Appl. Math. and Comp. Sci., 1992, vol.2, No.2, 251–276

SELF–PROGRAMMING OF NEURAL NETWORKS FOR INDEXING AND FUNCTION APPROXIMATION TASKS

M. B. ZAREMBA*, E. PORADA*

The problem of non–iterative mapping of stimulus–response associations directly onto an association matrix is addressed in this paper. The procedures presented are designed for the calculation of connection weights in multilayer, feedforward neural networks. Two distinct situations from the standpoint of object separability are analyzed in more detail. The first can be defined in the context of content addressable memories, where the network performs essentially indexing or classification tasks. The properties of the self–programming procedure are illustrated using a character recognition network as an example with the intermediate layer of dipoles incorporated into the network architecture. The second situation relates to cases where the approximation capabilities of a single output network are of principal importance.

1. Introduction

Applications requiring real–time image recognition, pattern classification, and other tasks of a similar nature that may involve a heavy computational burden make parallel distributed processing and its implementations on neural networks important. The networks, which are multilayer structures of a large number of heavily interconnected simple processors, have recently been the subject of considerable research interest. Although the first papers in the area of artificial neural networks date back several decades (Hebb, 1949; Rosenblatt, 1959), the discovery of powerful new learning algorithms (Rumelhart *et al.*, 1986) and the advances in analog VLSI (Hwang and Kung, 1989) and optical technologies (Kyuma *et al.*, 1988) have prompted a renewed interest in this exciting field, and have provided a basis for the rapid development of neural network engineering.

There are several advantages of neuromorphic architectures that are important from the practical applications standpoint. Inherently distributed processing gives neural networks a fault tolerant behavior, generally referred to as graceful degradation. This means that the system can operate successfully even if some of its components are damaged. The ability of neural networks to learn provides an interesting alternative to statistical methods of classification. Furthermore, no assumptions about the probabilistic model need to be made. Neural net classifiers

* Département d'informatique, Université du Québec à Hull, Hull, Québec, Canada J8X 3X7

are able to calculate higher–order decision boundaries under realistic constraints, while not being subject to combinatorial explosion or excessive requirements regarding the storage of learning samples (Barnard and Casasent, 1989; Gorman and Sejnowski, 1988).

Presently, feedforward perceptron–type neural architectures using the backpropagation training algorithm are being applied in the vast majority of practical situations. The backpropagation training scheme provides a powerful, general tool for the determination of connection weights. Its generality, however, is obtained at the expense of training speed, occasional poor convergence, and a limit to the number of training inputs. In a large class of problems the backpropagation training algorithm is unacceptable. Other training techniques for optimizing criterion functions have been proposed that involve an alternative to the steepest descent method used in backpropagation. These include the variable metric (Dennis *et al.*, 1983), the conjugate gradient (Fitch *et al.*, 1991), and the stochastic method (Barnard, 1992). The Cerebellar Model Arithmetic Computer (CMAC) with a built–in associative mapping that assures local generalization (Miller *et al.*, 1990) allows for orders of magnitude faster learning than the backpropagation algorithm.

A new Boolean–like training algorithm to be used on a four–layer perceptron–type feedforward neural network for the generation of binary–to–binary mappings is presented by Gray and Michel (1992). The applicability of this algorithm extends to some problems that are of interest to us in this paper. This paper also focuses on the assignment of weights to the network connections in an algorithmic way, with non–iterative presentation of training inputs. The self–programming method discussed here can be used, however, in a much broader class of applications, ranging from simple binary indexing problems to the approximation of parametric functions.

The indexing problem, i.e. access to a memory location containing an object index by a description of the object, or by its presentation to the neural memory, rather than by addressing, plays a crucial role in object recognition. The memory, upon presentation of an object, generates the address (or index) of the object by associating the desired response to the stimulus via a matrix operator (Kohonen, 1988). The computation of the matrix operator, being a type of regression problem (Poelzleitner and Wechsler, 1990), is the key issue. In order to derive the operator, we use procedures based on the method of matrix inversion by partitioning. There are two types of self–programming algorithms presented in this paper. In the first, the indexing of the objects is based on the identification of those elements of objects' internal representations that differentiate one object from all others. In this case, the indexing does not depend on the order of presentation of the training sequence. The second algorithm takes into account the elements that differentiate a given object from previously learned ones. The sequence of the presentation of the training objects now has an impact on the performance of the indexing process.

An extension of the discrete indexing problem into the real–value domain leads us to the function approximation problem. It has been proven that any continuous mapping can be approximately realized by multilayer feedforward neural networks with at least one hidden layer of neurons whose transfer functions are sigmoidal (Funahashi, 1989; Hornik *et al.*, 1989) or non–sigmoidal (Stinchcombe and White, 1989) functions. Theoretical justification for the use of multilayer feedforward networks in operations requiring simultaneous approximation of a function and its derivatives is given by Hornik *et al.* (1990). The problem of efficient learning of such function approximation networks is still open to research. A form of self–programming developed by us allows for an approximation of a parametric function such that the values of the function at the training points correspond precisely to the required values. Measurement systems are typical examples of an application where this characteristic of the network may be of importance (Zaremba *et al.*, 1991). The network output will be equal to the required calibrated values of the measurand at the calibration points.

The structure of this paper is as follows. In Section 2 we deal with the indexing problem. After a short discussion of the separability of an object's internal representations, the self–programming procedures are presented for the strong and weak seperability cases. Their features are illustrated using a character recognition example. The application of self–programming to parametric function approximation is discussed in Section 3. An example including an optimal approximation procedure is given.

2. The Indexing Function for Binary Images

In this section, we will consider binary images defined directly as subsets of activated neurons in a set N of neurons; N is supposed to be one network layer. In this way, an image x is a binary vector (an element of $\mathbb{R}^{\mathrm{Card(N)}}$, where Card(N) denotes the cardinality of N), but it is also a subset of N. For $\mathrm{n} \in \mathrm{N}$, we write interchangeably $x_{\mathrm{n}} = 1$ or $\mathrm{n} \in x$.

The task of binary image recognition, regarded as an indexing problem, can be expressed by

$$F(x^{(k)}) = \varepsilon_k, \qquad k = 1, 2, ..., K, \tag{1}$$

where F denotes the input–to–output function of the neural processor, $\{x^{(k)}\}$ is a family of binary images, ε_k is an impulse signal in the k–th output neuron, for each k.

We identify the output layer with the index set

$$\mathrm{K} = \{1, 2, ..., K\}.$$

Thus $\varepsilon_k = \alpha_k[\delta_{ki}]_{i=1,\ldots,K}$, where $\alpha_k > 0$ and δ_{ki} is Kronecker delta. Now we can say that the desired network performance consists of activating the index corresponding to the input, and only that index.

One layer of weighted connections between the input layer and the layer often fails to index a family of inputs properly, particularly when the inputs are overlaping one another's images. However, two layers of connections are able to deal with any indexing task. A proof of the above statment is the topic of this section, but the most important results concern the self-programmability of a network undertaking an image recognition task. The notion of self-programmability will be defined later. Here, let us introduce some more notations and explain how the processing task will be distributed among the different layers.

The input, hidden, and output layers of neurons are denoted by I, D, and K, respectively. The notation for the hidden layer comes from the *dipoles* that we define futher in this section. The transfer functions in I and K are the identity functions. The transfer function in D is the function

$$\phi(x) = \begin{cases} 0 & \text{if} \quad x \leq 0 \\ x & \text{if} \quad 0 < x < 1 \\ 1 & \text{if} \quad x \geq 1 \end{cases} \tag{2}$$

which becomes the hard delimiter (0/1) in the case of an integer variable x.

For a given input x, the output from D is a binary vector $y \in \mathbb{R}^{\mathrm{Card(D)}}$ called the internal representation of x. Thus, the input-to-output processing is the following composition of functions:

$$x \rightarrow y \rightarrow z = yw^* = F(x),$$

where w^* is the connectivity matrix in the layer $\mathrm{D} \times \mathrm{K}$ of connections.

The network indexes $\{x^{(k)}\}$ if, and only if, its second layer of connections indexes $\{y^{(k)}\}$, i.e., if

$$y^{(k)}w^* = \varepsilon_k, \qquad k = 1, 2, ..., K \tag{3}$$

A matrix w^* satisfying (3) exists if the internal representations are linearly separable binary vectors. The task of producing separable representations of the inputs $x^{(k)}$ will be assigned to the first layer $\mathrm{I} \times \mathrm{D}$ of connections.

2.1. Separable Internal Representations

To produce separable internal representations of the inputs $x^{(1)}, ..., x^{(K)}$, we will apply a method called dipole processing of binary images. The method gives a separability which is slightly stronger than the linear separability. In this paragraph we define that separability of the family $\{y^{(k)}\}$ and, assuming the separability, we construct the matrix w^* satisfying (3).

Definition 1. Let

$$\overline{y}^{(k)} = y^{(k)} \setminus \left(y^{(1)} \cup ... \cup y^{(k-1)}\right) = \left\{\mathrm{d} \in y^{(k)} : \mathrm{d} \notin y^{(l)} \quad \text{for} \quad l < k\right\} \tag{4}$$

The sequence $y^{(1)}, ..., y^{(K)}$ is *separable* if $\overline{y}^{(k)} \neq \emptyset$ for all k. The *family* $\{y^{(k)}\}$ is *separable* if it can be arranged in a separable sequence. The family is *strongly separable* if each arrangement in a sequence is a separable sequence.

We pay attention to the strong separability, because many recognition problems can be reduced to problems of indexing strongly separable representations. The indexers preserve the generalization ability that characterizes the neural processing systems (the examples in paragraph 2.3. illustrate the generalization functions of indexers).

