Representation and Learning in Feedforward Neural Networks

H.L. Trentelman
Mathematics Institute, University of Groningen,
P.O. Box 800, 9700 AV Groningen, The Netherlands,
Email: h.l.trentelman@math.rug.nl

This paper gives an introduction to feedforward neural networks. The aim of this paper is to present some of the basics of artificial neural networks, with a particular emphasis on the following two central issues. The first central issue of this paper is: in what sense do artificial neural networks represent mathematical functions, and what mathematical functions can be (approximately) represented by an artificial neural network? The second central issue of this paper is: what do we mean by 'learning' in artificial neural networks, and how can a network learn to (approximately) represent a given mathematical function?

1. Introduction

In the last decade, research on artificial neural networks has more and more become a popular research field. Going back to the fourties, the study of artificial neural networks was mainly inspired by the desire to gain insight into the principles that underly the functioning of the human brain (what is 'learning', how does the human memory work, what are dreams, etc.). Since long, it is believed that the human brain is built up from a large number ($10^{10} - 10^{11}$) of interconnected, basically identical, elementary units, called neurons. Each neuron is believed to function according to the same, relatively simple, biophysical principles. The idea of artificial neural networks is, roughly speaking, to model these simple biophysical principles into a single mathematical concept, called an *artificial neuron*, and to study interconnections of these artificial neurons as a model of the brain. The study of these interconnections typically takes place by mathematical analysis or by computer simulation, and is hoped to lead to a better understanding of how the brain processes information.

The more recent growth of interest in artificial neural networks seems to be caused by their promise to yield solutions to all kinds of technical problems of 'artificial intelligence' that traditional approaches do not yield. This promise is based on the observation that, while the working of the organic neuron is

based on such simple biophysical principles, the brain is capable of performing immensly complex tasks. This apparent contradiction is explained by the enormous amount of neurons that, in addition, are interconnected in parallel. Following this line of thought, there are reasons to believe that if we build a 'machine' consisting of a massive interconnection of artificial neurons (i.e., a technical realization of an artificial neural network), then this machine is, in principle, capable of performing complex tasks.

The present paper developed out of a three hour introductory talk on neural networks that was given by the author in the context of the so-called 'Reconstructie Seminar', a series of talks on various mathematically and physically oriented scientific subjects that was held within a group of Dutch researchers active in the field of Systems and Control during the academic year '92-'93. The main constraint of the 'Reconstructie Seminar' was that the speaker had to choose his/her subject outside the scope of his/her research area. The aim of this paper is the same as the aim of the talk that was given in the Seminar: to present some of the basics of artificial neural networks, with a particular emphasis on the following two central issues. The first central issue of this paper is: in what sense do artificial neural networks represent mathematical functions, and what mathematical functions can be (approximately) represented by an artificial neural network? The second central issue of this paper is: what do we mean by 'learning' in artificial neural networks, and how can a network learn to (approximately) represent a given mathematical function?

The outline of this paper is as follows. Section 2 is devoted to some of the basics of artificial neural networks. We briefly explain the working of the organic neuron, and introduce the notion of artificial neuron as a rough mathematical model for the organic neuron. We give a formal definition of artificial neural network and explain in what sense feedforward networks define functions in the mathematical sense. We discuss the notion of network architecture, and explain in what sense a network architecture defines a parametrized family of mathematical functions. Next, we explain what we mean by learning in artificial neural networks.

Section 3 is devoted to a discussion of a prototype neural network, the Perceptron. We explain what functions can be represented by a Perceptron. We also discuss the issue of learning in Perceptrons and talk about the famous Perceptron convergence theorem.

In section 4 we deal with general layered feedforward networks. Again, we concentrate in this section on the issues of representation and learning. We discuss some very recent results on the approximate representation of mathematical functions by feedforward networks with one hidden layer. Next, we come back to the issue of learning: in what sense can a layered feedforward network learn a given mathematical function. In this context we explain, for a simple special case, the famous Back Propagation Algorithm.

2. Artificial neural networks

2.1. Neurons

A typical neuron in the human brain consists of a central part, called the cell body or *soma*, and a long tiny tubular fiber originating from this cell body, called the *axon*. Also, the soma serves as the endpoint of a bundle of incoming branches, called the *dendrites*. The axon, in turn, splits into a bundle of tiny branches whose endpoints are called *synapses*. The neuron collects input signals from surrounding neurons via its dendrites. When the total activity of these input signals exceeds a certain value, called the neuron's *threshold value*, then the neuron sends a spike of electrical activity through its axon. This spike of electrical activity branches out to the neuron's synapses. At each synapse, the electrical activity causes an input signal to be send to a neighbouring neuron, via one of its dendrites.

The amount of influence of one neuron on another depends on the effectiveness of the synapse between the two neurons: a certain amount of electrical activity in a neuron causes a certain amount of input activity to be generated at each of its synapses. The more effective a synaps is, the more input activity it will generate. It is believed that the effectiveness of synapses can be subject to changes in time. These changes in effectiveness of synapses or, equivalently, these changes in the amount of influence that neurons have on other neurons, is often used to explain the phenomenon of 'learning'.

2.2. Artificial neurons

As a simple mathematical model to represent the most important features of the organic neuron, McCullogh and Pitts in 1943, [20] proposed the following definition. For a given positive integer n and real numbers $w_1, \ldots, w_n$ and θ, the artificial neuron with weights $w_1, \ldots, w_n$ and threshold θ is the function f from $\{0,1\}^n$ to $\{0,1\}$ given by

$$y = f(x_1, x_2, \ldots, x_n) := \mathcal{H}(\sum_{i=1}^{n} w_i x_i + \theta).$$

Here, $\mathcal{H}$ denotes the Heaviside step function:

$$\mathcal{H}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

If it takes the value 1, the neuron is said to 'fire', otherwise it is said to be 'at rest'.

The analogy with the working of the organic neuron is as follows. At a certain moment the neuron under consideration receives signals from all n neurons to which it is connected. The signal x_j coming from neuron j is either '0' or '1' (corresponding to whether neuron j is firing or at rest). The effectiveness of the synaps between neuron j and the neuron under consideration is measured by the weight w_j. Only if the total weighted sum $\sum_{i=1}^{n} w_j x_j$ of these signals (called the *activation*) exceeds the threshold value θ of the neuron under consideration,

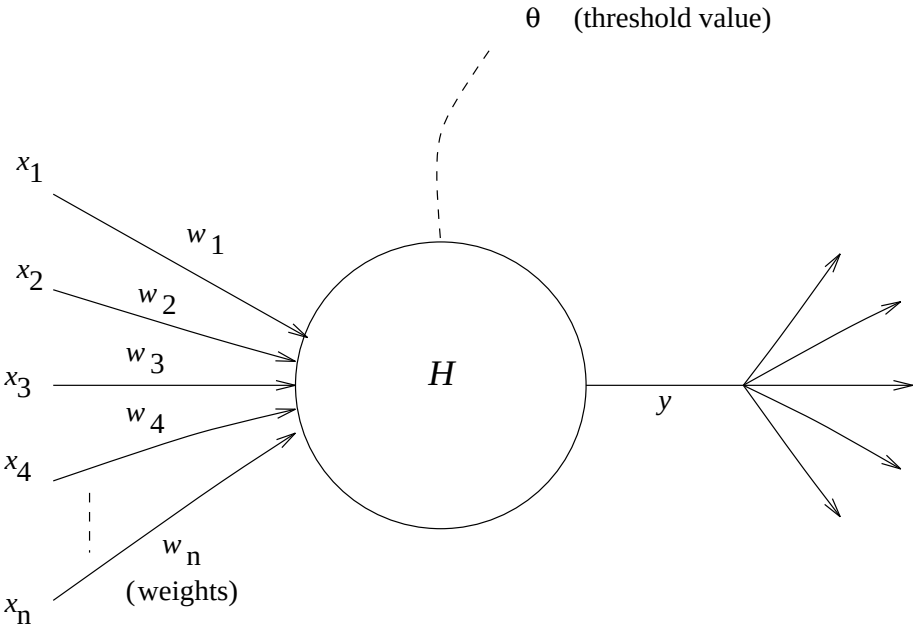

FIGURE 1. Artificial neuron

this neuron is assumed to be sufficiently activated: it will generate the value '1' (fire). If the weighted sum of the input signals is less than or equal to the threshold value, the neuron will generate the output value '0' (remain at rest).

