2. Neural network basics

Next commonalities among different neural networks are discussed in order to get started and show which structural parts or concepts appear in almost all networks. It is presented how neurons or nodes form weighted connections, how neurons create layers, and how activation functions affect the output of a layer. Let us begin with neurons and layers.

2.1 Neurons or nodes and layers

Most neural network structures use some (certain) type of neuron. There are several different kinds of neural networks. Thus, only most common, perhaps most frequently used will be considered.

An algorithm called a neural network is typically composed of individual, interconnected units usually called neurons, nodes or units.

Fig. 2.1 shows the structure of a single artificial neuron that receives input from one or more sources being other neurons or data input. The data can be binary, integers or real (floating point) values. If a symbolic or nominal variable occurs, this must first be encoded with a set of binary variables. For instance, a nominal variable of three alternatives {blue, grey, brown} could be encoded, e.g, with {0,1,2}. Sometimes, bipolar values 1 and -1 are used instead of binary 0 and 1.

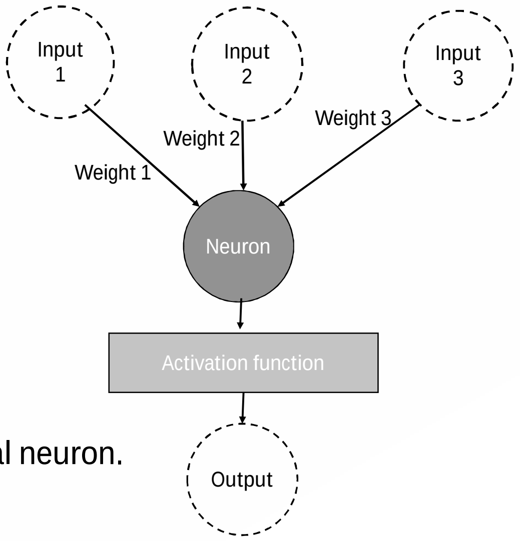

Fig. 2.1 An artificial neuron.

The node or artificial neuron multiplies each of these inputs by a *weight*. Then it adds the multiplications and passes the sum to an activation function. Some neural networks do not use an activation function when their principle is different. The following equation summarises the calculated output:

$$f(\mathbf{x},\mathbf{w}) = \phi(\mathbf{x}\cdot\mathbf{w}) = \phi\left(\sum_{i=1}^{p}(x_i\cdot w_i)\right) \tag{2.1}$$

In the equation, variables x and w represent the input vector and weight vector of the neuron when there are *p* inputs into the neuron. Greek letter ϕ (phi) denotes an activation function. The process results in a single output from a neuron.

Fig. 2.1 shows the structure with just one building component. Such nodes are chained together with many aritifical neurons to construct a network. In Fig. 2.2 there are three neurons. Now the activation functions and intermediate outputs are included implicitly in the nodes and weights in arcs (connections) between nodes. The strucure in Fig. 2.2 could still be a part of a larger network. The input and output are often a special type of neurons, either to accept input data or to generate output values of the network.

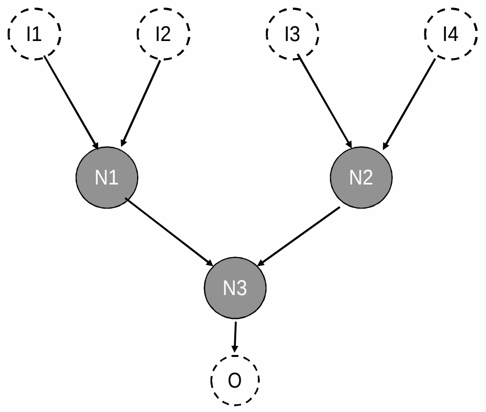

Fig. 2.2 An artificial neural network with four layers of input nodes {I1, I2, I3, I4}, hidden nodes {N1, N2}, {N3} and output node {O}.

In Fig. 2.2. there are four layers called input layer, two hidden layers and ouput layer. Normally, all nodes of a single layer have the same properties like activation function and type like input, hidden or output. Note that these node types are used in feedforward networks, that is multilayer percoptrons. Still, virtually always the nodes of the same layer are of the same type, and input and output have to be taken care of.

