Appl. Math. and Comp. Sci., 1998, Vol.8, No.3, 599–609

SOLVING DIFFERENTIAL EQUATIONS WITH NONLINEAR PERCEPTRON[†]

PIOTR S. SZCZEPANIAK*, BARTOSZ LIS*

The work concerns training neural networks for approximate mappings being solutions to differential equations, especially partial-differential equations. The presented approaches fall into two categories. In the first one, backpropagation training is combined with an arbitrary numerical method used for obtaining tabulated solutions to the equations for training sequences. In the other, the neural network is forced to suggest a solution to the equation and to keep on improving that mapping during the backpropagation process. The other approach implies certain modifications in the structures of the neural network, neuron and neural signals.

1. Introduction

In the last decade, artificial neural networks have been in the focus of interest of many researchers, and some important results, described in (Cybenko, 1989), showing that one-hidden-layer nonlinear perceptrons are capable of approximating an arbitrary function arbitrarily well, have been achieved.

Many applications of neural networks (c.f. Nerrand *et al.*, 1994) deal with the generating of solutions to differential equations. In most cases, especially when the equation under consideration contains a derivative with respect to time, apart from normal arguments, the network gets its delayed outputs as inputs. Such a trick allows the network, after some necessary training, to forecast the values of the solution after an arbitrarily given step.

This paper deals with one-hidden-layer perceptrons prepared to produce the value to the solution of the differential equation for any arguments contained in a polyhedral stretched on the arguments of the training sequence, when the solution values for certain points are presented only during training, and not during the operation time.

Section 3 presents such networks together with methods of preparing them to produce proper values. In Section 3.1, the method presented by Szczepaniak and Lis (1994a; 1994b) is briefly described. The network is trained on a number of solutions

[†] The paper was originally presented at the 2nd Conference *Neural Networks and Their Applications*, Szczyrk, Poland, 1996.

* Institute of Computer Science, Technical University of Łódź, ul. Sterlinga 16/18, 90–217 Łódź, Poland, e-mail: piotr@ics.p.lodz.pl, bartoszl@ics.p.lodz.pl.

obtained previously for a set of equation parameters, and then it is able to generate solutions for non-presented values of parameters. The solutions used for training can be computed using any numerical or analytical method. Subsections 3.2 and 3.3 present a method of forcing the network to generate a solution when only a differential equation and initial (or boundary) conditions are known during training. The method does not depend on other methods of solving differential equations. Subsection 3.2 presents the target function which is optimized during training. The method requires networks capable of computing derivatives of their outputs with respect to their inputs. Such networks are described in Subsection 3.3. Section 4 presents some numerical experiments with those networks.

2. Problem Formulation

Let us consider a system of ordinary differential equations of the form

$$\boldsymbol{f}(t, \boldsymbol{a}, \boldsymbol{y}, \dot{\boldsymbol{y}}) = \boldsymbol{0} \tag{1}$$

with the initial condition depending on a parameter vector $\boldsymbol{a}$:

$$\boldsymbol{y}(\bar{t}^{[1]}) = \bar{\boldsymbol{y}}(\boldsymbol{a}) \tag{2}$$

where $\boldsymbol{a} \in D_a \subset \mathbb{R}^A$, $t \in D_t \subset \mathbb{R}$, $\boldsymbol{y} \in D \subset \mathbb{R}^M$, $\dot{\boldsymbol{y}} \in \dot{D} \subset \mathbb{R}^M$; $M, A \in \mathbb{N}$ and $D_t, D, \dot{D}$ are closed sets. Moreover, $\boldsymbol{f}: D_a \times D_t \times D \times \dot{D} \to \mathbb{R}^M$, $\bar{\boldsymbol{y}}: D_a \to \mathbb{R}^M$, and all these mappings are of class C^1.

Such a formulation of the problem in which the derivative of the unknown mapping is given implicity does not guarantee the uniqueness of the solution. The theory states (c.f. Nikliborc, 1951) irregular points whose number is equal to the number of solutions to the equation $\boldsymbol{f}(\bar{t}^{[1]}, \boldsymbol{a}, \bar{\boldsymbol{y}}(\boldsymbol{a}), \boldsymbol{p}) = \boldsymbol{0}$ for variable $\boldsymbol{p}$. Assuming that the solution does not contain irregular points, we can ensure its uniqueness by adding the following conditions:

$$\frac{\partial \boldsymbol{y}}{\partial t}(\bar{t}^{[1]}) = \bar{\boldsymbol{p}}(\boldsymbol{a}) \tag{3}$$

where $\bar{\boldsymbol{p}}: D_a \to \mathbb{R}^M$, and $\bar{\boldsymbol{p}}$ is in C^1.