2.1.1. The Case of Strong Separability

The family $\{y^{(k)}\}$ is strongly separable if, and only if, the sets

$$\tilde{y}^{(k)} = \left\{ \mathrm{d} \in y^{(k)} : \mathrm{d} \notin y^{(l)} \text{ for } l \neq k \right\} \tag{5}$$

are all not empty. They are maximal sets fulfiling the following condition:

$$\tilde{y}^{(k)} \subset y^{(k)} \quad \text{and} \quad \tilde{y}^{(k)} \cap y^{(l)} = \emptyset \quad \text{for} \quad l \neq k \quad (k = 1, 2, ..., K)$$

Thus, the matrix w^* defined by

$$w^*_{\mathrm{d}k} = 1 \quad \text{if} \quad \mathrm{d} \in \tilde{y}^{(k)} \quad \text{and} \quad w^*_{\mathrm{d}k} = 0 \quad \text{elsewhere} \tag{6}$$

satisfies

$$y^{(k)} w^* = \mathrm{Card}(\tilde{y}^{(k)})[\delta_{ki}], \quad k = 1, 2, ..., K \tag{7}$$

Consequently, the processing system $<\mathrm{D}, w^*, \mathrm{K}>$ indexes the representations for which $\tilde{y}^{(k)} \neq \emptyset$. If the family is strongly separable, the system indexes all $y^{(k)}$.

2.1.2. The General Case of Separability

Here, the matrix w^* is defined by the following recurrent formula:

$$M^{(0)} = 0 \quad \text{(zero matrix)},$$

$$M^{(k)} = M^{(k-1)} + \left(\overline{y}^{(k)T} \times c^{(k)} \right) \tag{8}$$

where $\overline{y}^{(k)}$ (see (4)) is now regarded as a binary vector, ^{T} denotes the transposition, $\times$ is the cross (outer) product, and

$$c^{(k)} = [\delta_{ki}] - (1/\alpha_k) y^{(k)} M^{(k-1)}, \qquad \alpha_k = \mathrm{Card}\left(\overline{y}^{(k)}\right) \tag{9}$$

Note that $M^{(k)}$ is correctly defined only if $\alpha_k > 0$, i.e., if $\overline{y}^{(k)} \neq \emptyset$; this involves the separability condition. The matrix $M^{(K)}$ is our weight matrix w^*. The proof of the indexing property of w^* follows:

Theorem 1. *We have*

$$y^{(k)}w^* = \alpha_k\,[\delta_{ki}], \quad k = 1,2,\ldots,K$$

so, if the sequence $y^{(1)},\ldots,y^{(K)}$ is separable, the matrix w^* is a solution of the indexing problem (1).

Proof. Using the principle of induction on k, we prove that

$$y^{(l)}M^{(k)} = \alpha_l\,[\delta_{li}], \quad l = 1,2,\ldots,k \tag{10}$$

First, we verify that $y^{(1)}M^{(1)} = \alpha_1\,[\delta_{1i}]$. In fact, since $c^{(1)} = [\delta_{1i}]$ and $\overline{y}^{(1)} = y^{(1)}$,

$$y^{(1)}M^{(1)} = y^{(1)}\left(\overline{y}^{(1)T}\times[\delta_{1i}]\right) = \left(y^{(1)}\cdot\overline{y}^{(1)}\right)[\delta_{1i}] = \mathrm{Card}\left(\overline{y}^{(1)}\right)[\delta_{1i}] = \alpha_1\,[\delta_{1i}]$$

($\cdot$ is the dot (scalar) product of vectors; we drop the transposition sign in the dot product).

Second, supposing that

$$y^{(l)}M^{(k-1)} = \alpha_l\,[\delta_{li}], \quad l = 1,2,\ldots,k-1$$

we prove (10). Thus, by (8) and (9),

$$\begin{aligned} y^{(l)}M^{(k)} &= y^{(l)}\left[M^{(k-1)} + \left(\overline{y}^{(k)}\times c^{(k)}\right)\right] \\ &= y^{(l)}M^{(k-1)} + \left(y^{(l)}\cdot\overline{y}^{(k)}\right)\left\{[\delta_{ki}] - (1/\alpha_k)y^{(k)}M^{(k-1)}\right\} \end{aligned}$$

But, $y^{(l)}\cdot\overline{y}^{(k)} = 0$ for $l < k$ and $y^{(k)}\cdot\overline{y}^{(k)} = \alpha_k$, so

$$y^{(l)}M^{(k)} = y^{(l)}M^{(k-1)} = \alpha_l\,[\delta_{li}], \quad l = 1,2,\ldots,k-1$$

by the inductive assumption, while

$$y^{(k)}M^{(k)} = y^{(k)}M^{(k-1)} + \alpha_k\left\{[\delta_{ki}] - (1/\alpha_k)y^{(k)}M^{(k-1)}\right\} = \alpha_k[\delta_{ki}].$$

This ends the inductive proof. Now we have in particular

$$y^{(k)}w^* = y^{(k)}M^{(K)} = \alpha_k[\delta_{ki}] = \varepsilon_k, \quad k = 1,2,\ldots,K.$$

■

2.2. The Self–Programming Procedure

Installation of a connectivity matrix in a layer of connections $\mathrm{N}\times\mathrm{N}'$ can be managed by the network itself. We mean by this that, in the learning phase, each connection changes its weight according to a law which expresses the new state of the connection in terms of the present state and the training signals introduced into the ends of the connection. Such a procedure of modulating the internal parameters of system will be referred to as self-programming.

The training signals are data which must be generated before or during the self-programming session. The ***training data*** are a pair $(\boldsymbol{a}, \boldsymbol{b})$, where $\boldsymbol{a}$ and $\boldsymbol{b}$ are distributions of activations in N and N', respectively. In this way, each connection $(\mathrm{n}, \mathrm{n}')$ receives training signals a_{n} and $b_{\mathrm{n}'}$ through n and n', and new weights

$$w_{\mathrm{nn}'}^{\mathrm{new}} = \omega(a_{\mathrm{n}}, w_{\mathrm{nn}'}, b_{\mathrm{n}'})$$

are automatically assigned to the connections.

The function $\omega(a, w, b)$ is the *self-programming law*. The variable w runs over all possible states of the connections. We denote the initial state of a connection, when no weight is assigned to it, by NIL.

The local character of the self-programming law allows for weight modulation in all connections in parallel. However, a more important feature of the law is its explicit, noniterative form: modulation of weights is done in one step. A self-programming procedure builds the final connectivity matrix through successive modulation steps. Thus, starting with the initial connectivity matrix

$$\boldsymbol{w}^{(0)} = [\mathrm{NIL}],$$

the matrices

$$\boldsymbol{w}^{(k)} = \left[\omega\left(a_{\mathrm{n}}^{(k)}, w_{\mathrm{nn}'}^{(k-1)}, b_{\mathrm{n}'}^{(k)}\right)\right]_{\mathrm{n}\in\mathrm{N},\ \mathrm{n}'\in\mathrm{N}'}$$

are successively generated, $\boldsymbol{a}^{(k)}$ and $\boldsymbol{b}^{(k)}$ being the training data in step k.

A solution of the indexing problem (1) will be built in K steps by modulation of the connection layer $\mathrm{D} \times \mathrm{K}$. The method we have developed makes the evolving network produce current training data from the successive training inputs $\boldsymbol{x}^{(k)}$. Thus, we will have $\boldsymbol{a}^{(k)} = \boldsymbol{y}^{(k)}$, while $\boldsymbol{b}^{(k)}$ will be derived from the current network output. For this purpose, a working phase will follow each training step and the training input will be fed to the network to produce the current output $\boldsymbol{z}$. Generally speaking, $\boldsymbol{b}^{(k)}$ will represent a deviation of $\boldsymbol{z}$ from the desired signal ε_k. We can see from the description of the self-programming procedure that the definition of $\boldsymbol{w}^{(k)}$ is recursive, in spite of the explicit form of the self-programming law.

We have to be precise here about how the evolving network, which may contain NIL connections, processes the training inputs. One method is to assign a default weight of zero to each NIL connection. It is convenient to express such an assignment in terms of a weight incrementation: if a *weight* NIL is incremented by a value w, the new weight is w : NIL$+w = w$. In particular, the operation $\boldsymbol{w}+\mathbf{0}$ symbolizes the assignment of weights zero to all possible NIL connections.

2.2.1. Self–Programming in the Case of Strong Separability

The matrix $\boldsymbol{w}^*$ defined by (6), which performs the indexing function in the easy case of strongly separable inputs, can be generated by self–programming in a non–recursive way, where the associations $(\boldsymbol{x}^{(k)}, [\delta_{ki}])$ themselves are used as the training data. The following theorem precisely specifies that procedure, which we call ***diagonal self–programming.***

Theorem 2. *Let*

$$\omega(1,\ \mathrm{NIL}, b) = b \qquad \text{(Rule 1)}$$

$$\omega(1, 1, 0) = 0 \qquad \text{(Rule 2)}$$

and

$$\omega(a, w, b) = w, \textit{ in all other cases} \qquad \text{(Rule 3)}$$

If $\boldsymbol{a}^{(k)} = \boldsymbol{x}^{(k)}$, $\boldsymbol{b}^{(k)} = [\delta_{ki}]$, $k = 1, 2, ..., K$, then we have $\boldsymbol{w}^* = \boldsymbol{w}^{(K)} + \mathbf{0}$.

Proof. Consider an arbitrary hidden neuron d $\in$ D. Three cases are possible:
a) d does not belong to any $\boldsymbol{y}^{(l)}$
b) d $\in \boldsymbol{y}^{(k)}$, but d $\notin \boldsymbol{y}^{(l)}$ for any $l \neq k$, i.e., d $\in \tilde{\boldsymbol{y}}^{(k)}$ (see (5)),
c) d belongs to at least two sets $\boldsymbol{y}^{(l)}$.

