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.

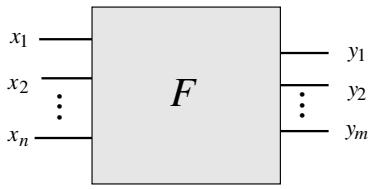

Fig. 2.1. A neural network as a black box

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.

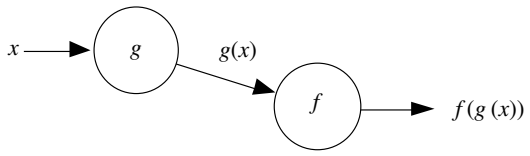

Fig. 2.2. Function composition

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.

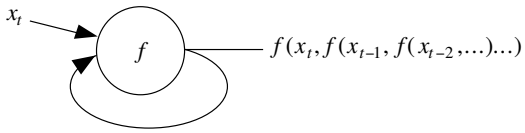

Fig. 2.3. Recursive evaluation

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.

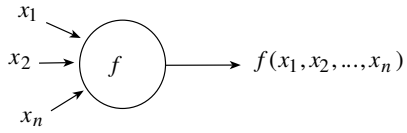

Fig. 2.4. Evaluation of a function of n arguments

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.

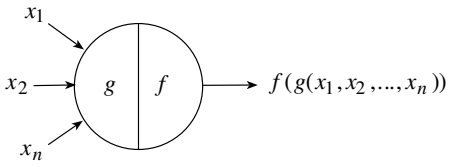

Fig. 2.5. Generic computing unit

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.

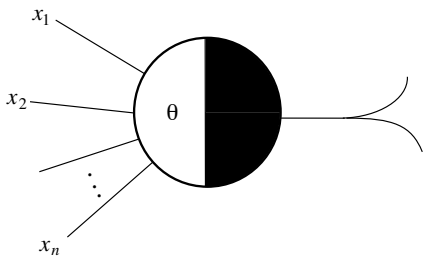

Fig. 2.6. Diagram of a McCulloch–Pitts unit

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.

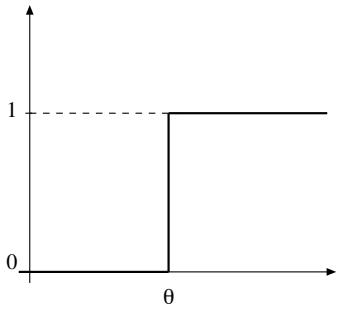

Fig. 2.7. The step function with threshold θ

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.

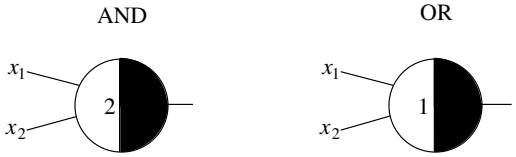

Fig. 2.8. Implementation of AND and OR gates

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.

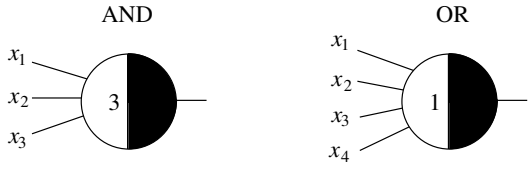

Fig. 2.9. Generalized AND and OR gates

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. □

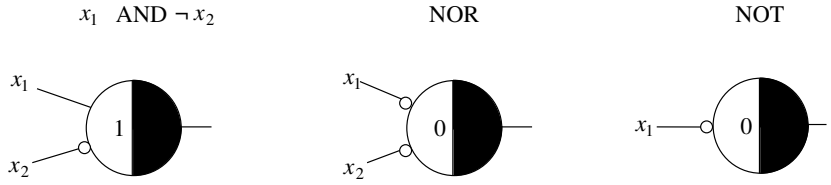

Fig. 2.10. Logical functions and their realization

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.