Now, let us consider a system of partial equations with first-order derivatives:

$$\boldsymbol{f}\left(\boldsymbol{t}, \boldsymbol{a}, \boldsymbol{y}, \frac{\partial \boldsymbol{y}}{\partial t_1}, \frac{\partial \boldsymbol{y}}{\partial t_2}, \dots, \frac{\partial \boldsymbol{y}}{\partial t_{M_1}}\right) = \boldsymbol{0} \tag{4}$$

with the condition

$$\boldsymbol{y}(\bar{\boldsymbol{t}}(\xi)) = \bar{\boldsymbol{y}}(\xi, \boldsymbol{a}) \tag{5}$$

where $\boldsymbol{t} \in D_t \subset \mathbb{R}^{M_1}$, $\boldsymbol{a} \in D_a \subset \mathbb{R}^A$, $\boldsymbol{y} \in D \subset \mathbb{R}^M$, $M_1, M \in \mathbb{N}, D_t, D$ are closed sets, $\boldsymbol{f}: D_t \times D_a \times D \times (\mathbb{R}^M)^{M_1} \to \mathbb{R}^M$ belongs to class C^1, $\xi \in D_\xi \subset \mathbb{R}^{M_1-1}$, $\bar{\boldsymbol{t}}: D_\xi \to \partial D_t$, $\bar{\boldsymbol{y}}: D_\xi \times D_a \to D$.

Without loss of generality, we can focus our considerations on systems with first-order derivatives only, because any equation (or a system of equations) is easily reducible to such a form.

Our task is to train a neural network to approximate the solution to (1)–(3) or (4)–(5) on the domains of arguments D_t and parameters D_a.

3. Presentation of the Methods

3.1. Training of a Neural Network on Tabulated Solutions

The first approach consists in finding and tabulating the solutions to a given equation or a system of equations in any numerical or analytical way for a number of parameters $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_L \in D_a$. Those tabulated values create the training sequence $\{(\tilde{\boldsymbol{t}}^{[\nu]}, \tilde{\boldsymbol{y}}^{[\nu]})\}_{\nu=1,\ldots,N_1}$, where $\tilde{\boldsymbol{t}}^{[\nu]} = [\boldsymbol{t}^{[\nu]^T}, \boldsymbol{a}^{[\nu]^T}]^T \in D_t \times \{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_L\}$, $\tilde{\boldsymbol{y}}^{[\nu]} = \boldsymbol{y}_{\boldsymbol{a}}(\boldsymbol{t}^{[\nu]})$ and $\boldsymbol{y}_{\boldsymbol{a}}(\boldsymbol{t})$ is a solution to (1)–(3) (in this case $\boldsymbol{t}$ is a scalar and $M_1 = 1$) or (4)–(5). A feed-forward network has $M_1 + A$ inputs and M outputs, and is trained with the above sequence, so it is expected to approximate the mapping defined as

$$\boldsymbol{u}\colon D_t \times D_a \to \mathbb{R}^M, \qquad \bigwedge_{\substack{\boldsymbol{t}\in D_t \\ \boldsymbol{a}\in D_a}} \boldsymbol{u}(\boldsymbol{t}, \boldsymbol{a}) = \boldsymbol{y}_{\boldsymbol{a}}(\boldsymbol{t}) \tag{6}$$

If $\boldsymbol{u}$ is continuous, it can be approximated with any precision depending only on the density of terms of the training sequence in their domain $D_t \times D_a$, and, of course, certain parameters of the network and its training: the number of neurons and a planned error on the training sequence. We will call this method an SA method (Solution Approximating method).

3.2. Target Function for Solving Differential Equations

The process of training a neural network is an iteration of succesive suggestions of mappings made by the network, followed by computation of a certain formula (the target function) with respect to the suggested mapping as the terms of the training sequence are being presented. Then the neurons' weights are changed so that the next generated mapping may produce a smaller value of the target function. One turn of the iteration is referred to as an epoch. The training concludes when the target function is close enough to zero.

Let the mapping the network produces in epoch τ be denoted by $\boldsymbol{z}^{[\tau]}\colon \mathbb{R}^{n^{(0)}} \to \mathbb{R}^{n^{(K)}}$, where the network contains $K+1$ layers numbered from 0 to K, $n^{(k)}$, $0 \le k \le K$ is the number of neurons in layer k.