The network in Fig. 2.2 is extended in Fig. 2.3 the network of which depicts a common type *feedforward network*, however, a small one as to the numbers of nodes in its layers. Note in the sense of a directed graph data structure it is ”complete” as to arcs between the layers: there exist all possible arcs from each node of a layer to the nodes of the following layer. On the other hand, there are no lateral arcs between the nodes of the same layer in feedforward networks.

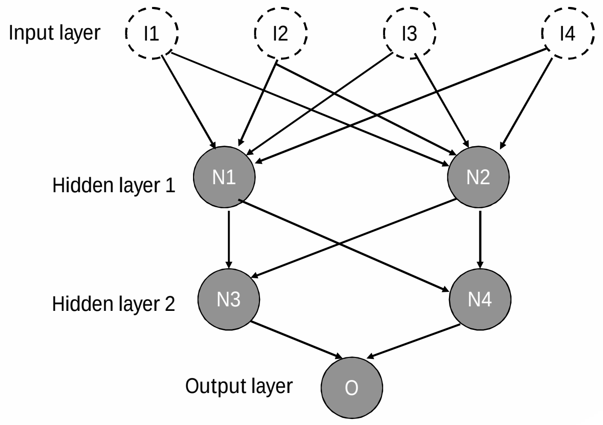

Fig. 2.3 A fully connected feedforward or multilayer perceptron network.

2.2 Types of neurons or nodes

The basic forms of neural networks are typically feedforward ones. Recursive networks do also exist even if obviously they do not have so many and versatile forms compared to the former. As mentioned above, the types or roles of nodes also vary. Sometimes the same node may have more than one role. For instance, Boltzmann machines are an example of a neural network architecture in which nodes are both input and output.

Normally the input to a neural network is represented as an array or vector as in Equation 2.1., in which the vector is of dimension $p=d_j$ and j denotes the layer. For the input layer the dimension d_1 is equal to the number of input variables.

Input, hidden and output nodes

Notice that input nodes do not have activation functions. Thus, they are little more than placeholders. The input is simply weighted and summed. Furthermore, the size of input and output vectors will be the same if the neural network has nodes that are both input and output.

Hidden nodes have two important characteristics. First, they only receive input from the other nodes, such as input or preceding hidden nodes. Second, they only output to other nodes, either as output or other, following hidden nodes. Hidden nodes are not directly connected to the incoming data or to the eventual output. They are often grouped into fully connected hidden layers.

A common question concerns the number of hidden nodes in a network. Since the answer is complex, this question will be considered in different contexts. It is good to notice that the numbers of layers and nodes affect the time complexity of the use of a neural network.

Prior the time of deep learning, it was suggested that one or two hidden layers are enough so that a feedforward network can function virtually as a universal approximator for any mathematical function. Let us remember that if there is one hidden layer, there are two processing layers, the hidden layer and output layer. The above-mentioned approximation of any function is, however, a theoretical thought, because it does not express how the approximation could be made.

Another reason why additional hidden layers seemed to be a problem was that they would require a very extensive training set to be able to compute weights for the network. Before deep learning, the former situation was actually a problem, since deep learning means networks of several hidden layers. Although networks of one or two hidden layers are able to learn ”everything” in theory, deep learning facilitates a more complex representation of patterns in the data.

Bias nodes

Bias nodes are added to feedforward neural networks to help these learn patterns. Bias nodes function like an input node that always produces constant value 1 or other constant. Because of this property, they are not connected to the previous layer. The constant 1 here is called the bias activation. Not all neural networks have bias nodes. Fig. 2.4 depicts a two-hidden-layer network with bias nodes. The network includes three bias nodes. Bias neurons allow the output of an activation function to be shifted. This will be presented later on, in the context of activation functions.

Regardless of the type of neuron, node or processing unit, neural networks almost always are constructed of weighted connections between these units.

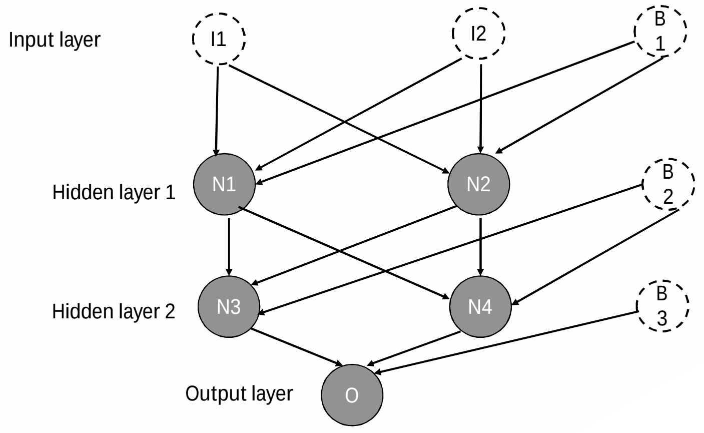

Fig. 2.4 A feedforward network with bias nodes B1, B2 and B3.