We will prove that, respectively,

$$\text{a}')\ w_{\mathrm{d}i}^{(K)} = \mathrm{NIL}, \qquad \text{b}')\ w_{\mathrm{d}i}^{(K)} = \delta_{ki}, \qquad \text{c}')\ w_{\mathrm{d}i}^{(K)} = 0$$

for all i $\in$ K.

In case a), the training input that is fed to d is zero in each self–programming step. Rule 3 will apply, so $w_{\mathrm{d}i}^{(1)} = w_{\mathrm{d}i}^{(2)} = ... = w_{\mathrm{d}i}^{(K)} =$ NIL.

In case b), Rule 3 applies to the connections (d, i), $i \in$ K, in steps $l = 1, 2, ..., k-1$, so $w_{\mathrm{d}i}^{(l)} =$ NIL. But $a_{\mathrm{d}}^{(k)} = y_{\mathrm{d}}^{(k)} = 1$, so $w_{\mathrm{d}i}^{(k)} = \omega(1,\ \mathrm{NIL}, b_i^{(k)}) = b_i^{(k)} = \delta_{ki}$. This assignment will not be changed in steps $k+1, ..., K$, because $a_{\mathrm{d}}^{(l)} = 0$ for $l > k$.

Consider case c). Let k be the smallest index such that d $\in \boldsymbol{y}^{(k)}$. Step k makes $w_{\mathrm{d}i}^{(k)} = \delta_{ki}$ (analogously to case b)). Let us see what happens to the connection (d, i) in steps $l > k$. If $i \neq k$, $w_{\mathrm{d}i}^{(k)} = 0$ and this assignment will not change in any step $l > k$, since only Rule 3 will apply. So, consider the connection (d, k); $w_{\mathrm{d}k}^{(k)} = 1$. Now we have $y_{\mathrm{d}}^{(l)} = 1 = a_{\mathrm{d}}^{(l)}$ for at least one $l > k$. Thus, if the weight of (d, k) was still 1 in step l–1, Rule 2 makes it zero in step l.

In this way, $w_{\mathrm{d}k}^{(K)} = 1$ for d $\in \tilde{\boldsymbol{y}}^{(k)}$ $(k = 1, 2, ..., K)$ and $w_{\mathrm{d}i}^{(K)} = 0$ in all other cases, i.e., when d$\notin \tilde{\boldsymbol{y}}^{(k)}$ for any k or d$\in \tilde{\boldsymbol{y}}^{(k)}$, but $k \neq i$. This proves the following equivalence

$$w_{\mathrm{d}k}^{(K)} + 0 = 1 \Leftrightarrow \mathrm{d} \in \tilde{y}^{(k)} \tag{11}$$

and gives the thesis $\boldsymbol{w}^{(K)} + \mathbf{0} = bw^*$; see (6).

2.2.2. The Test for Strong Separability

The self-programming method allows us to test the successive training inputs for separability before the corresponding modulation steps are executed. We avoid unsuccessful modulations (which are irreversible) in this way. For the test that follows self-programming step $K' \leq K$, we use the connectivity matrix $\boldsymbol{w}^{(K')}$, which is already installed on the network. Actually

$$\boldsymbol{y}^{(k)}\left(\boldsymbol{w}^{(K')} + \mathbf{0}\right) = \beta_k[\delta_{ki}], \quad k = 1, 2, ..., K'$$

This statement is an analogy to the statement (7); now, $\beta_k = \mathrm{Card}(\mathrm{B}_k)$, where

$$\mathrm{B}_k = \left\{\mathrm{d} \notin \boldsymbol{y}^{(k)} : \mathrm{d} \notin \boldsymbol{y}^{(l)} \text{ for } l \neq k, \quad l \leq K'\right\}$$

(in the case of $K' = K$, $\mathrm{B}_k = \tilde{\boldsymbol{y}}^{(k)}$; see(5)).

For testing, we assign default weights 1 to the connections that remain NIL after the K' steps of self-programming. This default weight matrix will be denoted by $\boldsymbol{w}_{\mathrm{test}}$. The test is based on the following proposition.

Proposition 1. *Suppose that* $\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(K')}$ *are strongly separable. Let* $\boldsymbol{z} = \boldsymbol{y}^{(K'+1)}\boldsymbol{w}_{test}$. *The family* $\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(K')}, \boldsymbol{y}^{(K'+1)}$ *is strongly separable if, and only if,*

$$z_k < \beta_k \quad k = 1, 2, ..., K', \ \textit{and} \ \ z_{K'+1} > 0$$

Proof. By analogy to (11), we have

$$w_{\mathrm{d}k}^{(K')} = 1 \Leftrightarrow \mathrm{d} \in \mathrm{B}_k \quad (k = 1, 2, ..., K')$$

Thus

$$z_k = \mathrm{Card}(\mathrm{B}_k \cap \boldsymbol{y}^{(K'+1)}), \ k = 1, 2, ..., K', \ \text{ and } \ z_{K'+1} = \mathrm{Card(A)}$$

where $\mathrm{A} = \left\{\mathrm{d} : w_{\mathrm{d},K'+1}^{(K')} = \mathrm{NIL}\right\}$.

Suppose that step $(K' + 1)$ of the diagonal self-programming was executed. We have $w_{\mathrm{d}k}^{(K'+1)} = 0$ for $\mathrm{d} \in \mathrm{B}_k \cap \boldsymbol{y}^{(K'+1)}$, because of Rule 2, thus

$$w_{\mathrm{d}k}^{(K'+1)} = 1 \Leftrightarrow \mathrm{d} \in \mathrm{B}_k \setminus \boldsymbol{y}^{(K'+1)}, \quad k = 1, 2, ..., K',$$

while

$$w_{\mathrm{d},K'+1}^{(K')} = 1 \Leftrightarrow \mathrm{d} \in \mathrm{A} \setminus \boldsymbol{y}^{(K'+1)}$$

because of Rule 1. By this

$$\begin{aligned} y^{(k)}w^{(K'+1)} &= \mathrm{Card}(\mathrm{B}_k \setminus y^{(K'+1)})[\delta_{ki}] = \mathrm{Card}(\mathrm{B}_k \setminus \mathrm{B}_k \cap y^{(K'+1)}))[\delta_{ki}] \\ &= (\beta_k - z_k)[\delta_{ki}] \\ y^{(K'+1)}w^{(K'+1)} &= \mathrm{Card}(\mathrm{A})[\delta_{K'+1,i}] = z_{K'+1}[\delta_{K'+1,i}] \end{aligned}$$

So, step $K'+1$ will be successful (what is equivalent to the strong separability of the family $y^{(k)}$, $k \le K'+1$) if, and only if, $\beta_k > z_k$ for $k \le K'$, and $z_{K'+1} > 0$. ■

2.2.3. Self–Programming in the General Case of Separability

The matrix $w^* = M^{(K)}$ defined 2.1.2. can be generated by a self–programming procedure, which we call *triangular*. A precise formulation of this result is the following theorem.

Theorem 3. *Let*

$$\omega(1,\ \mathrm{NIL}, b) = b \qquad \text{(Rule 1)}$$

$$\omega(a, w, b) = w \quad \textit{in all other cases} \qquad \text{(Rule 2)}$$

Taking $a^{(k)} = y^{(k)}$ and $b^{(k)} = c^{(k)}$ (see (9) for the definition of $c^{(k)}$) as training data, the self–programming will result in a matrix $w^{(K)}$ such that $w^{(K)}+0 = w^*$.

Proof. We will prove in fact that $w^{(k)} + 0 = M^{(k)}$ for all k. First, let us prove the folowing formula concerning the triangular self–programming with use of the training data $(y^{(k)}, c^{(k)})$:

$$\omega\left(y_{\mathrm{d}}^{(k)},\ w_{\mathrm{d}i}^{(k-1)},\ c_i^{(k)}\right) + 0 = w_{\mathrm{d}i}^{(k-1)} + \overline{y}_{\mathrm{d}}^{(k)} c_i^{(k)} \qquad (12)$$

for d $\in$ D and $i \in$ K.

This formula holds true when $w_{\mathrm{d}i}^{(k-1)} = \mathrm{NIL}$, independently of the training inputs: both sides of (12) are $c_i^{(k)}$ when Rule 1 is applicable, or they are zero when Rule 2 is to be applied. Thus, suppose that $w_{\mathrm{d}i}^{(k-1)} \neq \mathrm{NIL}$. We prove that both sides (12) are equal to $w_{\mathrm{d}i}^{(k-1)} + 0$. It is sufficient to prove that $\overline{y}_{\mathrm{d}}^{(k)} = 0$, i.e., that

$$w_{\mathrm{d}i}^{(k-1)} \neq \mathrm{NIL} \Rightarrow \overline{y}_{\mathrm{d}}^{(k)} = 0 \qquad (13)$$

Note that $\omega(a, w, b)$ can be NIL only if $w = \mathrm{NIL}$. So, once a connection is *switched on* (passes from the state NIL to a weighted state) in a self–programming step, it will never return to the state NIL. This means that for each $(\mathrm{d}, i) \in \mathrm{D} \times \mathrm{K}$ there is a unique l such that the connection is switched on in step l. In this step, the training input to d is obligatorily 1, i.e., $y_{\mathrm{d}}^{(l)} = 1$, while $y_{\mathrm{d}}^{(l')} = 0$ for $l' < l$ (and $w_{\mathrm{d}i}^{(l')} = \mathrm{NIL}$).