More general, the Heaviside function appearing in this definition can be replaced by an arbitrary function, say $\sigma : \mathbb{R} \to \mathbb{R}$. For a given function σ, a given positive integer n and real numbers $w_1, \ldots, w_n$ and θ, the artificial neuron with weights $w_1, \ldots, w_n$ and threshold θ is the function f from $\mathbb{R}^n$ to $\mathbb{R}$ given by

$$f(x_1, x_2, \ldots, x_n) := \sigma(\sum_{i=1}^{n} w_i x_i + \theta).$$

The function σ is often called the *transfer function*, *activation function*, or *squashing function* of the neuron. As mentioned, the transfer function can in principle be any function. In a large part of the literature on artificial neural networks, the transfer function is choosen to be a so-called *sigmoid function*, i.e., loosely speaking, a function whose graph resembles the shape of the character 'S'. Examples of these are the function given by

$$\sigma(x) = \frac{1}{e^{-\beta x} + 1},$$

and the function given by

$$\sigma(x) = \tanh(\beta x)$$

(with $\beta > 0$).

2.3. Artificial neural networks

Loosely speaking, any interconnection of a finite number of artificial neurons is called an artificial neural network. Formally, an artificial neural network with N neurons is defined to be a directed graph with N nodes, $1, 2 \ldots, N$, where node i is identified with the artificial neuron with transfer function σ_i, and threshold value θ_i, and where the branch from node j to node i is identified with the weight w_{ij}.

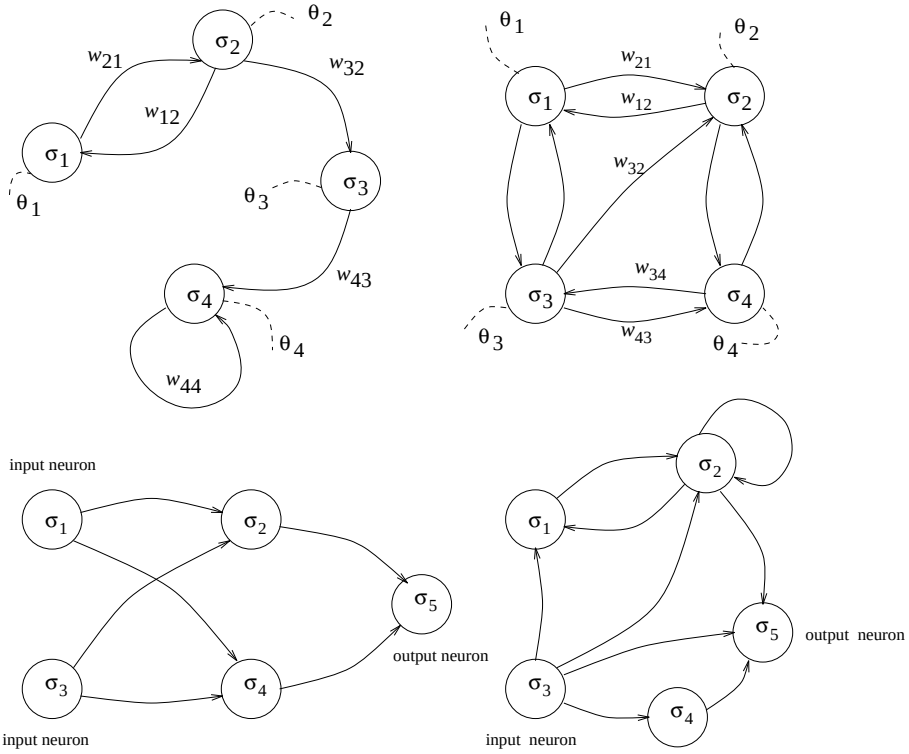

FIGURE 2. Four examples of neural networks

The neurons that correspond to the *sources* in the directed graph are called the *input neurons* of the neural network, while the neurons that correspond to the *sincs* in the graph are called the *output neurons* of the neural network. Any other neuron in the network is called a *hidden neuron.*

If the graph corresponding to the network has *no closed paths*, then the the network is called *a feedforward network.* Speaking in terms of input signals and output signals, in a feedforward network the signals only travel in one direction.

Any network that is not a feedforward network is called a *recursive network*. We will restrict ourselves in this paper to feedforward networks.

Now, after having defined a neuron to be a function of a particular structure, and a neural network to be a directed graph, we explain in what sense a feedforward network performs 'a cognitive task'. Suppose we have a feedforward network with m input neurons, p output neurons, and a number of hidden neurons. Such a network can always be interpreted as a function from $\mathbb{R}^m$ to $\mathbb{R}^p$, in the following way. The input neurons are considered as devices that generate the arguments of the function: input neuron i generates the value x_i. Together, the m input neurons generate the vector $\underline{x} = (x_1, \ldots, x_m)$. Next, the hidden neurons perform operations on these values x_i, according to the particular transfer functions that each hidden neuron has. Finally, the output neuron j generates the value y_j. Together, the p output neurons generate the p-vector $\underline{y} = (y_1, \ldots, y_p)$. In this sense, the neural network performs the task of calculating the value of the output vector $\underline{y}$ from the input vector $\underline{x}$; a feedforward network with m input neurons and p output neurons defines a function $F : \mathbb{R}^m \to \mathbb{R}^p$ (or $F : \mathcal{S} \to \mathbb{R}^p$, with $\mathcal{S}$ a subset of $\mathbb{R}^m$) (see also Figure 3).

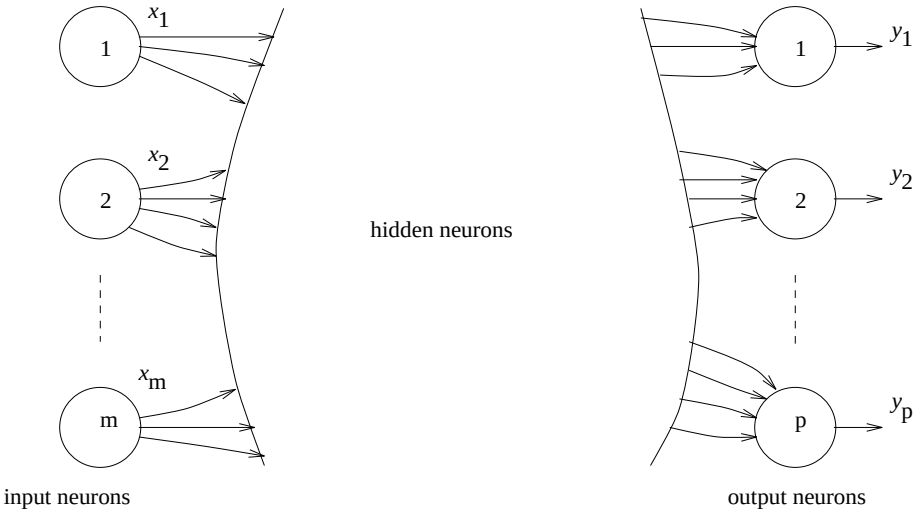

FIGURE 3. Feedforward network representing a function

As an example, consider the following network with two input neurons, two hidden neurons and one output neuron (see Figure 4). Assume that the transfer functions of the hidden neurons and the output neuron are all equal to one and the same function σ. According to the convention introduced above, the first input neuron generates the value x_1, and the second input neuron generates the value x_2. Assuming that the weights of the input channels of the first hidden neuron are equal to v_{11} and v_{12}, respectively, the activation of the first hidden

neuron is equal to $v_{11}x_1 + v_{12}x_2$. This hidden neuron then generates the output value s_1 given by

$$s_1 = \sigma(v_{11}x_1 + v_{12}x_2 + \eta_1),$$

where η_1 is the threshold value of the neuron. Likewise, the second hidden neuron generates the output value

$$s_2 = \sigma(v_{21}x_1 + v_{22}x_2 + \eta_2).$$

Let the weights associated with the output neuron be equal to w_1 and w_2, and let its threshold value be θ. Clearly, the activation of the output neuron is equal to $w_1s_1 + w_2s_2$. The output neuron generates the output value y given by

$$y = \sigma(w_1s_1 + w_2s_2 + \theta).$$

We conclude that this neural network defines a function F from $\mathbb{R}^2$ to $\mathbb{R}$ defined by $F(x_1, x_2) = y$.

