of input vectors is finite (see Chap. 3, Exercise 3).

2.3.2 Absolute and relative inhibition

In the last subsection we dealt only with the case of positive weights. Two classes of inhibition can be identified: *absolute* inhibition corresponds to the one used in McCulloch–Pitts units. *Relative* inhibition corresponds to the case of edges weighted with a negative factor and whose effect is to lower the firing threshold when a 1 is transmitted through this edge.

Proposition 4. *Networks of McCulloch–Pitts units are equivalent to networks with relative inhibition.*

Proof. It is only necessary to show that each unit in a network where relative inhibition is used is equivalent to one or more units in a network where absolute inhibition is used. It is clear that it is possible to implement absolute inhibition with relative inhibitory edges. If the threshold of a unit is the integer m and if n excitatory edges impinge on it, the maximum possible total excitation for this unit is $n - m$. If $m \geq n$ the unit never fires and the inhibitory edge is irrelevant. It suffices to fan out the inhibitory edge $n - m + 1$ times and make all these edges meet at the unit. When a 1 is transmitted through the inhibitory edges the total amount of inhibition is $n - m + 1$ and this shuts down the unit. To prove that relative inhibitory edges can be simulated with absolute inhibitory ones, refer to Figure 2.19. The network to the left contains a relative inhibitory edge, the network to the right absolute inhibitory ones. The reader can verify that the two networks are equivalent. Relative inhibitory edges correspond to edges weighted with -1. We can also accept any other negative weight w. In that case the threshold of the unit to the right of Figure 2.19 should be $m + w$ instead of $m + 1$. Therefore networks with negative weights can be simulated using unweighted McCulloch–Pitts elements. □

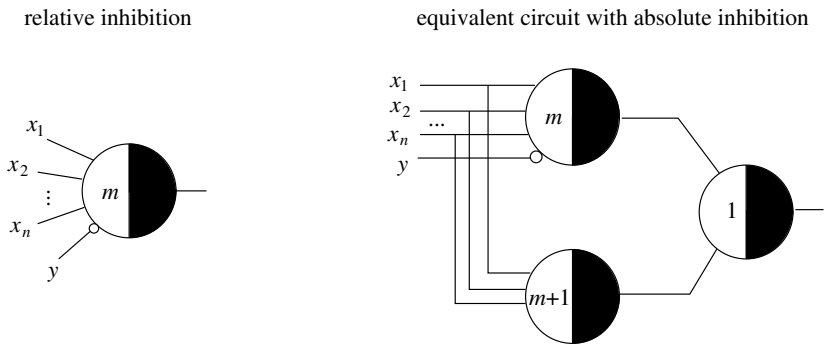

Fig. 2.19. Two equivalent networks

As shown above, we can implement any kind of logical function using unweighted networks. What we trade is the simplicity of the building blocks for a more convoluted topology of the network. Later we will always use weighted networks in order to simplify the topology.

2.3.3 Binary signals and pulse coding

An additional question which can be raised is whether binary signals are not a very limited coding strategy. Are networks in which the communication channels adopt any of ten or fifteen different states more efficient than channels which adopt only two states, as in McCulloch–Pitts networks? To give an

answer we must consider that unit states have a price, in biological networks as well as in artificial ones. The transmitted information must be optimized using the number of available switching states.