The connection (d, i) that we consider here was switched on in step $l < k$; this is the hypothesis of (13). Thus, $\mathrm{d} \in \boldsymbol{y}^{(l)}$, so $\mathrm{d} \notin \overline{\boldsymbol{y}}^{(k)}$, i.e., $\overline{y}_{\mathrm{d}}^{(k)} = 0$. Implication (13) and formula (12) are proven. Now we have:

$$\begin{aligned} \boldsymbol{w}^{(k)} + \mathbf{0} &= \left[\omega(y_{\mathrm{d}}^{(k)},\ w_{\mathrm{d}i}^{(k-1)},\ c_i^{(k)})\right] + \mathbf{0} = \left[w_{\mathrm{d}i}^{(k-1)}\right] + \mathbf{0} + \left[\overline{y}_{\mathrm{d}}^{(k)} c_i^{(k)}\right] \\ &= \boldsymbol{w}^{(k-1)} + \mathbf{0} + \overline{\boldsymbol{y}}^{(k)} \times \boldsymbol{c}^{(k)T} \end{aligned}$$

Since $\boldsymbol{w}^{(0)} + \mathbf{0} = \boldsymbol{M}^{(0)}$, induction gives

$$\boldsymbol{w}^{(k)} + \mathbf{0} = \boldsymbol{M}^{(k-1)} + \overline{\boldsymbol{y}}^{(k)} \times \boldsymbol{c}^{(k)T} = \boldsymbol{M}^{(k)}$$

■

Note that the *right-hand* training data $\boldsymbol{b}^{(k)}$ in the triangular self-programming procedure, which are $[\delta_{ki}] - (1/\alpha_k)\boldsymbol{y}^{(k)}\boldsymbol{w}^{(k-1)}$, in fact represent a deviation of the current output $\boldsymbol{z} = \boldsymbol{y}^{(k)}\boldsymbol{w}^{(k-1)}$ from the desired impulse signal $\varepsilon_k = \alpha_k[\delta_{ki}]$.

2.3. Dipole processing

Separable internal representations for an arbitrary family of binary inputs can be produced by the dipole processing introduced in this paragraph. The following theorem gives the theoretical result concerning the separability through dipole representations.

Theorem 4. *Consider a processing system* $<\mathrm{I}, \boldsymbol{w},\ \mathrm{D}>$ *(the transfer function in* D *is the function* ϕ *defined in (2)). For an arbitrary family of binary vectors* $\{\boldsymbol{x}^{(k)}\}$*, where* $k \leq \mathrm{Card(D)}$*, There exists an assignment of weights* $w_{\mathrm{id}} = -1, 0$ *or* 1 *such that the representations* $\boldsymbol{y}^{(k)}$ *are separable.*

We will prove the theorem at the end of the paragraph. First, let us introduce a simple version of the dipole processors, which seems to be an efficient separator in most practical applications involving the indexing of planar binary images.

Definition 2. A connection network $<\mathrm{I}, \boldsymbol{w},\ \mathrm{D}>$ is a *dipole processor* if for each $\mathrm{d} \in \mathrm{D}$ there exists exactly one ordered pair $(\mathrm{i}, \mathrm{i}')$ of input neurons such that $w_{\mathrm{id}} = 1$, $w_{\mathrm{i'd}} = -1$ and the other connections are NIL. The dipole d will be denoted by $\mathrm{d}(\mathrm{i}, \mathrm{i}')$.

We can assume that only the weighted connections can be physical realised, so there are $2\mathrm{n}(\mathrm{n}-1)$ connections and $\mathrm{n}(\mathrm{n}-1)$ dipoles in the network where $\mathrm{Card(I)} = \mathrm{n}$.

Let $\boldsymbol{y}$ be the internal representation of an image $\boldsymbol{x}$. Note that $\mathrm{d}(\mathrm{i},\mathrm{i}') \in \boldsymbol{y}$ if, and only if, $\mathrm{i} \in \boldsymbol{x}$ and $\mathrm{i}' \notin \boldsymbol{x}$.

The dipole processor can separate images even when there are linear dependences among them. As an illustration, consider two overlapping sets such as those in Figure 1 and the sequence

$$\boldsymbol{x}^{(1)} = \mathrm{A} \cap \mathrm{B}, \quad \boldsymbol{x}^{(2)} = \mathrm{A}, \quad \boldsymbol{x}^{(3)} = \mathrm{B}, \quad \boldsymbol{x}^{(4)} = \mathrm{A} \setminus \mathrm{B}, \quad \boldsymbol{x}^{(5)} = \mathrm{B} \setminus \mathrm{A}$$

The sequence $\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(5)}$ is separable. In fact, taking

$$\mathrm{d}_1 = \mathrm{d(b,a)}, \quad \mathrm{d}_2 = \mathrm{d(a,e)}, \quad \mathrm{d}_3 = \mathrm{d(c,e)}, \quad \mathrm{d}_4 = \mathrm{d(a,b)}, \quad \mathrm{d}_5 = \mathrm{d(c,b)},$$

one will have $\mathrm{d}_k \in \boldsymbol{y}^{(k)}$ and $\mathrm{d}_k \notin \boldsymbol{y}^{(l)}$ for $l < k$, $k = 1, ..., 5$.

When indexing regular figures (as opposed to noiselike patterns), ***short-range*** dipoles successfully separate the inputs. We say that $\mathrm{d(i, i')}$ is a short-range dipole if i and i′ are neighbours in the planar grid of input neurons. We mean horizontal, vertical, and diagonal connectivity, so there are less than $8n$ dipoles. This moderately sized set of hidden processing units gives separable internal representations in most of the realistic recognition problems, while ensuring, at the same time, a robust processing insensitive to input noises and degradations of input images. To an extent, damaged images are recognized and properly indexed. In this sense, the dipole solution of the indexing problem is also a solution of the image recognition problem.

There are families of images that the simple dipole processor does not separate. More complex dipoles do. To prove this, and the previously stated theorem, we generalize the notion of a dipole in the following way.

Definition 3. A hidden neuron d is a *dipole* if for exactly one input neuron i the connection (i, d) has the weight 1 and the set $\mathrm{A} = \{\mathrm{i} \in \mathrm{I} : w_{\mathrm{id}} = -1\}$ is not empty. The dipole d will be denoted by d(i, A).

Proof of Theorem 4. Note that the following equivalence holds true for $\boldsymbol{x} \subset \mathrm{I}$:

$$\mathrm{d(i,A)} \in \boldsymbol{y} \quad \text{if, and only if,} \quad \mathrm{i} \in \boldsymbol{x} \text{ and } \mathrm{A} \cap \boldsymbol{x} = \emptyset.$$

In fact,

$$\mathrm{d(i,A)} \in \boldsymbol{y} \Leftrightarrow y_{\mathrm{d}} = 1 \Leftrightarrow \phi\left(x_{\mathrm{i}} - \sum_{\mathrm{i}' \in \mathrm{A}} x_{\mathrm{i}'}\right) = 1$$

$$\Leftrightarrow x_{\mathrm{i}} = 1 \text{ and } x_{\mathrm{i}'} = 0 \text{ for } \mathrm{i}' \in \mathrm{A} \Leftrightarrow \mathrm{i} \in \boldsymbol{x} \text{ and } \boldsymbol{x} \cap A = \emptyset$$

(see (2) for the definition of ϕ).

Consider the dipoles $\mathrm{d}_k = \mathrm{d}(\mathrm{i}_k, \mathrm{I} \setminus \boldsymbol{x}^{(k)})$, where i_k is a neuron chosen in $\boldsymbol{x}^{(k)}$. We will prove that the dipole family $\{\mathrm{d}_k\}$ is sufficient for the separability of the sequence $\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, ..., \boldsymbol{y}^{(K)}$, if only the ordering of the sequence is an extention of the partial order $\supset$ in the family $\{\boldsymbol{x}^{(k)}\}$ (such an extention always exists). Thus, it is assumed that

$$\boldsymbol{x}^{(k)} \supset \boldsymbol{x}^{(l)} \Rightarrow k < l \tag{14}$$

We have to prove that if $l < k$, then $\mathrm{d}_k \notin \boldsymbol{y}^{(l)}$. By (14),

$$l < k \Rightarrow \neg(\boldsymbol{x}^{(k)} \supset \boldsymbol{x}^{(l)})$$

where $\neg$ signifies negation. Consequently, $(\mathrm{I}\backslash x^{(k)}) \cap x^{(l)} = \emptyset$, which implies $\mathrm{d}_k \notin y^{(l)}$. Finally, the K dipoles ensure the separability of the internal representations $y^{(k)}$. ■

2.4. Character Recognition

The recognition capabilities of dipole neural indexers were tested using images of the characters A–Z. A 68×68 array of black and white pixels in a fixed window on a computer monitor screen represented an image x. In this way, the window acted as the input layer of the network. A dipole d was represented as a record containing addresses of two neighbouring pixels in the window and the weights of connections (d,k), $k = 1,2,...,K$. Declared variables $z_1, z_2, ..., z_K$ were the outputs of the network.

The processing $x \to z$ consisted of the execution of the following two operations, separately and independently for each dipole:

a) read the values p, q of the two pixel addressed by d and compute $y_\mathrm{d} = \phi(p-q)$,

b) increment z_k by $y_\mathrm{d} w_{\mathrm{d}k}$, $k = 1,2,...,K$.

In the learning phase, the self-programming procedure updated the file of dipoles K times. In step k, the weights $w_{\mathrm{d}i}$ recorded in d were replaced by $\omega(y_\mathrm{d}, w_{\mathrm{d}i}, c_i)$ for each dipole, starting from initial weights NIL.

The diagonal self-programming was simulated first. Each step of the weights' modulation was proceded by the test described in 2.2.2. Since the internal representations of the images of A, B, ..., Z are not strongly separable, the test eliminated some of them from the family of training inputs. In fact, to ensure robustness of the processing, all condidates for $x^{(K'+1)}$ were eliminated for which

$$\beta_k - z_k \leq 50 \text{ for } k \leq K' \text{ or } z_{K'+1} \leq 50$$

in the notation introduced in 2.2.2.