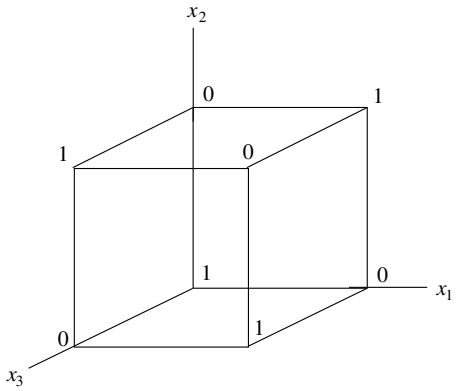

Fig. 2.11. Function values of a logical function of three variables

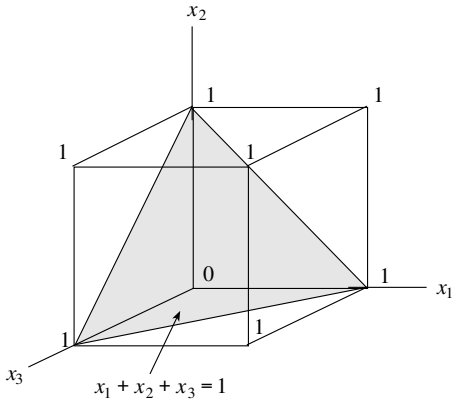

Fig. 2.12. Separation of the input space for the OR function

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

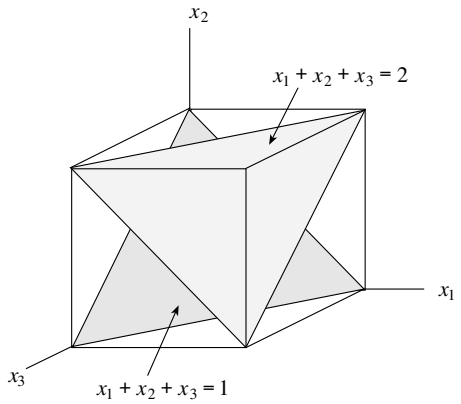

Fig. 2.13. Separating planes of the OR and majority functions

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.

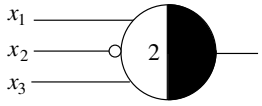

Fig. 2.14. Decoder for the vector $(1, 0, 1)$

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

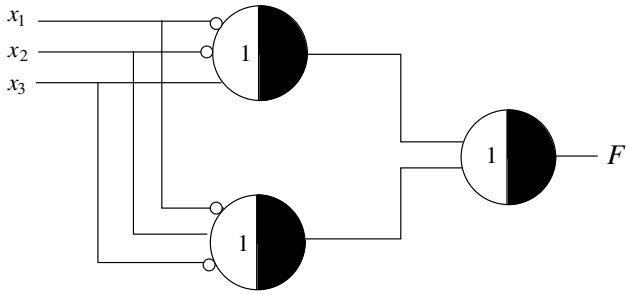

Fig. 2.15. Synthesis of the function F

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].

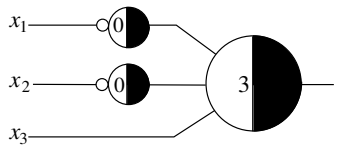

Fig. 2.16. A composite decoder for the vector $(0, 0, 1)$

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

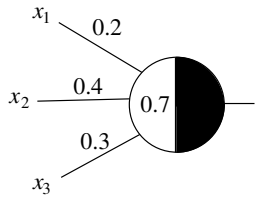

Fig. 2.17. Weighted unit

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

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.

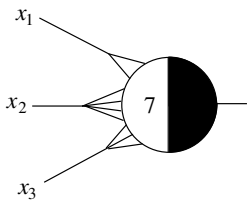

Fig. 2.18. Equivalent computing unit

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. □

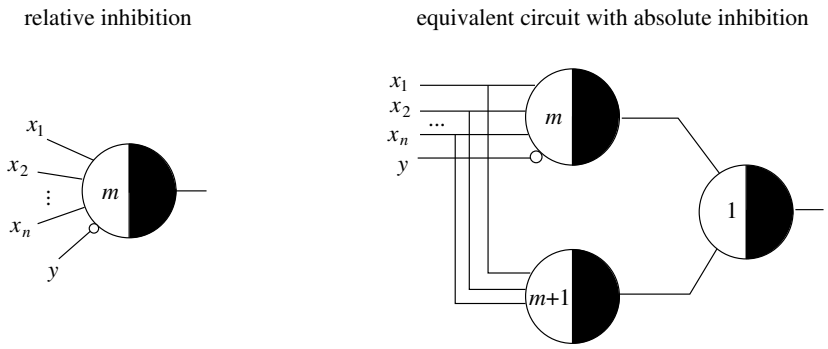

Fig. 2.19. Two equivalent networks

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.

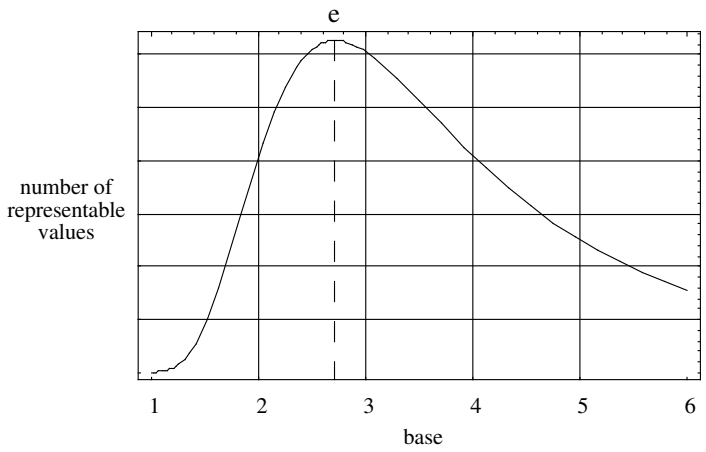

Fig. 2.20. Number of representable values as a function of the base

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.