2.3 Activation functions

In neurocomputing activation or transfer functions establish bounds for the output of neurons. Neural networks can use several different activation functions. The most common are dealt with in the following.

Selecting an activation function is an important consideration since it can affect how one has to format input data.

At first, the most basic activation function called linear function is shown. It has not practical use, but is rather a starting point.

$$\phi(x) = x \qquad (2.2)$$

It is the identity mapping. See also Fig. 2.6.

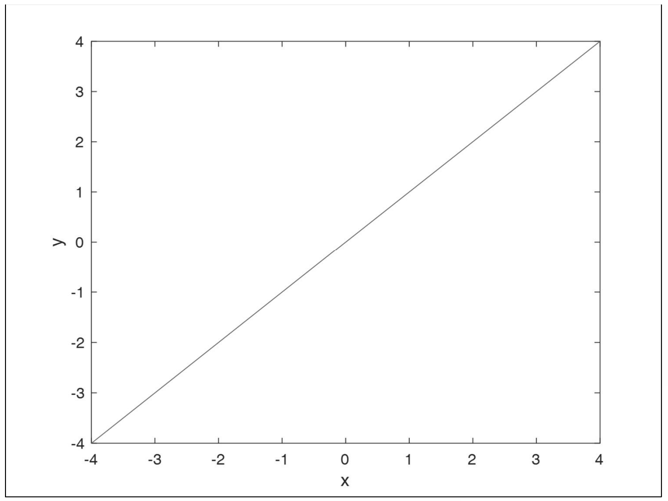

Fig. 2.6 Linear activation function.

Earlier artificial neurons of feedforward networks were called perceptrons. The step or threshold activation function is another simple function. McCulloch and Pitts (1943) introduced it and applied a step activation function:

$$\phi(x) = \begin{cases} 1, \text{if } x \geq 0.5 \\ 0, \text{otherwise} \end{cases} \tag{2.3}$$

Equation (2.3) outputs value 1 for inputs of 0.5 or greater and 0 for all other values. Step functions are also called threshold functions because they only return 1 (true) for those values above some threshold given, e.g., according to Fig. 2.7(a). The next phase is to form a ”ramp” as in Fig. 2.7.(b).

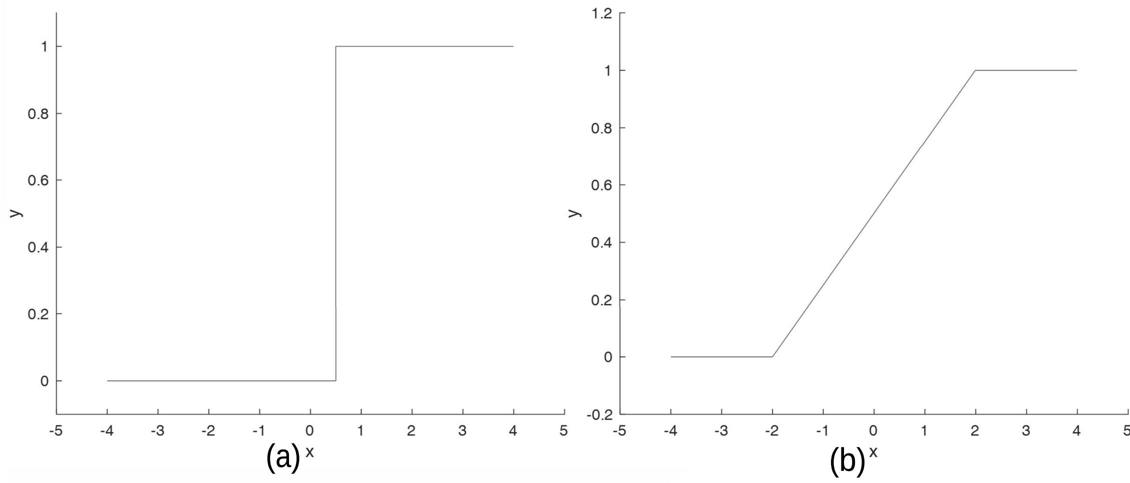

Fig. 2.7 (a) Step activation function; (b) Linear threshold between bounds, otherwise 0 or 1.