Thus, A, ..., E pased the test in steps 1, ..., 5, respectively, while in step 6, the image of F and G did not. The results of testing F, G (which were discarded) and H (which was accepted) are given in Table 1.

Tab. 1. Outputs from the test network after step 5.

index cell	1	2	3	4	$5(K')$	$6(K'+1)$
β	536	269	425	163	206	–
z(F)	0	0	0	0	157	4
z(G)	23	16	378	7	1	110
z(H)	13	105	0	54	33	136

Eventually, seventeen characters passed the test. After the training, the network was switched to the recall (working) phase. Figure 2 shows some results of processing different input images. To make the reading of results easier, indices 1, 2, ..., 17 were replaced by their corresponding images. The values of the stimulation of output neurons were normalised and range from 1 to 100. They represent the percentage of the maximum possible signal that can be produced in given index (when $\boldsymbol{x} = \boldsymbol{x}^{(k)}$, where k is the index), showing similarity between $\boldsymbol{x}$ and $\boldsymbol{x}^{(k)}$.

Figure 3 illustrates the results of processing by the procedure of triangular self–programming. The training images now appear in an order that is an extention of the inclusion reation in the image family. However, three letters are still missing. Their indices produced a nonsignificant signal. This is due to possible non–separability of the family A–Z when using only short–range dipoles.

3. Self–Programmable Linear Approximators

In this section, we address those problems that require processing of analog distributed signals. These kinds of signals can be encountered in measurement systems involving cameras and other array sensors. The input to the neural processor comes in the form of a sampled distributed signal; one input neuron i receives a sample x_{i} of the sensor signal X. Variation of the measured physical value ν modifies the distribution, but the dependence between ν and a sample $x_{\mathrm{i}}(\nu)$ is not, in general, given analytically. The task of the neural processor is to evaluate ν from the *vector sample*

$$\boldsymbol{x}(\nu) = [x_{\mathrm{i}}(\nu)]_{\mathrm{i} \in \mathrm{I}}$$

where I denotes the input layer.

Definition. A set of sampling operations is *separable* (in a measurement range $\nu_{\min} \leq \nu \leq \nu_{\max}$) if the corresponding inputs $x_{\mathrm{i}}(\nu)$ are mutualy linearly independent functions in some vicinity of each ν.

We will construct a linear network which combines the functions $x_{\mathrm{i}}(\nu)$ into a function F approximating the measurand ν:

$$|F(\boldsymbol{x}) - \nu| < \varepsilon \text{ for } \boldsymbol{x} = \boldsymbol{x}(\nu) \tag{15}$$

The bigger the number of separable sampling procedures (and, consequently, the number of inputs neurons), the better a combination of $x_{\mathrm{i}}(\nu)$ approximates ν uniformly in $[\nu_{\min}, \nu_{\max}]$. Our approch consists in improving the approximation with each new sampling operation involved in the processing—as long as the separability is preserved. We investigate in particular systems where the sensor output is a continuous distribution of an analog signal $X(\mathrm{p})$ over a planar domain $\{\mathrm{p}\}$, and where, for different values of the parameter ν, the corresponding distributions X_ν are mutualy linearly independent (as elements of the infinite–dimentional space of continuous functions in the domain $\{\mathrm{p}\}$). For each p, $X_\nu(\mathrm{p})$ varies smoothly with the variation of ν. In such a case, for an arbitrary set $\{\mathrm{p_i} : \mathrm{i} \in \mathrm{I}\} \subset \{\mathrm{p}\}$, the sample functions

$$x_{\mathrm{i}}(\nu) = X_\nu(\mathrm{p_i}), \quad \mathrm{i} \in \mathrm{I}$$

form a family of smooth functions linearly independent in a vicinity of each point $\nu \in [\nu_{\min},\ \nu_{\max}]$. Here, the vector samples $\boldsymbol{x}$ are produced by the set of ***sample points*** $\{\mathrm{p_i}\}$.

The vector $\boldsymbol{x}(\nu)$ varies smoothly with a variation of the measurand, thus, sufficient measurement precision can be achieved when interpolating the measurand on a discrete set of vector samples. The interpolation problem can be stated as follows: construct a neural processor which transforms $\boldsymbol{x}(\nu_k)$ into ν_k, $k = 1,2,...,K$, for a given finite set $\{\nu_k\} \subset [\nu_{\min},\ \nu_{\max}]$. The values ν_k will be called ***calibration values.***

The interpolation problem can be reduced to an indexing problem. Consider a network $<\mathrm{I}, \boldsymbol{w}^*, \mathrm{K}>$, supposing that $\boldsymbol{x}(\nu_k)\boldsymbol{w}^* = [\delta_{ki}]$, $k = 1,2,...,K$. Let us extend the network by an ***interpolation*** layer, which connects all index neurons k to one output cell o (see Figure 4.; the figure represents a network where Card(I) = Card(K) $= K$). Putting $w_{k\mathrm{o}} = \nu_k$, we will have

$$F(\boldsymbol{x}(\nu_k)) = \nu_k, \quad k = 1,2,...,K$$

where $F : \mathbb{R}^n \rightarrow \mathbb{R}$ is the function of the extended network $<\mathrm{I}, \boldsymbol{w}^*, \mathrm{K}, \boldsymbol{w}, \mathrm{o}>$. The function F is a linear functional of the form:

$$F(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{w}^*\boldsymbol{w}$$

where $\boldsymbol{w} = [\nu_1, \nu_2, ..., \nu_K]^{\mathrm{T}}$.

Such a solution poses the problem of selecting the calibration values ν_k : first, the samples $\boldsymbol{x}(\nu_k)$ must be linearly independent vectors, and second, the function F, besides interpolating, should approximate ν uniformly in the interval $[\nu_{\min}, \nu_{\max}]$. A method for selecting appropriate calibration values will be a part of the self-programming procedure presented in the next paragraph.

3.1. Self-Programming: Case of Analog Input Vectors

The self-programming procedure is based on the classical method of matrix inversion by partitioning. The procedure generates the matrix $\boldsymbol{w}^*$. In each successive weight modulation step k, a new calibration value ν_k is found and the training data are generated from the vector sample $\boldsymbol{x}(\nu_k)$. We describe the self-programming first; ν_k will be considered to be an arbitrary parameter of step k. Next, the method of finding the optimal value of ν_k, optimal with respect to the uniformity of the approximation (15) will be introduced.

Let the input neurons be arranged in a sequence $\mathrm{i}_1, \mathrm{i}_2, ..., \mathrm{i}_n$, where n =Card(I). A sample x_i where $\mathrm{i}=\mathrm{i}_k$, or a training input a_i to the neuron $\mathrm{i}=\mathrm{i}_k$, will be simply written x_k, or a_k, respectively. Similarly, w_{ji} will denote the weight $w_{\mathrm{i}i}$ where $\mathrm{i}=\mathrm{i}_j$. Initially, all connection weights in the network $<\mathrm{I}, \boldsymbol{w}^*, \mathrm{K}, \boldsymbol{w}, \mathrm{o}>$ are set to 0:

$$\boldsymbol{w}^{(0)} = \boldsymbol{0}, \quad w_{k\mathrm{o}} = 0, \quad k = 1,2,...,n$$

The self-programming law is defined by the incrementation formula

$$\omega(a, w, b) = w + ab \tag{16}$$

The training data in step k are computed from $\boldsymbol{x}(\nu_k)$ with the use of the present weight matrix $\boldsymbol{w}^{(k-1)}$. Thus

$$\boldsymbol{a}^{(k)} = [\delta_{kj}]^T - \boldsymbol{w}^{(k-1)}\boldsymbol{s}^{(k)} \tag{17}$$

$$\boldsymbol{b}^{(k)} = \boldsymbol{b}_k(\nu_k) \tag{18}$$

where: $[\delta_{kj}]$ represents an input signal equal to 1 in i_k and to 0 in all other input neurons i_j,

$$\boldsymbol{s}^{(k)} = [x_k(\nu_1), x_k(\nu_2), ..., x_k(\nu_{k-1}, 0, ..., 0]^T \in \mathbb{R}^n \tag{19}$$

$$\alpha_k(\nu) = -\boldsymbol{x}(\nu)\cdot\boldsymbol{a}^{(k)} \tag{20}$$

$$\boldsymbol{b}_k(\nu) = 1/\alpha_k(\nu)\left(\boldsymbol{x}(\nu)\boldsymbol{w}^{(k-1)} - [\delta_{ki}]\right) \tag{21}$$

Step k is concluded by setting the weight of the connection (k, o) in the interpolation layer equal to ν_k. K modulation steps $(K \leq n)$ yield $\boldsymbol{w}^* = \boldsymbol{w}^{(K)}$ and $\boldsymbol{w} = [\nu_1, \nu_2, ..., \nu_K]^T$.

Selection of a specific value of ν_k that is optimal from the point of view of measurement precision, will be based on the transition formula (27), which describes the errors of the new network in terms of the previous errors and ν_k. Let us prove the most elementary statements concerning the weight modulation using the self-programming law (16) and the training data (17), (18).

Proposition 2. *The matrix term* $w_{li}^{(k)}$ *is zero when* $l > k$ *or* $i > k$.

Proof. Induction on k allows for an easy proof that $a_l^{(k)} = 0$ and $b_l^{(k)} = 0$ for $l > k$. Thus (16) modifies only the connections (i_l, l') where $l, l' \leq k$; other connections preserve their initial weights of zero. ■

This proposition states, in particular, that $w_{ll'}^{(\mathrm{K})} = 0$ for $l > K$, so, the neurons i_l where $l > K$ remain mute. Only the principal $K \times K$ minor of $\boldsymbol{w}^*$ takes part in the processing; we can simply assume that $n = K$ (as in Figure 4).