To force the neural network to solve a differential equation, the target function should be expressed in such a way that it takes the zero value for the solution to the

equation. Thus its value $s^{[\tau]}$ in the τ-th epoch can be written down as

$$s^{[\tau]} = \frac{1}{2}\sum_{\nu=1}^{N_1}\left\|z^{[\tau]}(\tilde{t}^{[\nu]}) - \tilde{y}^{[\nu]}\right\|^2$$

$$+\frac{1}{2}\sum_{\nu=1}^{N_2}\left\|f\left(t^{[\nu]}, a^{[\nu]}, z^{[\tau]}(\tilde{t}^{[\nu]}), \frac{\partial z^{[\tau]}}{\partial t_1}(\tilde{t}^{[\nu]}), \frac{\partial z^{[\tau]}}{\partial t_2}(\tilde{t}^{[\nu]}), \dots, \frac{\partial z^{[\tau]}}{\partial t_{M_1}}(\tilde{t}^{[\nu]})\right)\right\|^2 \quad (7)$$

The first component, called the error function, is formed from the tabulated initial or boundary conditions (2) or (5) and is computed only on the first N_1 terms of the training sequence which is chosen from the domain of the initial condition. The second component, called the correction function, contains the function from (1) or (4) and is computed on arguments coming from the domain $D_t \times D_a$. If $N_1 < N_2$, the second component is computed on the boundary of $D_t \times D_a$ for the first N_1 terms and then on the interior of the domain for the other terms. Since the desired values of the mapping function are unknown for the terms from the interior, these terms are called the sampling sequence. The first N_1 terms for which the initial condition defines the desired values of the searched mapping are called the learning sequence.

To ensure the uniqueness of the solution, (7) can be expanded with another component, similar to the error function but based on (3), so instead of the unknown mapping and its desired value it contains their derivatives.

3.3. Finding Derivatives of Neural Network Outputs with Respect to its Inputs

For the target function (7) to be computed, the network should compute derivatives of its outputs with respect to its inputs. These derivatives can be computed in the forward propagation by a recurrent formula similar to that of the normal output (Hornik *et al.*, 1990):

- normal sum of inputs:

$$X_{i,0}^{(k)[\nu,\tau]} = \sum_{j=1}^{n^{(k-1)}} w_{i,j}^{(k)[\tau]} x_{j,0}^{(k-1)[\nu,\tau]} + w_{i,0}^{(k)[\tau]} \quad (8)$$

- auxiliary sum of derivatives:

$$X_{i,p}^{(k)[\nu,\tau]} = \sum_{j=1}^{n^{(k-1)}} w_{i,j}^{(k)[\tau]} x_{j,p}^{(k-1)[\nu,\tau]}, \quad p = 1,\dots,M_1 \quad (9)$$

- normal output of a neuron:

$$x_{i,0}^{(k)[\nu,\tau]} = \sigma^{(k)}\left(X_{i,0}^{(k)[\nu,\tau]}\right) \quad (10)$$

- its derivative:

$$x_{i,p}^{(k)[\nu,\tau]} \stackrel{\text{df}}{=} \frac{\partial x_{i,0}^{(k)[\nu,\tau]}}{\partial x_{p,0}^{(0)[\nu,\tau]}} = \left(\sigma^{(k)}\right)' \left(X_{i,0}^{(k)[\nu,\tau]}\right) X_{i,p}^{(k)[\nu,\tau]}, \quad p = 1,\ldots,M_1 \tag{11}$$

where $x_{i,0}^{(k)[\nu,\tau]}$ is a normal output of the i-th neuron from the k-th layer for the ν-th presented term of the training sequence in the τ-th epoch, $x_{i,p}^{(k)[\nu,\tau]}$, $p = 1,\ldots,M_1$ stands for its derivative after the p-th input to the neural network, $\sigma^{(k)}$ is the activation function of the k-th layer and $(\sigma^{(k)})'$ is its derivative, $w_{i,j}^{(k)[\tau]}$, $k = 1,\ldots,K$ denotes the weight of the link between the j-th element of the $(k-1)$-th layer to the i-th element of the k-th layer, $X_{i,p}^{(k)[\nu,\tau]}$, $p = 0,\ldots,M_1$ are auxiliary values (internal sums of inputs to the neuron and its derivatives).

For the input and output layers there exsist some obvious substitutions

- for the input layer:

$$x_{i,0}^{(0)[\nu,\tau]} = \tilde{t}_i^{[\nu]}, \quad x_{i,p}^{(0)[\nu,\tau]} = \begin{cases} 1 & \text{if } i = p, \\ 0 & \text{if } i \neq p, \end{cases} \quad i,p = 1,\ldots,M_1 \tag{12}$$

- for the output layer:

$$z_i^{[\tau]}\left(\tilde{t}^{[\nu]}\right) = x_{i,0}^{(K)[\nu,\tau]}, \quad \frac{\partial z_i^{[\tau]}}{\partial t_p}\left(\tilde{t}^{[\nu]}\right) = x_{i,p}^{(K)[\nu,\tau]} \tag{13}$$