The *sigmoid* or *logistic activation function* is a very common choice for feedforward neural networks that need to output only positive values. Despite its extensive use, the hyperbolic tangent or the rectified linear unit (ReLU) function are often more suitable. The sigmoid is as follows.

$$\phi(x) = \frac{1}{1+e^{-x}} \tag{2.4}$$

Its values are restricted between 0 and 1. See Fig. 2.8.

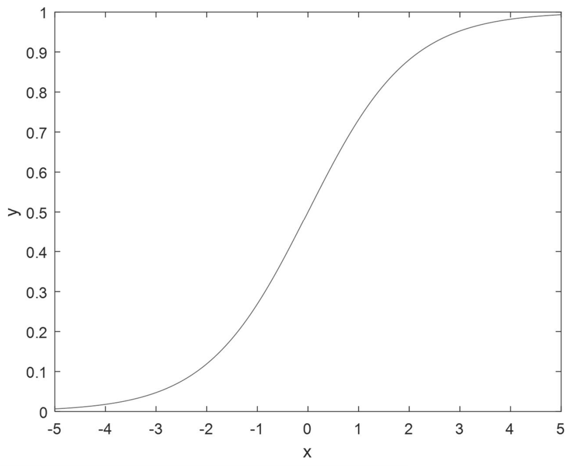

Fig. 2.8 Sigmoid activation function.

The *hyperbolic tangent function* is also one of the most important activation functions. It is restricted into the range between -1 and 1.

$$\phi(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \qquad (2.5)$$

It has a similar shape to the sigmoid function. It has some advantages over the sigmoid function. These involve the derivatives used in the training of the neural network, and they will be covered later for the section of Backpropagation algorithm.

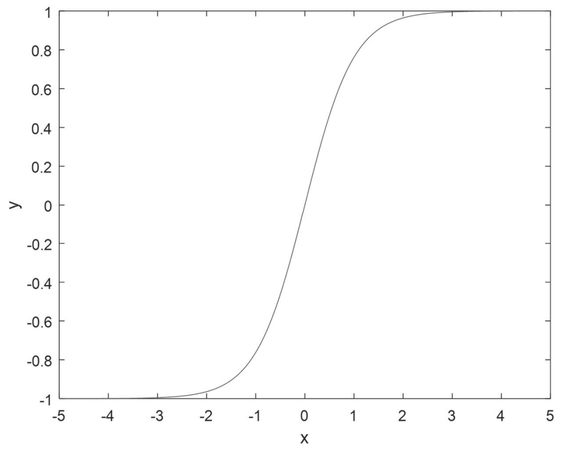

Fig. 2.9 Hyperbolic tangent activation function.

Teh and Hinton (2000) introduced the *rectified linear unit* (ReLU). It is simple and seen a good choice for hidden layers.

$$\phi(x) = \max(0, x) \qquad (2.6)$$

The advantage of the rectified linear unit comes partly from that it is a linear, non-saturating function. Unlike the sigmoid or hyperbolic tangent activation functions, ReLU does not saturate to -1, 0 or 1. See Fig. 2.10. A saturating activation function moves towards and eventually attains a value. For instance, the hyperbolic function saturates to -1 as *x* decreases and to 1 as *x* increases.

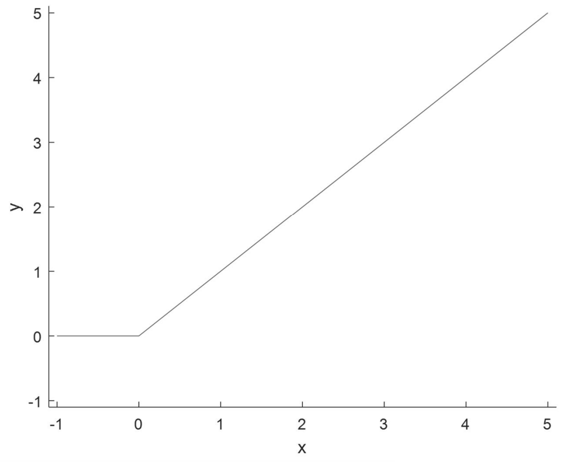

Fig. 2.10 Rectified linear unit activation function.