Proposition 3. *The α–function (20) can vanish only on a discrete set of points in the measurement range* $[\nu_{\min}, \nu_{\max}]$.

Proof. Note that $a_k^{(k)} = 1$, so $\boldsymbol{a}^{(k)}$ is a nonzero array. By the separability of the family $\{x_i(\nu)\}$, the dot product in (20) cannot vanish in a segment of the measurement range. This means that the α–function can be zero only at discrete points. ■

Theorem 5. *For* $l \leq k$, $\boldsymbol{x}(\nu_l)\boldsymbol{w}^{(k)} = [\delta_{li}]$.

Proof. Equation (16) signifies that $w_{ll'}^{(k)} = w_{ll'}^{(k-1)} + a_l^{(k)} b_{l'}^{(k)}$ for all $l, l' = 1, 2, ..., K$.

Thus

$$\boldsymbol{w}^{(k)} = \boldsymbol{w}^{(k-1)} + \boldsymbol{a}^{(k)} \times \boldsymbol{b}^{(k)}$$

Consequently,

$$\begin{aligned}\boldsymbol{x}(\nu)\boldsymbol{w}^{(k)} &= \boldsymbol{x}(\nu)\left[\boldsymbol{w}^{(k-1)} + \boldsymbol{a}^{(k)} \times \boldsymbol{b}_k(\nu_k)\right] \\ &= \boldsymbol{x}(\nu)\boldsymbol{w}^{(k-1)} + \left(\boldsymbol{x}(\nu)\cdot\boldsymbol{a}^{(k)}\right)\boldsymbol{b}_k(\nu_k) \\ &= \boldsymbol{x}(\nu)\boldsymbol{w}^{(k-1)} - \alpha_k(\nu)\boldsymbol{b}_k(\nu_k) \qquad (22)\end{aligned}$$

Now we can prove the theorem by induction on k.

The case $k = 1$. Because $\boldsymbol{b}_1(\nu) = -1/\alpha_1(\nu)[\delta_{1i}]$, we obtain immediately from (22)

$$\boldsymbol{x}(\nu_1)\boldsymbol{w}^{(1)} = -\alpha_1(\nu_1)\boldsymbol{b}_1(\nu_1) = [\delta_{1i}]$$

The inductive assumption:

$$\boldsymbol{x}(\nu_l)\boldsymbol{w}^{(k-1)} = [\delta_{li}], \quad l = 1, 2, ..., k-1.$$

This assumption implies

$$\alpha_k(\nu_l) = 0 \ \text{ for } \ l < k. \qquad (23)$$

In fact, by definitions (20) and (17),

$$\alpha_k(\nu_l) = -\boldsymbol{x}(\nu_l)\left\{[\delta_{kj}]^T - \boldsymbol{w}^{(k-1)}\boldsymbol{s}^{(k)}\right\} = -\boldsymbol{x}(\nu_l)\cdot[\delta_{kj}] + \boldsymbol{x}(\nu_l)\boldsymbol{w}^{(k-1)}\boldsymbol{s}^{(k)}$$

and, by the inductive assumption and definition (19),

$$\alpha_k(\nu_l) = -\boldsymbol{x}(\nu_l)\cdot[\delta_{kj}] + [\delta_{li}]\cdot\boldsymbol{s}^{(k)} = -x_k(\nu_l) + x_k(\nu_l) = 0$$

The inductive step. By the transition formula (22) and (23),

$$\boldsymbol{x}(\nu_l)\boldsymbol{w}^{(k)} = \boldsymbol{x}(\nu_l)\boldsymbol{w}^{(k-1)} = [\delta_{li}] \ \text{ for } \ l < k$$

while

$$\boldsymbol{x}(\nu_k)\boldsymbol{w}^{(k)} = \boldsymbol{x}(\nu_k)\boldsymbol{w}^{(k-1)} - \alpha_k(\nu_k)\boldsymbol{b}_k(\nu_k) = [\delta_{ki}]$$

by definition (21). ■

Concluding, the current function $F^{(k)}$ of the network interpolates the measurand ν at points $\boldsymbol{x}(\nu_1), \boldsymbol{x}(\nu_2), ..., \boldsymbol{x}(\nu_k)$, i.e.,

$$F^{(k)}\left(\boldsymbol{x}(\nu_l)\right) = \nu_l, \quad l = 1, 2, ..., k \qquad (24)$$

This interpolation formula follows the theorem above, since, obviously, the interpolation layer transforms $[\delta_{li}]$ into ν_l.

3.2. Analysis of the Uniform Approximation

Let us now prove a transition formula for the network's deviation function. Recall that after k steps of modulation, the network performs a mapping

$$\boldsymbol{x} \rightarrow F^{(k)}(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{w}^{(k)}\boldsymbol{\nu}^{(k)}$$

where $\boldsymbol{\nu}^{(k)} = [\nu_1, ..., \nu_{k-1}, \nu_k, 0, ..., 0]^T \in \mathbb{R}^K$.

The present deviation of $F^{(k)}(\boldsymbol{x})$ from ν, for $\boldsymbol{x} = \boldsymbol{x}(\nu)$, is the function

$$E_k(\nu) = F^{(k)}(\boldsymbol{x}(\nu)) - \nu = \boldsymbol{x}(\nu)\boldsymbol{w}^{(k)}\boldsymbol{\nu}^{(k)} - \nu \tag{25}$$

which also expresses the approximation errors of the system after the stage k. By the interpolation formula (24)

$$E_k(\nu_l) = 0, \quad l = 1, 2, ..., k \tag{26}$$

An estimate of the uniform convergence of the deviations to 0 when k increases is based on the following theorem:

Theorem 6. *The following formula expresses the relation between* $E_{k-1}(\nu)$ *and* $E_k(\nu)$:

$$E_k(\nu) = E_{k-1}(\nu) - \frac{\alpha_k(\nu)}{\alpha_k(\nu_k)} E_{k-1}(\nu_k) \tag{27}$$

Proof. Definition (21) gives

$$b_k(\nu_k)\cdot\boldsymbol{\nu}^{(k)} = 1 \Big/ \alpha_k(\nu_k) \left[\boldsymbol{x}(\nu_k)\boldsymbol{w}^{(k-1)}\boldsymbol{\nu}^{(k-1)} - \nu_k\right] = E_{k-1}(\nu_k) \Big/ \alpha_k(\nu_k)$$

(see(25)). Thus, we obtain by applying (22)

$$\begin{aligned} E_k(\nu) &= \boldsymbol{x}(\nu)\boldsymbol{w}^{(k)}\boldsymbol{\nu}^{(k)} - \nu = \boldsymbol{x}(\nu)\boldsymbol{w}^{(k-1)}\boldsymbol{\nu}^{(k)} - \nu - \alpha_k(\nu) b_k(\nu_k)\cdot\boldsymbol{\nu}^{(k)} \\ &= E_{k-1}(\nu) - [\alpha_k(\nu) / \alpha_k(\nu_k)] \, E_{k-1}(\nu_k) \end{aligned}$$

■

Let us transform the identity (27) into the following formula

$$E_k(\nu) = E_{k-1}(\nu) - Q(\nu_k)\alpha_k(\nu) = E_{k-1}(\nu)\,[1 - Q(\nu_k) / Q(\nu)] \tag{28}$$

where $Q(\nu) = E_{k-1}(\nu)/\alpha_k(\nu)$. The two functions E_{k-1} and α_k are smooth functions sharing the same set of roots (see (26) and (23)). They are, in fact, nearly proportional. An evaluation of this proportionality is beyond the scope of this paper. Let us note, however, that this proportionality promotes the modulation method that we are investigating here: Q is a continuous function (the zeros of the numerator and denominator can be simplified), everywhere positive or everywhere negative, fitting a constant function well enough, so that $Q(\nu_k)/Q(\nu)$ approximates 1 for all ν_k. In any event, $|E_k(\nu)|$ is smaller than $|E_{k-1}(\nu)|$ at points ν where $Q(\nu_k)/Q(\nu) < 2$. The better the proportionality between the α–function and the deviation function, the closer $Q(\nu_k)/Q(\nu)$ approaches 1 and, by (28), the magnitude of the deviation diminishes in a larger neigbourhood of the current calibration value. On the other hand, the present deviation function (28) is expressed as a pointwise product of two factors, thus, its uniform norm will decrease in the most efficient way if a zero for one factor coincides with the extremum of another factor. Consequently, to temper the uniform norm of $E_k(\nu)$, we chose

$$\nu_k = \textit{maximum point of the first factor } |E_{k-1}(\nu)| \tag{29}$$

the second factor $1 - Q(\nu_k)/Q(\nu)$ is zero in the point $\nu = \nu_k$.

This selection has one more advantage. Due to the proportionality, ν_k is also a maximum point of $|\alpha_k|$, so the magnitude of connection weights grows in the slowest possible way ($\alpha_k(\nu_k)$ is a divisor). The selection (29) delays the moment when the computation errors overtake the approximation errors. In our experiments, this did not happen until the approximation errors diminished to an ε of the order of 10^{-4}. That precision required 12 calibration steps. The matrix inversion process and all computations were performed by means of single precision floating point operations, for a more faithful simulation of real world situations.

3.3. Example

We illustrate the self-programming method with a simulated of ν from the Gaussian distributions

$$X_\nu(\mathrm{p}) = c(\nu)\exp(-\nu^2 r(\mathrm{p})^2),$$

where $r(\mathrm{p})$ is the distance between p and the center of the distribution and $c(\nu)$ is a smooth function. The accuracy of the processing versus the network size is evaluated in the measurement range $0 \le \nu \le 10$, when $c(\nu) = 5\sin(\nu/4)$. The sample of X_ν is an array $x_k = x_k(\nu) = X_\nu(\mathrm{p})$, $k = 1, 2, ..., K$, where p_k are sampling points. In our example, $r(\mathrm{p}_k) = k/40$, $k \le K = 12$.