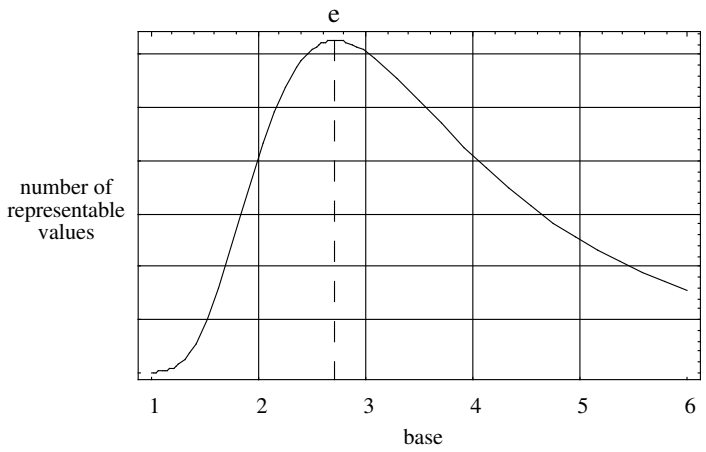

Fig. 2.20. Number of representable values as a function of the base

Assume that the number of states per communication channel is b and that c channels are used to input information. The cost K of the implementation is proportional to both quantities, i.e., $K = \gamma bc$, where γ is a proportionality constant. Using c channels with b states, b^c different numbers can be represented. This means that $c = K/\gamma b$ and, if we set $\kappa = K/\gamma$, we are seeking the numerical base b which optimizes the function $b^{\kappa/b}$. Since we assume constant cost, κ is a constant. Figure 2.20 shows that the optimal value for b is the Euler constant e. Since the number of channel states must be an integer, three states would provide a good approximation to the optimal coding strategy. However, in electronic and biological systems decoding of the signal plays such an important role that the choice of two states per channel becomes a better alternative.

Wiener arrived at a similar conclusion through a somewhat different argument [452]. The binary nature of information transmission in the nervous system seems to be an efficient way to transport signals. However, in the next chapters we will assume that the communication channels can transport *arbitrary real numbers*. This makes the analysis simpler than when we have to deal explicitly with frequency modulated signals, but does not lead to a minimization of the resources needed for a technical implementation. Some researchers prefer to work with so-called *weightless networks* which operate exclusively with binary data.

2.4 Recurrent networks

We have already shown that feed-forward networks can implement arbitrary logical functions. In this case the dimension of the input and output data is predetermined. In many cases, though, we want to perform computations on an input of variable length, for example, when adding two binary numbers being fed bit for bit into a network, which in turn produces the bits of the result one after the other. A feed-forward network cannot solve this problem because it is not capable of keeping track of previous results and, in the case of addition, the carry bit must be stored in order to be reused. This kind of problem can be solved using recurrent networks, i.e., networks whose partial computations are recycled through the network itself. Cycles in the topology of the network make storage and reuse of signals possible for a certain amount of time after they are produced.

2.4.1 Stored state networks

McCulloch–Pitts units can be used in recurrent networks by introducing a temporal factor in the computation. We will assume that computation of the activation of each unit consumes a time unit. If the input arrives at time t the result is produced at time $t + 1$. Up to now, we have been working with units which produce results without delay. The numerical capabilities of any feed-forward network with instantaneous computation at the nodes can be reproduced by networks of units with delay. We only have to take care to coordinate the arrival of the input values at the nodes. This could make the introduction of additional computing elements necessary, whose sole mission is to insert the necessary delays for the coordinated arrival of information. This is the same problem that any computer with clocked elements has to deal with.

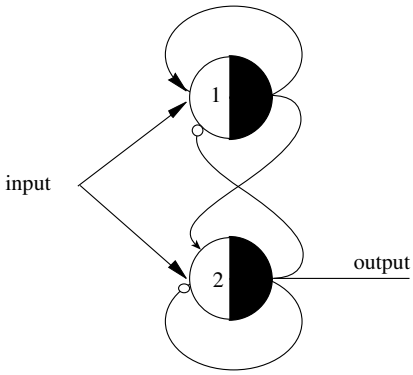

Fig. 2.21. Network for a binary scaler

Figure 2.21 shows a simple example of a recurrent circuit. The network processes a sequence of bits, giving off one bit of output for every bit of input, but in such a way that any two consecutive ones are transformed into the sequence 10. The binary sequence 00110110 is transformed for example into the sequence 00100100. The network recognizes only two consecutive ones separated by at least a zero from a similar block.

2.4.2 Finite automata

The network discussed in the previous subsection is an example of an automaton. This is an abstract device capable of assuming different states which change according to the received input. The automaton also produces an output according to its momentary state. In the previous example, the state of the automaton is the specific combination of signals circulating in the network at any given time. The set of possible states corresponds to the set of all possible combinations of signals traveling through the network.

state transitions

input \ state	Q_0	Q_1
0	Q_0	Q_0
1	Q_1	Q_1

output table

input \ state	Q_0	Q_1
0	0	1
1	0	1

Fig. 2.22. State tables for a binary delay

Finite automata can take only a finite set of possible states and can react only to a finite set of input signals. The state transitions and the output of an automaton can be specified with a table, like the one shown in Figure 2.22. This table defines an automaton which accepts a binary signal at time t and produces an output at time $t+1$. The automaton has two states, Q_0 and Q_1, and accepts only the values 0 or 1. The first table shows the state transitions, corresponding to each input and each state. The second table shows the output values corresponding to the given state and input. From the table we can see that the automaton switches from state Q_0 to state Q_1 after accepting the input 1. If the input bit is a 0, the automaton remains in state Q_0. If the state of the automaton is Q_1 the output at time $t+1$ is 1 regardless of whether 1 or 0 was given as input at time t. All other possibilities are covered by the rest of the entries in the two tables.