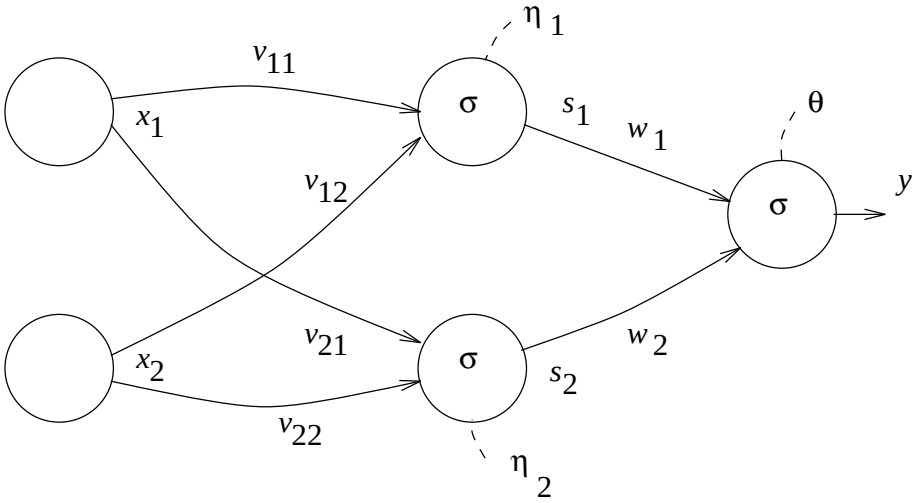

FIGURE 4. Example of feedforward network

2.4. Network architecture

We note that as soon as the transfer functions of the neurons, and the directed graph are specified, the structure of a neural network is completely determined. The only remaining freedom are the values of the weights and the thresholds of the hidden neurons and the output neurons. Of course, the properties of the neural network are highly dependent of the particular value of these parameter values. In order to stress that the weights and threshold values are considered as free parameters, the fixed directed graph together with the fixed

transfer functions are often called *the network architecture*. Given a network architecture, each choice of weights and threshold values yields exactly one neural network. This means that a network architecture can be considered as a parametrized family of functions in the following way: as soon as the directed graph, together with the transfer functions are specified, the number of input neurons (m), and the number of output neurons (p) are fixed. The remaining freedom is exactly given by the weights and thresholds w_{ij}, θ_i. Thus, the network architecture defines a *family of functions* $F_{w_{ij},\theta_i} : \mathbb{R}^m \to \mathbb{R}^p$, parametrized by the w_{ij}'s and θ_i's. In the example above, the parameter is equal to the joint vector $(v_{11}, v_{12}, v_{21}, v_{22}, w_1, w_2, \eta_1, \eta_2, \theta) \in \mathbb{R}^9$. Often, the terminology 'neural network' is used if in fact we are dealing with a network *architecture*. Also in this paper, we will often speak about neural networks as being families of functions parametrized by the weights and thresholds.

2.5. Neural networks, representation, and learning

In the human brain the process of learning takes place. In artificial neural networks the process of learning is modelled as *change of weights and threshold values*. We will come back to this later in this paper. Central issues in the theory of feedforward networks are the following. Suppose a network architecture with m inputs and p outputs is given. In addition, suppose a fixed function $G : \mathbb{R}^m \to \mathbb{R}^p$ is given.

- **Representation.** Does there exist a particular choice of weights and threshold values such that the corresponding network function F is (approximately) equal to G?
- **Learning.** Is it possible for the network architecture to *learn* the function G, i.e., can we come up with some mechanism or algorithm that keeps adapting the values of the weights and thresholds until the resulting network function F is (approximately) equal to G?

3. The Perceptron

A simple example of a feedforward network is the following feedforward network consisting of m input neurons, 1 output neuron and no hidden neurons. The input neurons are labeled $1, 2, \ldots, m$. The weight associated with the connection between input neuron j and the output neuron is equal to w_j. The threshold value of the output neuron is equal to θ. We assume that the output neuron has transfer function, $\mathcal{H}$, the Heaviside step function.

This feedforward network is called *the Perceptron* and was proposed in 1959 by F. Rosenblatt [8]. Let $\underline{w} := (w_1, w_2, \ldots, w_p)$ be the vector consisting of the weights. If the input vector to the network is $\underline{x} = (x_1, x_2, \ldots, x_m)$, then clearly the output generated by the network is equal to

$$y = \mathcal{H}(\underline{w} \cdot \underline{x} + \theta).$$

where $\cdot$ denotes the standard inner product on $\mathbb{R}^m$. As explained above, this can be interpreted by saying that the Perceptron (or rather: the Perceptron

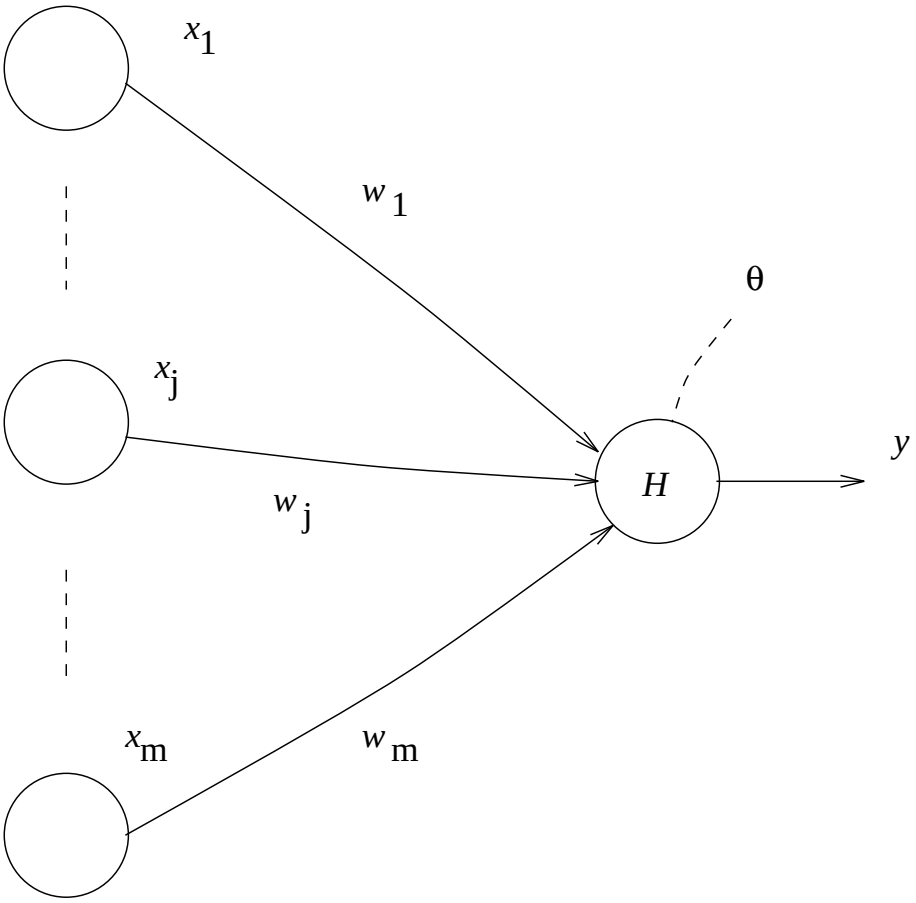

FIGURE 5. The Perceptron

architecture) defines a family of functions, from $\mathbb{R}^m$ to $\{0,1\}$, parametrized by $(\underline{w},\theta) \in \mathbb{R}^{m+1}$, and given by

$$F_{(\underline{w},\theta)}(\underline{x}) = \mathcal{H}(\underline{w}\cdot\underline{x}+\theta).$$

Any particular choice of parameters $(\underline{w},\theta)$ yields exactly one Perceptron.

In this section we will study for the Perceptron the two central issues that were raised in the previous subsection, the issues of *representation* and *learning*. The first issue that we will consider is the issue of representation.

3.1. The Perceptron representation theorem

The first question that we will study is the following: suppose that $\mathcal{S}$ is a given subset of $\mathbb{R}^m$, and $G : \mathcal{S} \to \{0,1\}$ a given function, do there exist parameter values $\underline{w} \in \mathbb{R}^m$ and $\theta \in \mathbb{R}$ such that the corresponding network function $F_{(\underline{w},\theta)}$ restricted to $\mathcal{S}$ is equal to G, i.e. such that for all $\underline{x} \in \mathcal{S}$ we have

$G(\underline{x}) = \mathcal{H}(\underline{w} \cdot \underline{x} + \theta)$.

As an example, consider the situation that $m = 2$. Let $\mathcal{S}$ be the subset of $\mathbb{R}^2$ consisting of the four vectors $(1,0), (0,1), (0,0), (1,1)$. Define G by $G(1,0) = 1$, $G(0,1) = 1$, $G(0,0) = 0$, and $G(1,1) = 1$ (G is the Boolean function 'OR'). The question is now: do there exist w_1, $w_2 \in \mathbb{R}$ and $\theta < 0$ such that the vector $(0,0)$ is seperated from the points $(1,0), (0,1)$ and $(1,1)$ by the hyperplane $w_1 x_1 + w_2 x_2 + \theta = 0$. Clearly one of the many choices is to take $w_1 = 1$, $w_2 = 1$ and $\theta = -\frac{1}{2}$. Hence we have found that the Boolean function G is representable by a Perceptron: $G(x_1, x_2) = \mathcal{H}(x_1 + x_2 - \frac{1}{2})$ for all $(x_1, x_2) \in \mathcal{S}$.