As little as 6 calibration steps make the error of the network practically negligible; see the plot of $F^{(6)}(x(\nu))$ versus ν in Figure 5. The uniform decrease of the deviation function is illustrated in Figure 6. Thus, Figure 6a illustrates step 7 of the self-programming. Although the deviation and α–functions versus ν are represented on a logarithmic scale, we can easily see that they are proportional to a great degree. The small triangles indicate the previous calibration values. According to (26) and (23), the two functions are zero at those points. The new calibration value (where the deviation is maximum) is indicated by an arrow at the bottom of the grid. Figure 6b shows the situation in step 9; at this step the deviation in the whole measurement range is well below 1/100. In step 12 (see figure 6c), the computer errors become visible in the display of the deviation function. The α–function is no longer distinguishable from zero. Beyond step 12 (figure 6d), the computation errors contribute significantly to the approximation errors, so any continuation of the modulation is pointless. Finally, the network uses only 12 input neurons and 12 samples of the input distribution; the maximum deviations are smaller than 10^{-4}.

4. Conclusions

In this paper we have presented a self–programming approch to non–iterative learning of the association matrix to be used on a multilayer feedforward neural network. We have addressed two particular problems: indexing as a content addressable memory problem and function approximation.

A key issue in object recognition by indexing is the separability of the object family. Two self–programming algorithms have been proposed: diagonal self–programming for strongly separable objects and triangular self–programming for the weakly separable object family. The processing of object images by a short–range dipole operator was used in order to achieve separability. The object recognition process was illustrated by an example of the recognition of a set of characters. The results show very good immunity to noise of the dipole–processing–based recognition. In certain situations, however, short–range simple dipoles do not provide sufficient separability. In those cases, more complex dipoles should be considered.

The extention of self–programming method to the real number domain led us to the solution of the function approximation problem, oriented toward the requirements of the measurement systems. In those systems, it is ofen necessary that the value of the measurand at the calibration points be extracted very precisely, while maintaining precise interpolation between the points. We showed that modulation of connection weights with regard to more and more refined interpolation does not require more than resolving linear problems and that the solution can be generated by a self–programming procedure. Thus, we construct universal approximators of the measurands, using only the two–layer feedforward *linear* networks. The modulation is a few–step process of defining the next optimal interpolation point and adjusting the weights to fit the network to the increasing interpolation task. The method provides the user with a possibility of on–going insight into approximation precision and with the means to determine optimal selection of the successive training inputs.

References

Barnard E. and Casasent D. (1989): *A comparison between criterion functions for linear classifiers, with an application to neural nets.– IEEE Trans. Systems, Man, and Cybern.*, v.19, Sept./Oct. pp.1030–1041.

Barnard E. (1992): *Optimization for training neural nets.– IEEE Trans. Neural Networks*, v.3, No.2, pp.232–240.

Dennis J.E., Jr. and Schnabel R.B. (1983): *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.*– Englewood Cliffs, NJ: Prentice–Hall.

Fitch J.P., Lehman S.K., Dowla F.U., Lu S.Y., Johansson E.M. and Goodman D.M. (1991): *Ship wake-detection procedure using conjugate gradient trained artificial neural networks.*– *IEEE Trans. Geoscience and Remote Sensing*, v.29, No.5, pp.718–725.

Funahashi K.-I. (1989): *On the approximate realization of continuous mappings by neural networks.*– *Neural Networks*, v.2, pp.183–192.

Gorman R.P. and Sejnowski T.J. (1988): *Analysis of hidden units in a layered network trained to classify sonar signals.*– *Neural Networks*, v.1, No.1, pp.75–90.

Gray D.L. and Michel A.N. (1992): *A training algorithm for binary feedforward neural networks.*– *IEEE Trans. Neural Networks*, v.3, No.2, pp.176–194.

Hebb D.O. (1949): *Organization of Behavior.*– New York: Science Editions.

Hornik K., Stinchcombe M. and White A. (1989): *Multilayer feedforward networks are universal approximators.*– *Neural Networks*, v.2, pp.359–366.

Hornik K., Stinchcombe M. and White A. (1990): *Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks.*– *Neural Networks*, v.3, pp.551–560.

Hwang J.N. and Kung S.Y. (1989): *Parallel algorithms/architectures for neural networks.*– *VLSI Signal Processing*, pp.221–251.

Kohonen T. (1988): *Self-Organization and Associative Memory.*– New York: Springer–Verlag.

Kyuma K., Ohta J., Kojima K. and Nakayama T. (1988): *Optical neural systems: system and device technologies.*– Proc. *Optical Computing'88*, Toulon, France, 29 Aug. – 2 Sept., pp.475–484.

Miller W.T., Glanz F.H. and Kraft L.G. (1990): *CMAC: An associative neural network alternative to backpropagation.*– Proc. IEEE, v.78, No.10, pp.1561–1567.

Poelzleitner W. and Wechsler H. (1990): *Selective and focused invariant recognition using distributed associative memories (DAM).*– *IEEE Trans. Pattern Analysis and Machine Intelligence*, v.12, No.8, pp.809–814.

Porada E. and Zaremba M.B. (1992): *A connectionist method for distributed sensory signal processing.*– *Research report* No.92/01–2, Dept. d'informatique, UQAH, Canada.

Rosenblatt R. (1959): *Principles of Neurodynamics.*– New York: Spartan Books.

Rumelhart D.E., Hinton G.E. and Williams R.J. (1986): *Learning internal representations by error propagation. In*: Parallel Distributed Processing.– Cambridge: MIT Press, MA, v.1, pp.318–362.

Stinchcombe M. and White H. (1989): *Universal approximation using feedforward networks with non-sigmoidal hidden layer activation functions.*– Proc. Int. Joint Conf. *Neural Networks*, Washington, USA, v.1, pp.613–617.

Zaremba M.B., Bock W.J., Porada E. and Skorek A. (1991): *The recognition and measurement of optically detected variables using interactional neural networks.*– Proc. Int. Conf. *Neural Networks*, San Diego, USA, 29–31 May, v.2, pp.77–85.

Received April 9, 1992
Revised August 5, 1992

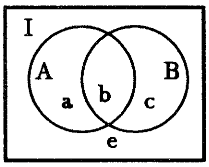


Fig. 1. Feature representation sets.

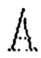
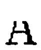
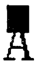
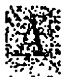
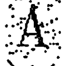
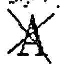
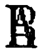
	64	0	0	17	2	0	0	6	4	17	0	4	0	5	4	0	0
	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	64	0	5	0	0	0	0	0	4	1	0	0	0	1	0	1	0
	68	0	17	17	0	17	18	24	24	16	13	14	0	20	2	26	20
	90	0	10	7	0	17	0	6	7	1	0	3	0	7	5	0	0
	91	0	7	17	0	0	9	0	4	3	0	11	0	3	7	3	0
	70	55	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Z	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100
⋮									⋮								
B	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	A	B	C	D	E	H	J	K	M	N	R	S	T	W	X	Y	Z

Fig. 2. Results of diagonal self-programming.

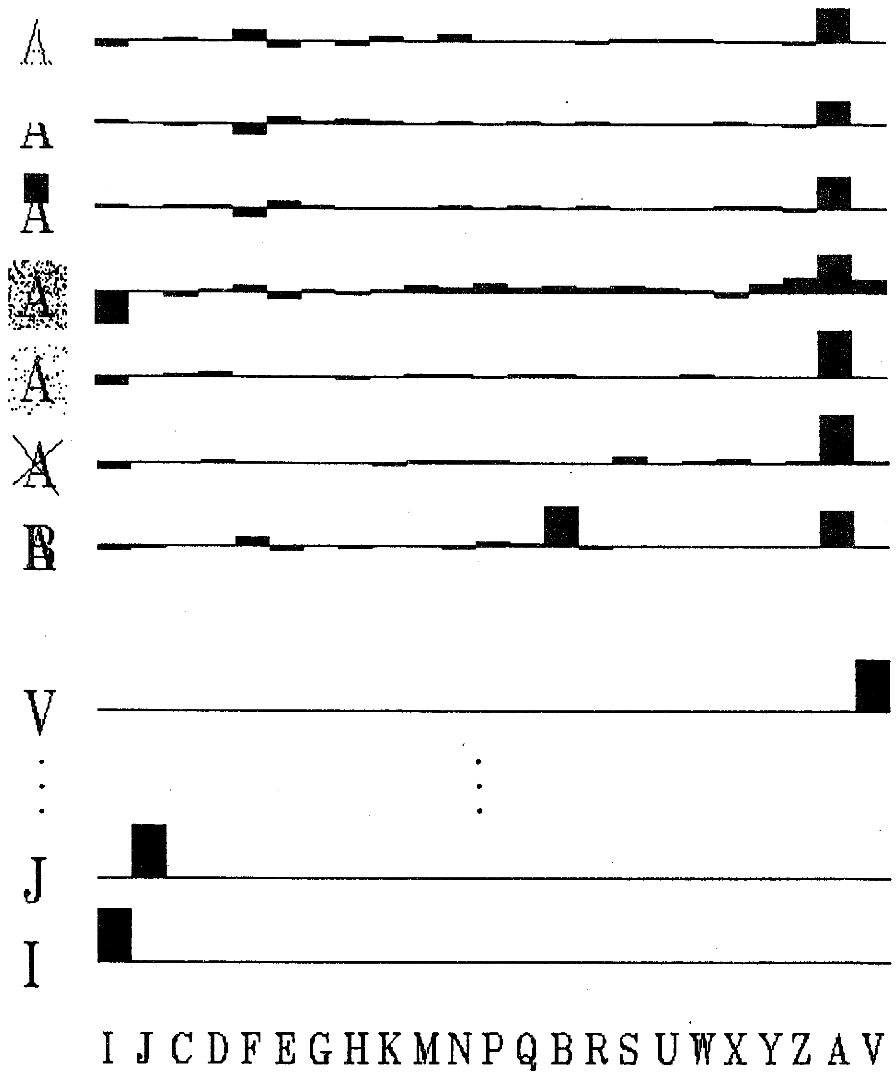


Fig. 3. Results of triangular self-programming.