The diagram in Figure 2.23 shows how the automaton works. The values at the beginning of the arrows represent an input bit for the automaton. The values in the middle of the arrows are the output bits produced after each

new input. An input of 1, for example, produces the transition from state Q_0 to state Q_1 and the output 0. The input 0 produces a transition to state Q_0. The automaton is thus one that stores only the last bit of input in its current state.

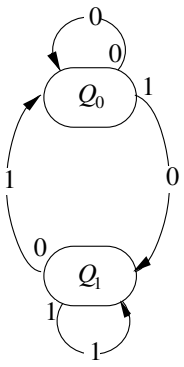

Fig. 2.23. Diagram of a finite automaton

Finite automata without input from the outside, i.e., free-wheeling automata, unavoidably fall in an infinite loop or reach a final constant state. This is why finite automata cannot cover all computable functions, for whose computation an infinite number of states are needed. A Turing machine achieves this through an infinite storage band which provides enough space for all computations. Even a simple problem like the multiplication of two arbitrary binary numbers presented sequentially cannot be solved by a finite automaton. Although our computers are finite automata, the number of possible states is so large that we consider them as universal computing devices for all practical purposes.

2.4.3 Finite automata and recurrent networks

We now show that finite automata and recurrent networks of McCulloch–Pitts units are equivalent. We use a variation of a constructive proof due to Minsky [311].

Proposition 5. *Any finite automaton can be simulated with a network of McCulloch–Pitts units.*

Proof. Figure 2.24 is a diagram of the network needed for the proof. Assume that the input signals are transmitted through the input lines I_1 to I_m and at each moment t only one of these lines is conducting a 1. All other input lines are passive (set to 0). Assume that the network starts in a well-defined