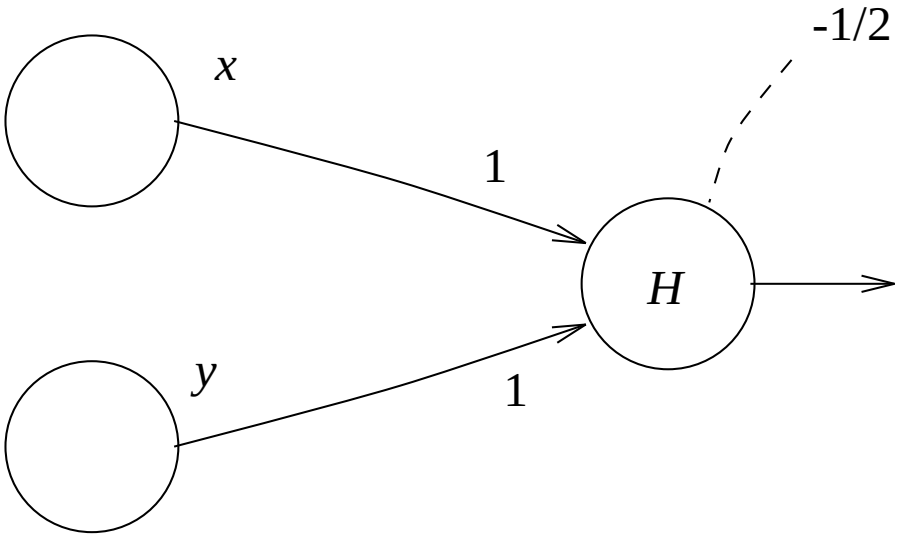

FIGURE 6. Perceptron representing G

It is clear that, in general, $G(\underline{x}) = \mathcal{H}(\underline{w} \cdot \underline{x} + \theta)$ for all $\underline{x} \in \mathcal{S}$ if and only if the points in $\mathcal{S}$ which satisfy $G(\underline{x}) = 1$ are separated from the points in $\mathcal{S}$ which satisfy $G(\underline{x}) = 0$ by the line $\underline{w} \cdot \underline{x} + \theta = 0$.

This observation motivates the following definition: a subset $\mathcal{S}$ of $\mathbb{R}^m$ is called *linearly separable with respect to* G if there exists $\underline{w} \in \mathbb{R}^m$ and $\theta \in \mathbb{R}$ such that for all $\underline{x} \in \mathcal{S}$ we have:

$$G(\underline{x}) = 1 \iff \underline{w} \cdot \underline{x} + \theta > 0,$$

$$G(\underline{x}) = 0 \iff \underline{w} \cdot \underline{x} + \theta \leq 0.$$

The following result then precisely characterizes the functions G that are representable by a Perceptron:

THEOREM 3.1. *Let $\mathcal{S}$ be a subset of $\mathbb{R}^m$ and let G be a function from $\mathcal{S}$ to $\{0,1\}$. There exists a perceptron with m input neurons that represents the function G if and only if $\mathcal{S}$ is linearly separable with respect to G.*

From this it is immediately clear that there exist very simple Boolean functions that can not be represented by a Perceptron: take for example $\mathcal{S} = \{(1,0),(0,1),(0,0),(1,1)\}$ and define a function G on $\mathcal{S}$ by $G(1,0) = 1$, $G(0,1) = 1$, $G(0,0) = 0$, and $G(1,1) = 0$. (This Boolean function is called the 'exclusive OR' function.) Clearly, the set $\mathcal{S}$ is not linearly separable with respect to the function G and consequently G cannot be represented by a Perceptron. The observation that there exist very simple Boolean functions that cannot be represented by a Perceptron was made in 1969 by M. MINSKY and S. PAPERT [16].

3.2. The Perceptron and learning

We will now consider the second central issue, that of learning. Suppose that, again, a subset $\mathcal{S}$ of $\mathbb{R}^m$ together with a function G from $\mathcal{S}$ to $\{0,1\}$ are given. The question we want to study here is: can we find some mechanism or algorithm that adapts the values of the weights $\underline{w}$ and threshold θ until the resulting network function $F_{(\underline{w},\theta)}$ is (approximately) equal to G?

The basic idea for such an algorithm could be as follows:

- Start with taking an arbitrary sequence $\{X_1, X_2, \ldots\}$ with $X_i \in \mathcal{S}$. Assume that we know the correct values of G in these points, i.e. we know $G(X_1), G(X_2), \ldots$.
- 'Present' these correct pairs $(X_i, G(X_i))$, $i = 1, 2 \ldots$ to the Perceptron architecture.
- On the basis of these 'learning examples' update the values of the weight vector $\underline{w}$ and the threshold θ.
- After having presented the perceptron a large number of correct examples, let the updated values of the weight vector and threshold be $\underline{w}^*$ and θ^*, respectively. Now hope that the network function $F_{(\underline{w}^*,\theta^*)}$ of the corresponding Perceptron is (approximately) equal to the given function G

Formalizing the above idea leads to the so-called *Perceptron learning algorithm* (given in 1959 by F. ROSENBLATT, [8]), which is defined inductively as follows: let ε be some fixed positive real number.

- At $t = 0$, choose arbitrary initial values $\underline{w}_0$ and θ_0.
- At time $t = n+1$ present the input vector X_{n+1}. Now update the current values $\underline{w}_n$ and θ_n according to the following rule:
  - If $\mathcal{H}(\underline{w}_n X_{n+1} + \theta_n) = G(X_{n+1})$ then take $\underline{w}_{n+1} = \underline{w}_n$ and $\theta_{n+1} = \theta_n$.
  - If $\mathcal{H}(\underline{w}_n X_{n+1} + \theta_n) = 0$ but $G(X_{n+1}) = 1$, then take $\underline{w}_{n+1} = \underline{w}_n + \varepsilon X_{n+1}$ and $\theta_{n+1} = \theta_n + \varepsilon$.
  - If $\mathcal{H}(\underline{w}_n X_{n+1} + \theta_n) = 1$ but $G(X_{n+1}) = 0$, then take $\underline{w}_{n+1} = \underline{w}_n - \varepsilon X_{n+1}$ and $\theta_{n+1} = \theta_n - \varepsilon$.

The rationale behind this updating rule is of course that if at step $n+1$ the network corresponding to the parameter values $\underline{w}_n$ and θ_n happens to give the correct functional value $G(X_{n+1})$, then there is no reason to change the current value of the weights. If on the other hand, for example, $\mathcal{H}(\underline{w}_n X_{n+1} + \theta_n) = 0$ but $G(X_{n+1}) = 1$, then the updating rule $\underline{w}_{n+1} = \underline{w}_n + \varepsilon X_{n+1}$ and $\theta_{n+1} = \theta_n + \varepsilon$ yields

$$\underline{w}_{n+1} X_{n+1} + \theta_{n+1} = (\underline{w}_n X_{n+1} + \theta_n) + \varepsilon \|X_{n+1}\|^2 + \varepsilon.$$

Now, the first term on the right in the above equation, $\underline{w}_n X_{n+1} + \theta_n$, is less than or equal to 0, and exactly this fact caused the network to give the wrong value 0. The updating rule at least yields

$$\underline{w}_{n+1} X_{n+1} + \theta_{n+1} \geq \underline{w}_n X_{n+1} + \theta_n$$

This means that if in the next step of the algorithm the same vector X_{n+1} would be presented, then the network associated with parameter values $\underline{w}_{n+1}$ and θ_{n+1} would be 'closer to' giving the correct value. In a sense this means that the parameter values are 'pushed in the right direction' while presenting the examples. One could think of this as a process of learning on the basis of examples. The number $\varepsilon > 0$ is called the *learning rate* of the algorithm. It turns out that under certain assumptions on $\mathcal{S}$ and G the parameter sequences $\{\underline{w}_n\}$ and $\{\theta_n\}$ converge:

THEOREM 3.2. *Let $\mathcal{S}$ be a finite subset of $\mathbb{R}^m$ and G a function from $\mathcal{S}$ to $\{0,1\}$. Assume that $\mathcal{S}$ is linearly separable with respect to G. Let $\varepsilon > 0$. Then for each sequence $\{X_n\}$ in $\mathcal{S}$ and for all initial values $\underline{w}_0$, θ_0, there exists an integer N_0 such that for all $n \geq N_0$ we have $\underline{w}_n = \underline{w}_{N_0}$ and $\theta_n = \theta_{N_0}$. Define $\underline{w}^* := w_{N_0}$ and $\theta^* = \theta_{N_0}$. Then for all $n \geq N_0$ we have $\mathcal{H}(\underline{w}^* X_n + \theta^*) = G(X_n)$.*

The above theorem is called *the Perceptron convergence theorem.* The first statement of the theorem says that the parameter sequences $\{\underline{w}_n\}$ and $\{\theta_n\}$ become stationary after finitely many steps, say $\underline{w}^*$ and θ^*, respectively. The second statement says that the network function corresponding to these particular values will take the correct functional values in the remaining terms of the sequence of examples. We note that this result does *not* say that the stationary values $\underline{w}^*$ and θ^* yield the correct network function on the entire set $\mathcal{S}$: only in the remaining terms of the sequence of examples the correct values are attained. Of course, an extreme case is to take a sequence consisting of the constant term $X_n = \underline{x} \in \mathcal{S}$. It is indeed reasonable that the network function will learn the correct value $G(\underline{x})$ by presenting the example $(\underline{x}, G(\underline{x}))$ over and over again, but it can not be expected that the network will learn anything about the other points in $\mathcal{S}$. On the other hand, the theorem *does* say that by choosing a suitable sequence $\{X_n\}$ it is possible to find values of $\underline{w}^*$ and θ^* that yield the correct network function on the entire set $\mathcal{S}$. Indeed, assume that $\mathcal{S} = \{x_1, x_2, \ldots, x_r\}$ and take the sequence

$$\{X_n\} = (x_1, x_2, \ldots, x_r, x_1, x_2, \ldots, x_r, x_1, x_2, \ldots, x_r, \ldots).$$

This shows that the Perceptron architecture is capable of learning the correct values of the given function by presenting it the complete set of correct function values.

4. Layered feedforward networks

The Perceptron is a feedforward network whithout hidden neurons. The output neuron has transfer function $\mathcal{H}$. We saw in the previous section that in connection with the representation of functions by perceptrons, we have the restriction of linear separability. In this section we will admit hidden neurons in the network. We will restrict ourselves here to feedforward networks in which the hidden neurons are grouped into what we will call *layers*. Suppose we have a feedforward network with m input neurons and p output neurons. The network is called *layered* if all paths from sources to sincs have the same length, say ℓ. In that case we say that the network has $h := \ell - 1$ hidden layers.

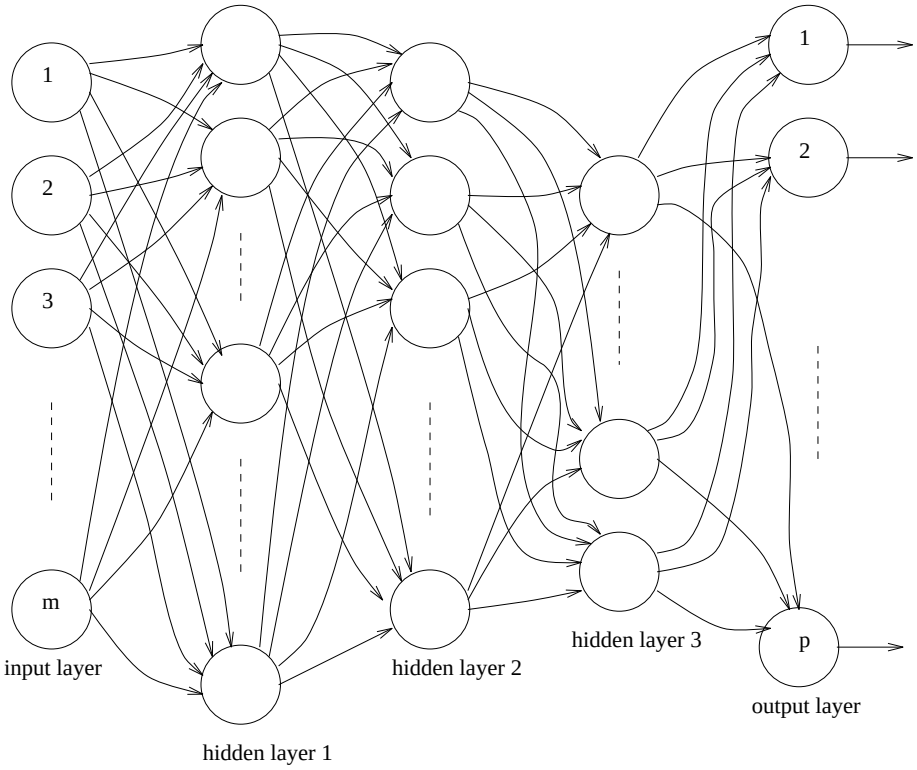

FIGURE 7. Layered feedforward network with 3 hidden layers

We will also admit more general transfer functions in the network. The hidden neurons will all be assumed to have the same transfer function, say σ, where σ is an arbitrary function from $\mathbb{R}$ to $\mathbb{R}$. Sometimes we will assume that the output neurons also have this transfer function σ. Depending on the

context, sometimes the output neurons will have the transfer function $\mathcal{H}$ or the transfer function $f(x) = x$.

It turns out that admitting hidden neurons extends the capability of representing functions by feedforward networks. Recall from the previous section that the 'exclusive OR' function could not be represented by a Perceptron. It turns out that if, in addition to the two input neurons and the output neuron, we admit one layer consisting of two hidden neurons, then there exist a choice of weights and thresholds such that the corresponding network function is equal to the 'exclusive OR' function. Indeed, if for the transfer function of the hidden neurons and the output neuron we take $\mathcal{H}$, then the network function $F(x_1, x_2)$ is given by

$$F(x_1, x_2) = \mathcal{H}(w_1 s_1 + w_2 s_2 + \theta),$$

with s_1 and s_2 given by

$$s_1 = \mathcal{H}(v_{11} x_1 + v_{12} x_2 + \eta_1),$$

$$s_2 = \mathcal{H}(v_{21} x_1 + v_{22} x_2 + \eta_2).$$

It is easily seen that if we take $\theta = \eta_1 = \eta_2 = 0$, $v_{11} = v_{22} = w_1 = w_2 = 1$, and $v_{12} = v_{21} = -1$, then the corresponding network function equals the 'exclusive OR' function.

This example illustrates how we can get around the requirement of linear seperability by adding a layer of hidden neurons. Of course, there still remains the issue of learning: is it possible, in the context of more complex, multi-layered, network architectures, to develop learning algorithms that lead to network functions that are (approximately) equal to an a priori given function? We will come back to this when we discus the so-called 'Back Propagation Algorithm'.

4.1. Representation and approximation of functions by feedforward networks

In this subsection we will discuss the issue of representation or approximation of a given function by a multi-layered feedforward network. The question we want to study is the following: suppose we have a given function $G : \mathbb{R}^m \to \mathbb{R}^p$, does there exist a feedforward network achitecture such that for a suitable choice of weights and thresholds the corresponding network function is (approximately) equal to G? Of course, the network architectures we would be looking for have m input neurons and p output neurons. However, in connection with the problem stated, we could ask: if a certain function G can be represented or approximated by a given feedforward network, then how many hidden layers, and how many neurons per hidden layer would be needed for this?

It will turn out that any given continuous function $G : \mathbb{R}^m \to \mathbb{R}^p$ can be approximated arbitrarily close (in a sense to be explained) by a feedforward network with *one* hidden layer. The number of hidden neurons, and the values of the weights and thresholds will depend on the desired degree of accuracy of approximation.

We will take a closer look at this for the case that $m = p = 1$. For a given function $\sigma : \mathbb{R} \to \mathbb{R}$, let $\mathcal{N}(n, \sigma)$ be the set of all functions from $\mathbb{R}$ to $\mathbb{R}$ that can be represented exactly by a feedforward net with one input neuron, one hidden layer of n neurons, all having tranfer function σ, and one output neuron having transfer function $f(x) = x$. Denote the weights associated with the connections between the input neuron and the hidden neurons by $v_1, \ldots, v_n$, and denote the thresholds of the hidden neurons by $\eta_1, \ldots, \eta_n$. Denote the weights of the connections between the hidden neurons and the output neuron by $w_1, \ldots, w_n$, and the threshold value of the output neuron by θ. It is then clear that $\mathcal{N}(n, \sigma)$ consist exactly of those functions $F : \mathbb{R} \to \mathbb{R}$ that can be written as

$$F(x) = \theta + \sum_{i=1}^{n} w_i \sigma(v_i x + \eta_i),$$

for certain v_i, η_i, w_i and θ. Let $\mathcal{N}(\sigma)$ be the union over n of all $\mathcal{N}(n, \sigma)$, i.e., the set of all functions $F : \mathbb{R} \to \mathbb{R}$ that can be represented exactly by a feedforward net with one hidden layer, with all hidden neurons having transfer function σ.