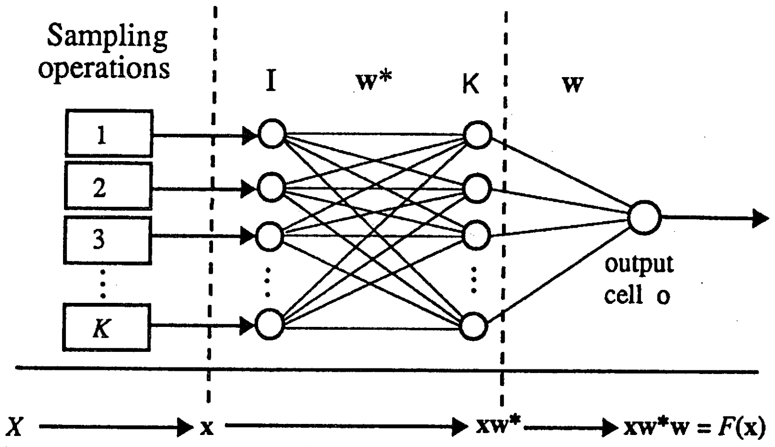


Fig. 4. Network architecture.

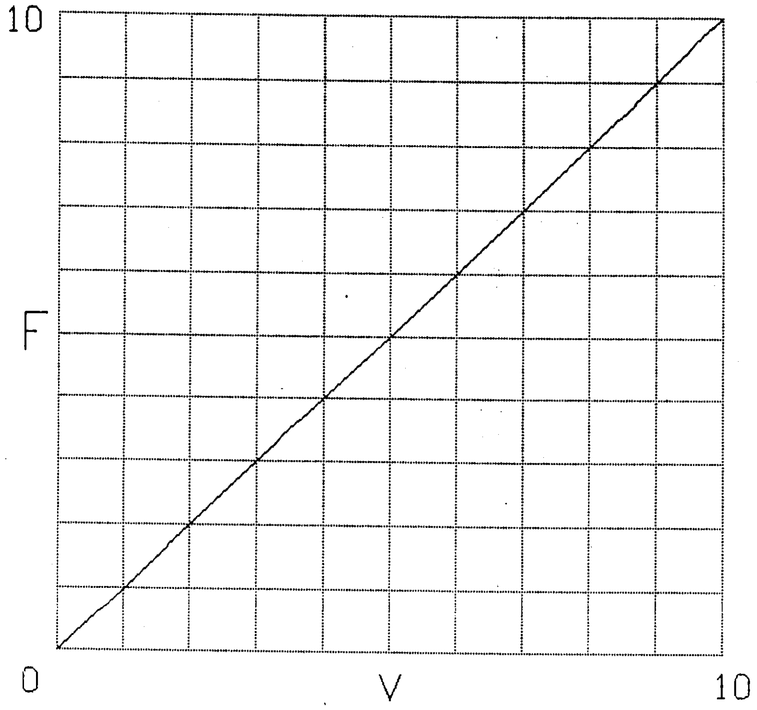


Fig. 5. Network outputs versus measurand after 6 steps.

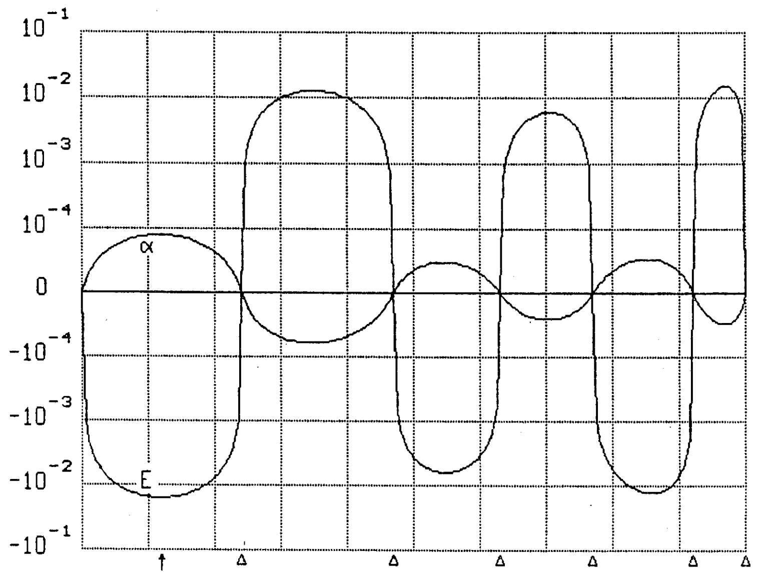


Fig. 6a. Precision of the self–programming procedure before the execution of step 7.

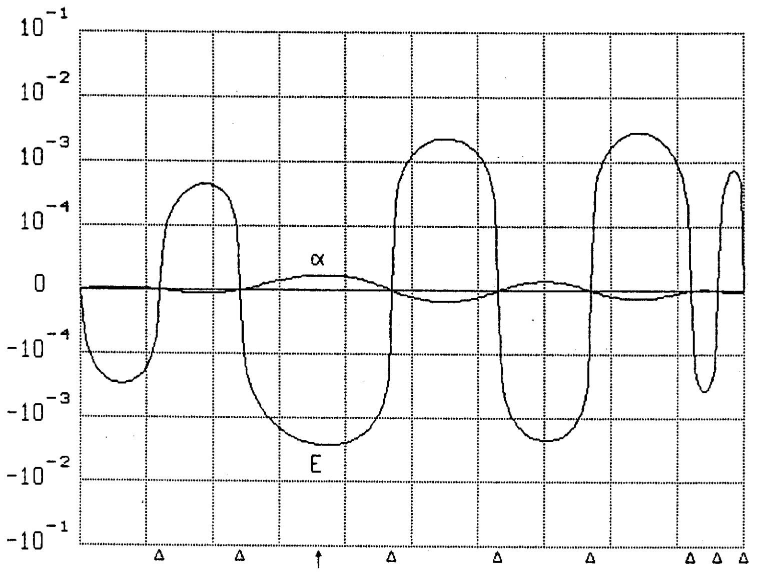


Fig. 6b. Precision of the self–programming procedure before the execution of step 9.

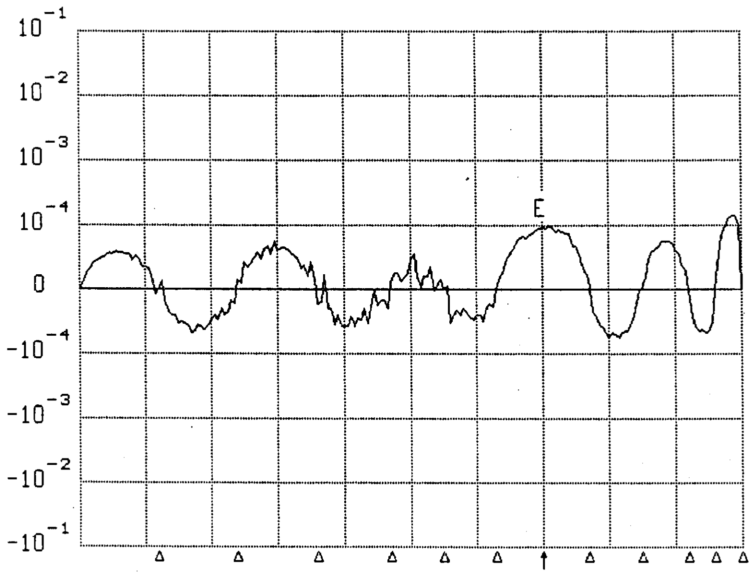


Fig. 6c. Precision of the self–programming procedure before the execution of step 12.

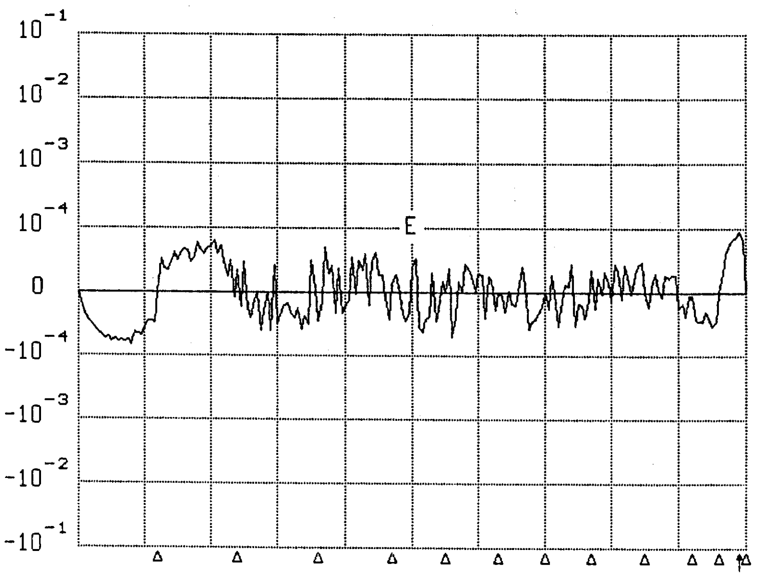


Fig. 6d. Precision of the self–programming procedure before the execution of step 13.