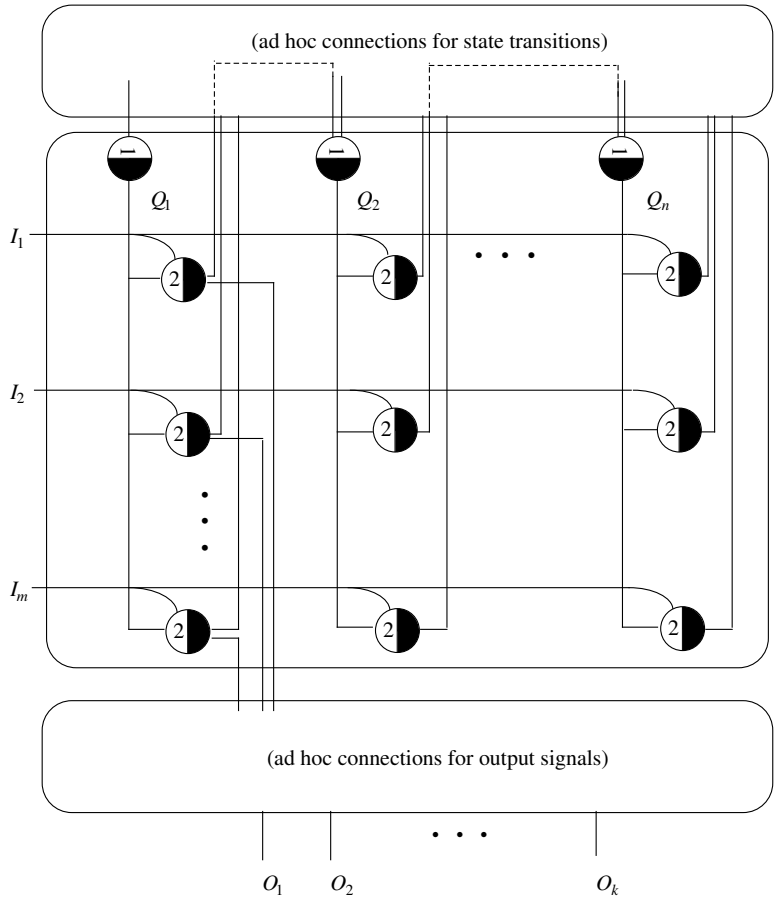

Fig. 2.24. Implementation of a finite automaton with McCulloch–Pitts units

state Q_i. This means that one, and only one, of the lines labeled $Q_1, \ldots, Q_n$ is set to 1 and the others to 0. At time $t+1$ only one of the AND units can produce a 1, namely the one in which both input and state line are set to 1. The state transition is controlled by the ad hoc connections defined by the user in the upper box. If, for example, the input I_1 and the state Q_1 at time t produce the transition to state Q_2 at time $t+1$, then we have to connect the output of the upper left AND unit to the input of the OR unit with the output line named Q_2 (dotted line in the diagram). This output will become active at time $t+2$. At this stage a new input line must be set to 1 (for example I_2) and a new state transition will be computed (Q_n in our example). The connections required to produce the desired output are defined in a similar way. This can be controlled by connecting the output of each AND unit to

the corresponding output line $O_1, \ldots, O_k$ using a box of ad hoc connections similar to the one already described. □

A disadvantage of this constructive method is that each simulated finite automaton requires a special set of connections in the upper and lower boxes. It is better to define a universal network capable of simulating any other finite automaton without having to change the topology of the network (under the assumption of an upper bound for the size of the simulated automata). This is indeed an active field of research in which networks learn to simulate automata [408]. The necessary network parameters are found by a learning algorithm. In the case of McCulloch–Pitts units the available degrees of freedom are given by the topology of the network.

2.4.4 A first classification of neural networks

The networks described in this chapter allow us to propose a preliminary taxonomy of the networks we will discuss in this book. The first clear separation line runs between weighted and unweighted networks. It has already been shown that both classes of models are equivalent. The main difference is the kind of learning algorithm that can be used. In unweighted networks only the thresholds and the connectivity can be adapted. In weighted networks the topology is not usually modified during learning (although we will see some algorithms capable of doing this) and only an optimal combination of weights is sought.

The second clear separation is between synchronous and asynchronous models. In synchronous models the output of all elements is computed instantaneously. This is always possible if the topology of the network does not contain cycles. In some cases the models contain layers of computing units and the activity of the units in each layer is computed one after the other, but in each layer simultaneously. Asynchronous models compute the activity of each unit independently of all others and at different stochastically selected times (as in Hopfield networks). In these kinds of models, cycles in the underlying connection graph pose no particular problem.

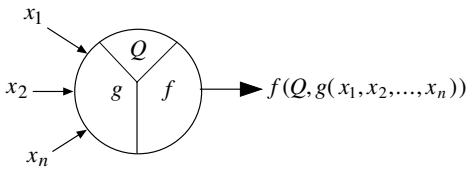

Fig. 2.25. A unit with stored state Q

Finally, we can distinguish between models with or without stored unit states. In Figure 2.5 we gave an example of a unit without stored state. Figure 2.25 shows a unit in which a state Q is stored after each computation.

The state Q can modify the output of the unit in the following activation. If the number of states and possible inputs is finite, we are dealing with a finite automaton. Since any finite automaton can be simulated by a network of computing elements without memory, these units with a stored state can be substituted by a network of McCulloch–Pitts units. Networks with stored-state units are thus equivalent to networks *without* stored-state units. Data is stored in the network itself and in its pattern of recursion.

It can be also shown that time varying weights and thresholds can be implemented in a network of McCulloch–Pitts units using cycles, so that networks with time varying weights and thresholds are equivalent to networks with constant parameters, whenever recursion is allowed.

2.5 Harmonic analysis of logical functions

An interesting alternative for the automatic synthesis of logic functions and for a quantification of their implementation complexity is to do an analysis of the distribution of its non-zero values using the tools of harmonic analysis. Since we can tabulate the values of a logical function in a sequence, we can think of it as a one-dimensional function whose fundamental "frequencies" can be extracted using the appropriate mathematical tools. We will first deal with this problem in a totally general setting and then show that the Hadamard–Walsh transform is the tool we are looking for.

2.5.1 General expression

Assume that we are interested in expressing a function $f : \mathbb{R}^m \rightarrow \mathbb{R}$ as a linear combination of n functions $f_1, f_2, \ldots, f_n$ using the n constants $a_1, a_2, \ldots, a_n$ in the following form

$$f = a_1 f_1 + a_2 f_2 + \cdots + a_n f_n.$$

The domain and range of definition are the same for f and the base functions.