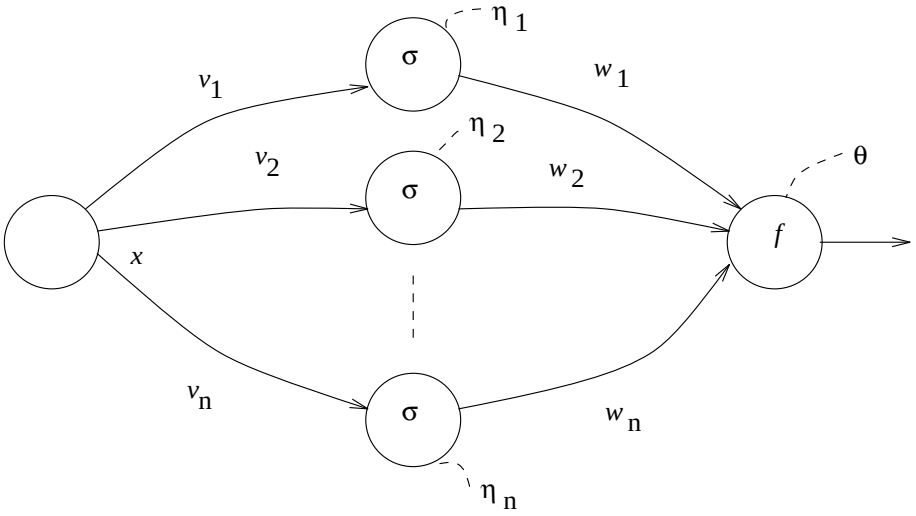

FIGURE 8. Network representing a typical element from $\mathcal{N}(n, \sigma)$

We want to study the question in what sense a given function $G : \mathbb{R} \to \mathbb{R}$ can be approximated by functions $F \in \mathcal{N}(\sigma)$. We will consider approximations uniformly on compact intervals.

Following [7], we will call a transfer function σ *a universal transfer function*, if every continuous function $G : \mathbb{R} \to \mathbb{R}$ can be approximated arbitrarily close, uniformly on any compact interval, by functions from $\mathcal{N}(\sigma)$. More concrete: σ will be called a universal transfer function if for every continuous function $G : \mathbb{R} \to \mathbb{R}$, for all $a, b \in \mathbb{R}$, and for each $\varepsilon > 0$, there exist an integer n, and

real numbers $v_1, \ldots, v_n, \eta_1, \ldots, \eta_n, w_1, \ldots, w_n, \theta$, such that the corresponding network function F satisfies

$$\sup_{x \in [a,b]} |F(x) - G(x)| < \varepsilon.$$

The problem now is to characterize the set of universal transfer functions.

Before we take a closer look at this problem we would like to compare it with the classical problem of approximating a given function on compact intervals by trigonometric polynomials. It is well-known that every continuous periodic function from $\mathbb{R}$ to $\mathbb{R}$ can be approximated arbitrarily close by trigonometric polynomials, uniformly on $\mathbb{R}$. From this it is easily seen that for every given continuous function function G, for all $a, b \in \mathbb{R}$, and every $\varepsilon > 0$, their exist real numbers $\alpha_0, \alpha_1, \ldots, \alpha_n$, $\beta_0, \ldots, \beta_n$, such that the trigonometric polynomial P given by

$$P(x) = \alpha_0 + \sum_{k=1}^{n} \alpha_k \sin(\frac{\pi}{b-a} kx + \beta_k)$$

satisfies

$$\sup_{x \in [a,b]} |P(x) - G(x)| < \varepsilon.$$

In neural network terminology, this can be restated by saying that the function $x \mapsto \sin x$ is a universal transfer function!

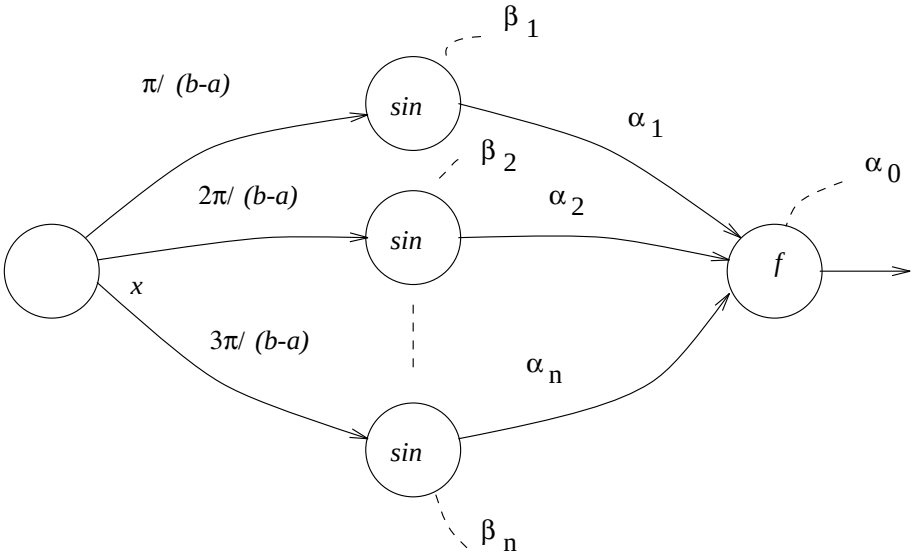

FIGURE 9. Representation of a trigonometric polynomial

From the latter point of view, we think that it is quite a fascinating problem to characterize *all* universal transfer functions. Before giving such characterization, we first note that it is quite easy to come up with functions that certainly are *not* universal: indeed, let $\sigma(x)$ be a polynomial in x, say of degree k. It is then obvious that any network function, i.e., any element of $\mathcal{N}(\sigma)$, is also a polynomial, of degree less than or equal to k. Since, needless to say, not all continuous functions can be approximated arbitrarily close by functions from a class of polynomials with a fixed upper bound to their degree, polynomial transfer functions fail to be universal.

This shows that for a function to be a universal transfer function it is necessary that it is not a polynomial. A beautiful recent result by LESHNO, LIN, PINKUS and SCHOCKEN [15], shows that this condition is also *sufficient*: a function σ is a universal transfer function if it is not equal to a polynomial almost everywhere.

THEOREM 4.1. *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be bounded on each closed interval $[a, b]$, and continuous almost everywhere. Then σ is a universal transfer function if and only if there does not exist a polynomial p such that $\sigma(x) = p(x)$ almost everywhere.*

We note that if a given function is bounded on a closed interval $[a, b]$, then it is continuous almost everywhere on that interval if and only if it is Riemann integrable over that interval (see RUDIN [19], Theorem 11.33). Hence, in the above theorem, the condition 'continuous almost everywhere' can be replaced by 'Riemann integrable over all closed intervals $[a, b]$'.

Examples of universal transfer functions are of course manifold. Of interest in the context of neural networks is the fact that the commonly used sigmoid functions

$$\sigma(x) = \frac{1}{e^{-\beta x} + 1},$$

and

$$\sigma(x) = \tanh(\beta x)$$

are universal transfer functions. Also the Heaviside function is universal. We would like to stress here that, if we compare this result with the approximation result using trigonometric polynomials, then there is one fundamental difference: the coefficients α_k and β_k in the trigonometric polynomial approximation can be calculated explicitly in terms of Fourier coefficients (one could take for P the Cesàro mean), whereas the approximation theorem for neural networks is only *an existence result*. The theorem only states that suitable weights and thresholds exist, but does not give general formulas to calculate these real numbers.

In the context of neural networks, the issue is rather to find schemes, algorithms, or mechanisms to *learn* the appropriate values of the weights and

thresholds by presenting the network examples of values that the function to be approximated takes in certain points.

The problem of approximating a given function by neural networks has been the subject of a large amount of research activity in the field of neural nets in the past six years. Related questions can already be found in the work of KOLMOGOROV [1]. Important contributions can also be found in the work of HECHT-NIELSEN, [18]. In 1989, HORNIK [12] showed that every non-constant, bounded, and continuous function σ is a universal transfer function. Of course, the latter now follows from the more recent theorem stated above. Among other relevant references, we mention CYBENKO [9] and FUNAHASHI [13].