The final activation function is the *softmax function*. Along with the linear function, softmax is usually found in the output layer of a neural network. The node that has the greatest value claims the input as a member of its class. Because it is a preferable method, the softmax activation function forces the output of the neural network to represent the probability that the input falls into each of the classes. Without the softmax, the node's outputs are simply numeric values, with the greatest indicating the winning class.

Let us recall the iris data containing flowers from three iris species. When we input a data case to the neural network applying the softmax activation function, this allows the network to give the probability that these measurements belong to each of three species. For example, their probabilities could be 80%, 15% and 5%. Since these are probabilities, their sum must add up 100%. Output nodes do not inherently specify the probabilities of the classes. Therefore, softmax is useful, when it produces such probabilites. The softmax function is as follows.

$$\phi_i = \frac{e^{z_i}}{\sum\limits_{j \in group} e^{z_j}} \qquad (2.7)$$

In the formula, *i* represents the index of the output node, and *j* represents the indexes of all nodes in the group or level. The variable *z* designates the array of the output nodes. It is important to note that softmax is computed differently from the other activation functions given. When using softmax, the output of a single node is dependent on the other output nodes. In Equation (2.7), the output of the other output nodes is contained in the variable *z*, unlike in those other activation functions.

The role of bias

Together, the weight *w* and bias *b* of a node shape the output of the activation function. Equation (2.8) represents a single-input sigmoid activation function neural network

$$f(x,w,b) = \frac{1}{1+e^{-(wx+b)}} \qquad (2.8)$$

Eq. (2.8) is the combination of Eq. (2.1) of a neural network and Eq. (2.4) of the sigmoid activation function. Fig. 2.11(a) shows the effect of weight variation on the output of the sigmoid function. Fig. 2.11(b) shows the effect of bias variation.

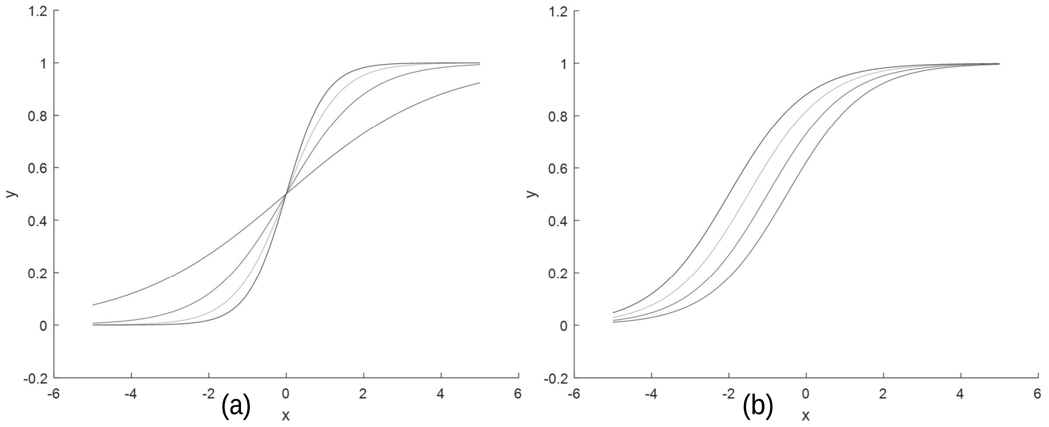

Fig. 2.11(a) Sigmoids for weights *w* in {0.5, 1.0, 1.5, 2.0}, the greater weight, the steeper curve, and (b) bias *b* in {0.5, 1.0, 1.5, 2.0} (*w*=1.0), the greater bias, the leftmost curve because of the shift being not complete when the all curves merge together at the top or bottom left.