We can determine the quadratic error E of the approximation in the whole domain of definition V for given constants $a_1, \ldots, a_n$ by computing

$$E = \int_V (f - (a_1 f_1 + a_2 f_2 + \cdots + a_n f_n))^2 dV.$$

Here we are assuming that f and the functions f_i, $i = 1, \ldots, n$, are integrable in its domain of definition V. Since we want to minimize the quadratic error E we compute the partial derivatives of E with respect to $a_1, a_2, \ldots, a_n$ and set them to zero:

$$\frac{dE}{da_i} = -2 \int_V f_i (f - a_1 f_1 - a_2 f_2 - \cdots - a_n f_n) dV = 0, \text{ for } i = 1, \ldots, n$$

This leads to the following set of n equations expressed in a simplified notation:

$$a_1 \int f_i f_1 + a_2 \int f_i f_2 + \cdots + a_n \int f_i f_n = \int f_i f, \text{ for } i = 1, \ldots, n.$$

Expressed in matrix form the set of equations becomes:

$$\begin{pmatrix} \int f_1 f_1 & \int f_1 f_2 & \cdots & \int f_1 f_n \\ \int f_2 f_1 & \int f_2 f_2 & & \int f_2 f_n \\ \vdots & & \ddots & \vdots \\ \int f_n f_1 & \int f_n f_2 & \cdots & \int f_n f_n \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \int f_1 f \\ \int f_2 f \\ \vdots \\ \int f_n f \end{pmatrix}$$

This expression is very general. The only assumption we have used is the integrability of the partial products of the form $f_i f_j$ and $f_i f$. Since no special assumptions on the integral were used, it is also possible to use a discrete version of this equation. Assume that the function f has been defined at m points and let the symbol $\sum f_i f_j$ stand for $\sum_{k=1}^{m} f_i(x_k) f_j(x_k)$. In this case the above expression transforms to

$$\begin{pmatrix} \sum f_1 f_1 & \sum f_1 f_2 & \cdots & \sum f_1 f_n \\ \sum f_2 f_1 & \sum f_2 f_2 & & \sum f_2 f_n \\ \vdots & & \ddots & \vdots \\ \sum f_n f_1 & \sum f_n f_2 & \cdots & \sum f_n f_n \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \sum f_1 f \\ \sum f_2 f \\ \vdots \\ \sum f_n f \end{pmatrix}$$

The general formula for the polynomial approximation of m data points $(x_1, y_1), \ldots, (x_m, y_m)$ using the primitive functions $x^0, x^1, x^2, \ldots, x^{n-1}$ translates directly into

$$\begin{pmatrix} m & \sum x_i & \cdots & \sum x_i^{n-1} \\ \sum x_i & \sum x_i^2 & & \sum x_i^n \\ \vdots & & \ddots & \vdots \\ \sum x_i^{n-1} & \sum x_i^n & \cdots & \sum x_i^{2n-2} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \\ \vdots \\ \sum x_i^{n-1} y_i \end{pmatrix}$$

In the case of base functions that are mutually orthogonal, the integrals $\int f_k f_j$ vanish when $k \neq j$. In this case the $n \times n$ matrix is diagonal and the above equation becomes very simple. Assume, as in the case of the Fourier transform, that the functions are sines and cosines of the form $\sin(k_i x)$ and $\cos(k_j x)$. Assume that no two sine functions have the same integer wave number k_i and no two cosine functions the same integer wave number k_j. In this case the integral $\int_0^{2\pi} \sin(k_i x) \sin(k_j x)$ is equal to π, whenever $i = j$, otherwise it vanishes. The same is true for the cosine functions. The expression transforms then to

$$\pi \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \int f_1 f \\ \int f_2 f \\ \vdots \\ \int f_n f \end{pmatrix}$$

which is just another way of computing the coefficients for the Fourier approximation of the function f.

2.5.2 The Hadamard–Walsh transform

In our case we are interested in expressing Boolean functions in terms of a set of primitive functions. We adopt bipolar coding, so that now the logical value false is represented by -1 and the logical value true by 1. In the case of n logical variables $x_1, \ldots, x_n$ and the logical functions defined on them, we can use the following set of 2^n primitive functions:

- The constant function $(x_1, \ldots, x_n) \mapsto 1$
- The $\binom{n}{k}$ monomials $(x_1, \ldots, x_n) \mapsto x_{l_1} x_{l_2} \cdots x_{l_k}$, where $k = 1, \ldots, n$ and $l_1, l_2, \ldots, l_k$ is a set of k different indices in $\{1, 2, \ldots, n\}$