We note that the result by Leshno, Lin, Pinkus and Schocken also holds for continuous functions G from $\mathbb{R}^m$ to $\mathbb{R}^p$. In the definition of universal transfer function and in the statement of the theorem, the interval $[a, b]$ should then be replaced by an arbitrary compact set $\mathcal{K} \subset \mathbb{R}^m$.

It is expected that if, for a certain compact set $\mathcal{K} \subset \mathbb{R}^m$, we require the accuracy of appproximation to increase (i.e, if we let ε become smaller and smaller), then the number of neurons in the hidden layer should increase. On the question how exactly the number of neurons depends on the accuracy of approximation, we mention recent results by BARRON, [2, 1], and by JONES [14]. They proved the following remarkable result for neural networks with sigmoid tansfer function: For a given compact subset $\mathcal{K} \subset \mathbb{R}^m$, a sufficiently smooth target function $G : \mathcal{K} \to \mathbb{R}$ can be approximated in L_2-sense by a neural network with 1 hidden layer containing n neurons at a rate $O(\frac{1}{\sqrt{n}})$.

If, instead of general functions from $\mathbb{R}^m$ to $\mathbb{R}^p$, we restrict ourselves to Boolean functions, i.e., functions from $\{0,1\}^m$ to $\{0,1\}^p$, then the situation is somewhat clearer. It was shown in 1987 by DENKER, SCHWARZ, WITTNER, SOLLA, HOWARD, JACKEL and HOPFIELD [11] (see also [4], page 55), that every function $G : \{0,1\}^m \to \{0,1\}^p$ can be *exactly* represented by a feedforward network with one hidden layer consisting of $p2^m$ neurons, provided the hidden neurons and output neurons all have transfer function $\mathcal{H}$. Of course, the number $p2^m$ can in certain situations be very conservative: we showed that the 'exclusive OR' can be represented using only 2 hidden neurons.

4.2. Learning in general feedforward networks

In the previous subsection we saw that if σ is a universal transfer function, then every continuous function $G : \mathbb{R}^m \to \mathbb{R}^p$ can, on any compact set, be approximated arbitrarily close by a feedforward net with one hidden layer, where the hidden neurons have transfer function σ. As noted, this result is basically an existence result and the question remains: how should we determine, for a given compact set $\mathcal{K}$ and a certain accuracy $\varepsilon > 0$, the *number* of hidden neurons and *suitable values* for the weights and thresholds. In neural nets one would like to obtain suitable values by some kind of learning algorithm. In this subsection we will discuss a basic learning algorithm for feedforward nets, called the *Back Propagation Algorithm*.

For simplicity, we assume that $p = m = 1$. Suppose that σ is a universal

transfer function, and suppose we have a continuous function G from $\mathbb{R}$ to $\mathbb{R}$. Let $a, b \in \mathbb{R}$, and let $\varepsilon > 0$. The problem is how to determine an integer n, vectors $\underline{v} = (v_1, \ldots, v_n)$, $\underline{\eta} = (\eta_1, \ldots, \eta_n)$, $\underline{w} = (w_1, \ldots, w_n)$, and a real number θ such that

$$\sup_{x \in [a,b]} |G(x) - F_{\underline{v},\underline{\eta},\underline{w},\theta}(x)| < \varepsilon, \tag{1}$$

where the network function corresponding to the particular values for the weights and thresholds is given by

$$F_{\underline{v},\underline{\eta},\underline{w},\theta}(x) = \theta + \sum_{i=1}^{n} w_i \sigma(v_i x + \eta_i)$$

The idea is to take a fixed network architecture (i.e. to fix the number of hidden neurons n) and then try to obtain suitable values for the weights and thresholds by presenting the network architecture a number of learning examples. More concrete, one chooses $x_1, \ldots, x_q \in \mathbb{R}$ and 'shows' the network the examples $(x_1, G(x_1)), \ldots, (x_q, G(x_q))$ in the following sense: starting with arbitrary values for the weights and thresholds $\underline{v}, \underline{\eta}, \underline{w}, \theta$ one calculates the values $F_{\underline{v},\underline{\eta},\underline{w},\theta}(x_1), \ldots, F_{\underline{v},\underline{\eta},\underline{w},\theta}(x_q)$ the network generates in the points $x_1, \ldots, x_q$. One compares these values with the values $G(x_1), \ldots, G(x_q)$ the network *should have generated*. Then, on the basis of the error that occurs, the values for the weights and thresholds are updated. Next, the experiment is repeated with these updated values. One hopes that after a sufficiently large number of repetitions of this experiment the values of the weights and thresholds are such that (1) holds.

To be more specific, for given values of the weights and thresholds, the network makes a quadratic error

$$E(\underline{v}, \underline{\eta}, \underline{w}, \theta) := \frac{1}{2} \sum_{i=1}^{q} (F_{\underline{v},\underline{\eta},\underline{w},\theta}(x_\mu) - G(x_\mu))^2,$$

We stress that the error only depends on the values of the weights and thresholds. In fact, the error is a function from $\mathbb{R}^{3n+1}$ to $\mathbb{R}^+$. In order to obtain suitable values for the weights and thresholds, it is not unreasonable to *minimize* the error function $E(\cdot)$. Now, the idea is to have the updating of the weights and thresholds based on minimizing the error function *iteratively*. If for the iterative method we use ideas from the method of *steepest descent*, we arrive at the celebrated Back Propagation Algorithm.

Recall that if $f : \mathbb{R}^N \to \mathbb{R}$ is a differentiable function, then for a given $x \in \mathbb{R}^N$ the direction in $\mathbb{R}^N$ along which the function decreases most rapidly is given by $-\nabla f(x)$, where ∇f denotes the gradient of f. The method of steepest descent is an iterative method that is aimed at finding $x^* \in \mathbb{R}^N$ in which the function f attains a minimum. Starting with an initial guess x_0 of x^*, a sequence $x_0, x_1, x_2, \ldots$ is defined iteratively by

$$x_{k+1} = x_k - s_k \nabla f(x_k),$$

where $s_k > 0$ is choosen to minimize the function $\phi_k(s) := f(x_k - s_k \nabla f(x_k))$. This leads to a sequence $\{x_k\}$ that would then, ideally, converge to a minimizing x^*.

In neural nets, a rudimentary version of this iterative algorithm is used to tackle the problem of iteratively minimizing the error function $E(\cdot)$. Instead of performing, at each step k, minimization of the function $\phi_k(s)$ to obtain s_k, one simply fixes a small positive real number $\varepsilon > 0$ and defines a sequence $\{x_k\}$ by

$$x_{k+1} = x_k - \varepsilon \nabla f(x_k).$$

It is then hoped that this sequence leads to a minimizing point x^*.

We will now explain how these ideas lead to a learning algorithm. In order to simplify notation, denote

$$\underline{p} := (\underline{v}, \underline{\eta}, \underline{w}, \theta)$$

The value of $\underline{p}$ at iteration step k is denoted by $\underline{p}(k)$. The vector $\underline{p}(k)$ has components $v_i(k), \eta_i(k), w_i(k)$ and $\theta(k)$. Now, fix the values $x_1, \ldots, x_q \in \mathbb{R}$ (this set of fixed numbers is called the *batch*). Denote $G_i := G(x_i),\ i = 1, \ldots, q$.

For the sake of exposition, in the remainder of this subsection we will take a particular transfer function, the sigmoid transfer function

$$\sigma(x) = \frac{1}{e^{-\beta x} + 1}.$$

In the sequel, we will use the fact that, if $\beta = 1, \sigma$ satisfies the differential equation

$$\sigma' = \sigma(1 - \sigma). \tag{2}$$

Suppose that, at iteration step k, the current values of the weights and thresholds are given by $\underline{p}(k)$. At this moment, the samples $x_1, \ldots, x_q$, are presented to the network. In response to the input value x_i, the following signals occur at the output branches of the hidden neurons and the output neuron:

- the hidden neuron j generates the output value $s_{ij}(k) = \sigma(x_i v_j(k) + \eta_j(k))$,
- the output neuron generates the output value $y_i(k) = F_{\underline{p}(k)}(x_i)$.

Let us assume that, in some way, during this experiment, we make a record of these output values $s_{ij}(k)$ and $y_i(k)$ $(j = 1, \ldots, n,\ i = 1, \ldots, q)$.