2.5 Logic with neural networks

Logical operators can be implemented with neural networks. Let us look at the truth table of operators AND, OR, NOT. Neural networks can represent these according to Fig. 2.12

0 AND 0 =0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
NOT 0 = 1
NOT 1 = 0

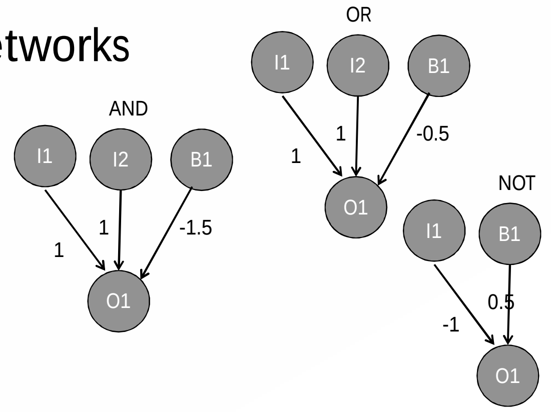

Fig. 2.12 The logical operators as networks. AND with 1 inputs: 1*1+1*1+(-1.5)=0.5>0; true

In Fig. 2.12 the following function is used

$$y = f_h\left(\sum_{i=1}^{p} w_i x_i - b\right) \qquad (2.9)$$

where f_h is a step function named the Heaviside function with *p* variables and bias *b*

$$f_h(x) = 1 \quad x > 0$$
$$f_h(x) = 0 \quad x \le 0 \qquad (2.10)$$

and which produces outputs either 1 or 0.

Perceptron: a vectorial perspective

From Eq. (2.9) (b=w_0, x_0=1) expression

$$\sum_{i=0}^{p} w_i x_i = \mathbf{w} \cdot \mathbf{x} \qquad (2.11)$$

can be represented as a line mapped in 2-dimensional (two variables) Euclidean space to distinguish two separate classes of cases or datapoints. Vector x represents any case in the variable space. (A situation of two fully separate classes is idealistic, in fact, not encountered in actual data sets.)

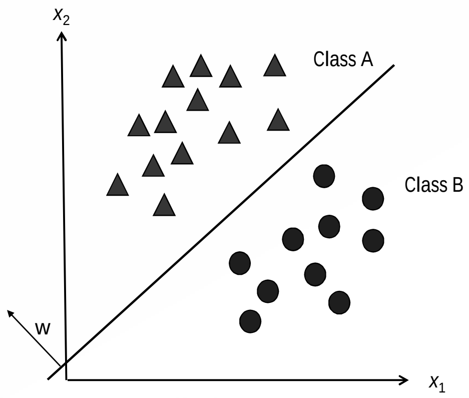

Fig. 2.13 Two distinct sets of cases or patterns in 2-dimensional space.

Exclusive or (XOR)

One can find out easily that XOR is not possible to implement with a single (processing) layer of nodes when looking at Fig. 2.14 and noticing that a single feedforward or perceptron layer can correspond to linear mappings only. Namely, by locating a line at whatever positions it is not possible to distinguish the two classes of true outputs for {(1,0),(0,1)} and false outputs for {(0,0),(1,1)} by using one line only.

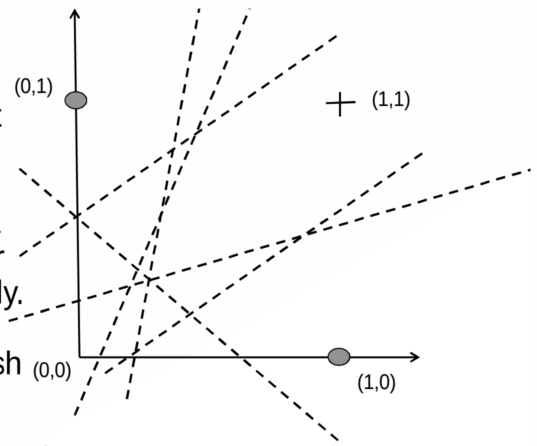

Fig. 2.14 Two classes of XOR cannot be separated with one line.

Using two or more processing layers XOR (operator $\oplus$) for inputs p and q

$$p \oplus q = (p \vee q) \wedge \neg (p \wedge q) \quad (2.12)$$

can be implemented as depicted in Fig. 2.15.

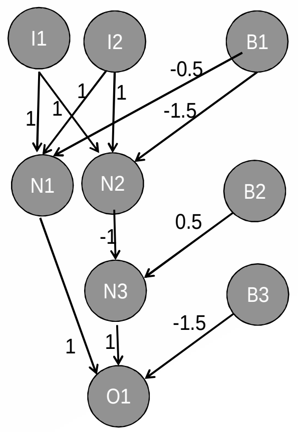

Fig. 2.15 Two classes of XOR can be separated with more than one processing layer.