All these functions are mutually orthogonal. In the case of two input variables the transformation becomes:

$$4 \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix}$$

In the general case we compute 2^n coefficients using the formula

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{2^n} \end{pmatrix} = H_n \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{2^n} \end{pmatrix}$$

where the matrix H_n is defined recursively as

$$H_n = \frac{1}{2} \begin{pmatrix} H_{n-1} & H_{n-1} \\ -H_{n-1} & H_{n-1} \end{pmatrix}$$

whereby

$$H_1 = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}.$$

The AND function can be expressed using this simple prescription as

$$x_1 \wedge x_2 = \frac{1}{4}(-2 + 2x_1 + 2x_2 + 2x_1 x_2).$$

The coefficients are the result of the following computation:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix}.$$

The expressions obtained for the logical functions can be wired as networks using weighted edges and only two operations, addition and binary multiplication. The Hadamard–Walsh transform is consequently a method for the synthesis of Boolean functions. The next step, that is, the optimization of the number of components, demands additional techniques which have been extensively studied in the field of combinatorics.

2.5.3 Applications of threshold logic

Threshold units can be used in any application in which we want to reduce the execution time of a logic operation to possibly just two layers of computational delay without employing a huge number of computing elements. It has been shown that the *parity* and *majority* functions, for example, cannot be implemented in a fixed number of layers of computation without using an exponentially growing number of conventional logic gates [148, 464], even when unbounded fan-in is used. The majority function k out of n is a threshold function implementable with just a single McCulloch–Pitts unit. Although circuits built from n threshold units can be built using a polynomial number $P(n)$ of conventional gates the main difference is that conventional circuits cannot guarantee a *constant* delay. With threshold elements we can build multiplication or division circuits that guarantee a constant delay for 32 or 64-bit operands. Any symmetric Boolean function of n bits can in fact be built from two layers of computing units using $n+1$ gates [407]. Some authors have developed circuits of threshold networks for fast multiplication and division, which are capable of operating with constant delay for a variable number of data bits [405]. Threshold logic offers thus the possibility of harnessing parallelism at the level of the basic arithmetic operations.

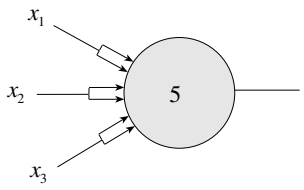

Fig. 2.26. Fault-tolerant gate

Threshold logic also offers a simpler way to achieve fault-tolerance. Figure 2.26 shows an example of a unit that can be used to compute the conjunction of three inputs with inherent fault tolerance. Assume that three inputs x_1, x_2, x_3 can be transmitted, each with probability p of error. The probability of a false result when x_1, x_2 and x_3 are equal, and we are computing the conjunction of the three inputs, is $3p$, since we assume that all three values are transmitted independently of each other. But assume that we transmit

each value using two independent lines. The gate of Figure 2.26 has a threshold of 5, that is, it will produce the correct result even in the case where an input value is transmitted with an error. The probability that exactly two ones arrive as zeros is p^2 and, since there are 15 combinations of two out of six lines, the probability of getting the wrong answer is $15p^2$ in this case. If p is small enough then $15p^2 < 3p$ and the performance of the gate is improved for this combination of input values. Other combinations can be analyzed in a similar way. If threshold units are more reliable than the communication channels, redundancy can be exploited to increase the reliability of any computing system.