Now, the updating of $\underline{p}(k)$ is done according to $\underline{p}(k+1) = \underline{p}(k) - \varepsilon \nabla E(\underline{p}(k))$, so we should in some way try to calculate the value of $\nabla E(\underline{p}(k))$. Clearly,

$$\nabla E = \left(\frac{\partial E}{\partial \underline{v}}, \frac{\partial E}{\partial \underline{\eta}}, \frac{\partial E}{\partial \underline{w}}, \frac{\partial E}{\partial \theta}\right),$$

so in order to update the weights and thresholds $\underline{w}(k)$ and $\theta(k)$ of the output neuron we should calculate $\frac{\partial E}{\partial \underline{w}}(\underline{p}(k))$ and $\frac{\partial E}{\partial \theta}(\underline{p}(k))$. It turns out that these vectors of partial derivatives can be calculated explicitly, in terms of the recorded output values. Indeed, it is straightforward to verify that

$$\frac{\partial E}{\partial w_\ell}(\underline{p}(k)) = \sum_{i=1}^{q}(y_i(k) - G_i)s_{i\ell}(k),$$

and

$$\frac{\partial E}{\partial \theta}(\underline{p}(k)) = \sum_{i=1}^{q}(y_i(k) - G_i).$$

If we introduce the *error of the output neuron* at step k corresponding to the sample x_i by

$$\Delta_i(k) := (y_i(k) - G_i),$$

then the updating rules for $\underline{w}(k)$ and $\theta(k)$ can be written as

$$w_\ell(k+1) = w_\ell(k) - \varepsilon \sum_{i=1}^{q} \Delta_i(k)s_{i\ell}(k),$$

$$\theta(k+1) = \theta(k) - \varepsilon \sum_{i=1}^{q} \Delta_i(k).$$

To find the updating rules for the weigths $\underline{v}(k)$ and thresholds $\underline{\eta}(k)$ of the hidden neurons, we should calculate $\frac{\partial E}{\partial \underline{v}}(\underline{p}(k))$ and $\frac{\partial E}{\partial \underline{\eta}}(\underline{p}(k))$. We calculate:

$$\frac{\partial E}{\partial v_\ell}(\underline{p}(k)) = \sum_{i=1}^{q}(y_i(k) - G_i)w_\ell(k)s_{i\ell}(k)(1 - s_{i\ell}(k))x_i,$$

and

$$\frac{\partial E}{\partial \eta_\ell}(\underline{p}(k)) = \sum_{i=1}^{q}(y_i(k) - G_i)w_\ell(k)s_{i\ell}(k)(1 - s_{i\ell}(k)).$$

Here, we used the fact that σ satisfies the differential equation (2). Introduce the notation

$$\tilde{\Delta}_{i\ell}(k) := (y_i(k) - G_i)w_\ell(k)s_{i\ell}(k)(1 - s_{i\ell}(k)).$$

Then the updating rules for $\underline{v}(k)$ and $\underline{\eta}(k)$ can be written as

$$v_\ell(k+1) = v_\ell(k) - \varepsilon \sum_{i=1}^{q} \tilde{\Delta}_{i\ell}(k)x_i,$$

$$\eta_\ell(k+1) = \eta_\ell(k) - \varepsilon \sum_{i=1}^{q} \Delta_{i\ell}(k).$$

Note the similarity in structure between the updating rules for the output neuron and the hidden neurons.

The most striking feature of these updating rules is that one can interpret the updating to take place in two separate stages, in the following sense. One should first note that the $\tilde{\Delta}_{i\ell}(k)$'s can be calculated from $\Delta_i(k)$ by the following formula:

$$\tilde{\Delta}_{i\ell}(k) = \Delta_i(k) w_\ell(k) s_{i\ell}(k)(1 - s_{i\ell}(k)). \tag{3}$$

Thus one could consider the updating of the weights and threshold of the output neuron (which only uses $\Delta_i(k)$) as the first stage of the updating procedure. In the second stage of the updating procedure, one first calculates the $\tilde{\Delta}_{i\ell}(k)$'s from $\Delta_i(k)$, and then updates the weights and thresholds of the hidden neurons using these numbers. One could consider $\tilde{\Delta}_{i\ell}(k)$ as a kind of error, the *error of the ℓ-th hidden neuron at step k corresponding to the sample x_i*. Formula (3) can then be interpreted as a formula that calculates the errors of the hidden neurons using the error of the output neuron. In this sense, the error is *propagated backwards through the network*, starting at the output neuron. This structure of the updating algorithm explains the terminology *Back Propagation Algorithm*.

The above describes the kth iteration step. At each iteration step, the same batch $x_1, \ldots, x_q$ is used. In principle, the algorithm stops at stage N, if N is such that

$$E(\underline{v}(N), \underline{\eta}(N), \underline{w}(N), \theta(N)) < \gamma,$$

where γ is some a priori given tolerance.

In the above, for the sake of exposition we have restricted ourselves to the case that $p = m = 1$, and that we have only *one* hidden layer. In the general case the ideas remain the same. If the network has h hidden layers, then each iteration step is subdivided into $h + 1$ stages. Counting layers from the right to the left, each stage corresponds to a calculation of the errors in a layer in terms of the errors in the previous layer ('back propagation of errors'), and an updating of the weigths and thresholds in the layer.

The Back Propagation Algorithm was discovered in 1974 by PAUL J. WERBOS [17]. After being ignored for over two decades, it was rediscovered independently in 1985 by DAVID E. RUMELHART [6] and DAVID B. PARKER [5]. The algorithm plays an important role in artificial neural networks. Together with its variations based on more advanced iterative minimization algorithms (like e.g. the conjugate gradient method), it provides a reasonable training method for multi-layer feedforward networks. Because of its simple structure, the algorithm can be easily implemented on electronic computers. For a more general treatment of the Back Propagation Algorithm we refer to [4], [10], or [3].

References

1. A.K. Kolmogorov (1957). On the representation of continuous functions of several variables by superposition of continuous functions of one variable and addition. *Dokl. Akad. Nauk SSSR*, **114**, page 953.
2. A.R. Barron (1991). Universal approximation bounds for superpositions of a sigmoidal function. preprint, Dept. of Statistics, University of Illinois, Urbana, USA.
3. B.J.A. Kröse and P.P. van der Smagt (1991). *An Introduction to Neural Networks.* The University of Amsterdam.
4. B. Müller and J. Reinhardt 1990. *Neural Networks, an Introduction.* Springer Verlag.
5. D.B. Parker (1985). Learning-logic: casting the cortex of the human brain in silicon. MIT Techn. Report TR-47.
6. D.E. Rumelhart, G.E. Hinton and R.J. Williams (1986). Learning representations by back-propagating errors. *Nature* **323**, pages 533–536.
7. E.D. Sontag (1993). Neural networks for control. In H.L. Trentelman and J.C. Willems, editors, *Essays on Control: Perspectives in the Theory and its Applications*, pages 339–380. Birkhäuser, Boston.
8. F. Rosenblatt (1959). *Principles of Neurodynamics.* Spartan Books, New York.
9. G. Cybenko (1989). Approximation by superpositions of a sigmoidal function. *Math. Control, Signals, and Systems* **2**, pages 303–314.
10. J. Hertz, A. Krogh and R.G. Palmer (1991). *Introduction to the Theory of Neural Computation.* Addison-Wesley, Redwood City.
11. J.S. Denker, D. Schwartz, B. Wittner, S. Solla, R. Howard, L. Jackel and J. Hopfield (1987). In *Complex Systems* **1**, page 877.
12. K. Hornik (1991). Aproximation capabilities of multilayer feedforwrd networks. *Neural Networks* **4**, pages 251–257.
13. K.I. Funahashi (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks* **2**, pages 183–192.
14. L.K. Jones (1990). Constructive approximation for neural networks by sigmoidal functions. In *Proceedings of the IEEE 78*, pages 1586–1589.
15. M. Leshno, V. Ya. Lin, A. Pinkus and S. Schocken (1993). Multilayer feedforward networks with a non-polynomial activation function can approximate any function. To appear in: *Neural Networks.*
16. M. Minsky and S. Papert (1969). *Perceptrons: An Introduction to Computational Geometry.* The MIT Press.
17. P.J. Werbos (1974). Beyond regression: new tools for prediction and analysis in the behavioral sciences. Master's thesis, Harvard University.
18. R. Hecht-Nielsen (1989). Theory of backpropagation neural networks. In *Proceedings of the Int. Joint Conf. on Neural Networks, San Diego: SOS Printings*, pages 593–606.
19. W. Rudin (1976). *Principles of Mathematical Analysis.* McGraw-Hill, New York.
20. W.S. McCullogh and W. Pitts (1943). A logical calculus of the ideas

immanent in nervous activity. *Bulletin of Mathematical Biophysics* **5**, pages 115–133.