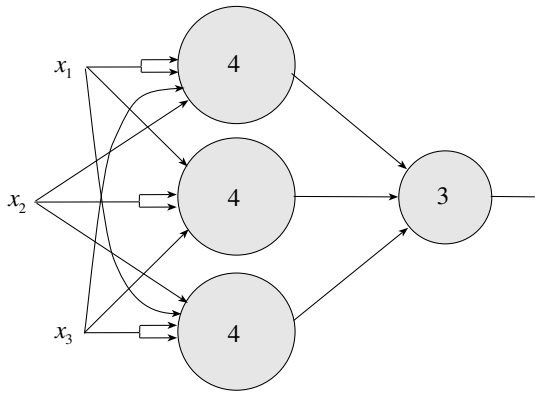

Fig. 2.27. A fault-tolerant AND built of noisy components

When the computing units are unreliable, fault tolerance is achieved using redundant networks. Figure 2.27 is an example of a network built using four units. Assume that the first three units connected directly to the three bits of input x_1, x_2, x_3 all fire with probability 1 when the total excitation is greater than or equal to the threshold θ but also with probability p when it is $\theta - 1$. The duplicated connections add redundancy to the transmitted bit, but in such a way that all three units fire with probability one when the three bits are 1. Each unit also fires with probability p if two out of three inputs are 1. However each unit reacts to a different combination. The last unit, finally, is also noisy and fires any time the three units in the first level fire and also with probability p when two of them fire. Since, in the first level, at most one unit fires when just two inputs are set to 1, the third unit will only fire when all three inputs are 1. This makes the logical circuit, the AND function of three inputs, built out of unreliable components error-proof. The network can be simplified using the approach illustrated in Figure 2.26.

2.6 Historical and bibliographical remarks

Warren McCulloch started pondering networks of artificial neurons as early as 1927 but had problems formulating a general, biologically plausible model since at that time inhibition in neurons had not yet been confirmed. He also had problems with the mathematical treatment of recurrent networks. Inhibition and recurrent pathways in the brain were confirmed in the 1930s and this cleared the way for McCulloch's investigations.

The McCulloch–Pitts model was proposed at a time when Norbert Wiener and Arturo Rosenblueth had started discussing the important role of feedback in biological systems and servomechanisms [301]. Wiener and his circle had published some of these ideas in 1943 [297]. Wiener's book *Cybernetics*, which was the first best-seller in the field of Artificial Intelligence, is the most influential work of this period. The word *cybernetics* was coined by Wiener and was intended to underline the importance of adaptive control in living and artificial systems. Wiener himself was a polymath, capable of doing first class research in mathematics as well as in other fields, such as physiology and medicine.

McCulloch and Pitts' model was superseded rapidly by more powerful approaches. Although threshold elements are, from the combinatorial point of view, more versatile than conventional logic gates, there is a problem with the assumed unlimited fan-in. Current technology has been optimized to handle a limited number of incoming signals into a gate. A possible way of circumventing the electrical difficulties could be the use of optical computing elements capable of providing unlimited fan-in [278]. Another alternative is the definition of a maximal fan-in for threshold elements that could be produced using conventional technology. Some experiments have been conducted in this direction and computers have been designed using exclusively threshold logic elements. The *DONUT* computer of Lewis and Coates was built in the 1960s using 614 gates with a maximal fan-in of 15. The same processor built with NOR gates with a maximal fan-in of 4 required 2127 gates, a factor of approximately 3.5 more components than in the former case [271].

John von Neumann [326] extensively discussed the model of McCulloch and Pitts and looked carefully at its fault tolerance properties. He examined the question of how to build a reliable computing mechanism built of unreliable components. However, he dealt mainly with the redundant coding of the units' outputs and a canonical building block, whereas McCulloch and his collaborator Manuel Blum later showed how to build reliable networks out of general noisy threshold elements [300]. Winograd and Cowan generalized this approach by replicating modules according to the requirements of an error-correcting code [458]. They showed that sparse coding of the input signals, coupled with error correction, makes possible fault-tolerant networks even in the presence of transmission or computational noise [94].

Exercises

1. Design a McCulloch–Pitts unit capable of recognizing the letter "T" digitized in a 10×10 array of pixels. Dark pixels should be coded as ones, white pixels as zeroes.
2. Build a recurrent network capable of adding two sequential streams of bits of arbitrary finite length.
3. Show that no finite automaton can compute the product of two sequential streams of bits of arbitrary finite length.
4. The parity of n given bits is 1 if an odd number of them is equal to 1, otherwise it is 0. Build a network of McCulloch–Pitts units capable of computing the parity function of two, three, and four given bits.
5. How many possible states can assume the binary scaler in Figure 2.21? Write the state and output tables for an equivalent finite automaton.
6. Design a network like the one shown in Figure 2.24 capable of simulating the finite automaton of the previous exercise.
7. Find polynomial expressions corresponding to the OR and XOR Boolean functions using the Hadamard–Walsh transform.
8. Show that the Hadamard–Walsh transform can be computed recursively, so that the number of multiplications becomes $O(n \log n)$, where n is the dimension of the vectors transformed (with n a power of two).
9. What is the probability of error in the case of the fault-tolerant gate shown in Figure 2.26? Consider one, two, and three faulty input bits.
10. The network in Figure 2.27 consists of unreliable computing units. Simplify the network. What happens if the units *and* the transmission channels are unreliable?