The gradient of the target function is back-propagated with the use of the following formulae:

$$E_{i,p}^{(k)[\nu,\tau]} := \frac{\partial s^{[\tau]}}{\partial X_{i,p}^{(k)[\nu,\tau]}}$$

$$= \begin{cases} e_{i,0}^{(k)[\nu,\tau]} \left(\sigma^{(n)}\right)' \left(X_{i,0}^{(k)[\nu,\tau]}\right) \\ \quad + \sum_{\lambda=1}^{M_1} e_{i,\lambda}^{(k)[\nu,\tau]} \left(\sigma^{(n)}\right)'' \left(X_{i,0}^{(k)[\nu,\tau]}\right) X_{i,\lambda}^{(k)[\nu,\tau]} & \text{for } p = 0 \\ e_{i,p}^{(k)[\nu,\tau]} \left(\sigma^{(n)}\right)' \left(X_{i,0}^{(k)[\nu,\tau]}\right) & \text{for } p = 1,\ldots,M_1 \end{cases} \tag{14}$$

$$e_{i,p}^{(k)[\nu,\tau]} := \frac{\partial s^{[\tau]}}{\partial x_{i,p}^{(k)[\nu,\tau]}} = \sum_{i=1}^{n^{(k+1)}} E_{i,p}^{(k+1)[\nu,\tau]} w_{i,j}^{(k)[\nu,\tau]} \quad \text{for } 0 < k < K \tag{15}$$

From (14) and (15) we get

$$e_{i,0}^{(K)[\nu,\tau]} = x_{i,0}^{(K)[1,\tau]} - \tilde{y}_i^{[\nu]} + \sum_{j=1}^{n^{(K)}} \frac{\partial f_j}{\partial y_i} \left(\tilde{t}^{[\nu]}, x^{(K)[\nu,\tau]}\right) f_j \left(\tilde{t}^{[\nu]}, x^{(K)[\nu,\tau]}\right) \tag{16}$$

and

$$e_{i,p}^{(K)[\nu,\tau]} = \sum_{j=1}^{n^{(K)}} \frac{\partial f_j}{\partial \left((y_i)'_{t_p}\right)} \left(\tilde{t}^{[\nu]}, x^{(K)[\nu,\tau]}\right) f_j \left(\tilde{t}^{[\nu]}, x^{(K)[\nu,\tau]}\right) \tag{17}$$

where (16) is for $\nu \leq N_1$ and $p = 0$, whereas (17) for the other values of ν and p. In both the equations $x^{(K)[\nu,\tau]}$ denotes the matrix of the outputs from the last layer of the network:

$$x^{(K)[\nu,\tau]} = \left[x_{i,p}^{(K)[\nu,\tau]}\right]_{\substack{i=1,\ldots,n^{(K)} \\ p=0,\ldots,M_1}}$$

$$d_{i,j}^{(k)[\tau]} := \frac{\partial s^{[\tau]}}{\partial w_{i,j}^{(k)[\tau]}} = \begin{cases} \sum\limits_{\nu=1}^{N_2} E_{i,0}^{(k)[\nu,\tau]} & \text{for } j = 0 \\ \sum\limits_{\nu=1}^{N_2} \sum\limits_{p=0}^{M_1} E_{i,p}^{(k)[\nu,\tau]} x_{j,p}^{(k)[\nu,\tau]} & \text{for } j = 1, \ldots, n^{(k-1)} \end{cases} \tag{18}$$

To compute these quantities, some approaches tend to complicate the structure of the feed-forward network in order to extract the most atomic operational elements and to comprise all the dependencies in the structure of neural links. We suggest another approach: signals passing from neuron to neuron can be treated not as scalar real values but as aggregates of $M_1 + 1$ real values. In this approach, except for the "size" of its signals, the structure of the network remains the same as for usual perceptrons. The surplus of complexity is encapsulated into neurons (see Fig. 1). Thus the output of the i-th neuron of the k-th layer is the vector $x_i^{(k)[\nu,\tau]} = [x_{i,p}^{(k)[\nu,\tau]}]_{p=0,\ldots,M_1}$ and the input is the matrix

$$x^{(k-1)[\nu,\tau]} = \left[x_{j,p}^{(k-1)[\nu,\tau]}\right]_{\substack{j=1,\ldots,n^{(k-1)} \\ p=0,\ldots,M_1}}$$

The backpropagated signals create vectors and matrices, as well.

Having a network which computes derivatives of its outputs, we can also extend the SA method by teaching not only values of the solution but also formerly computed values of the derivatives of the solution. It is shown in (Hornik *et al.*, 1990) that for any mapping there exists a perceptron which approximates this mapping and its derivatives with arbitrary precision.

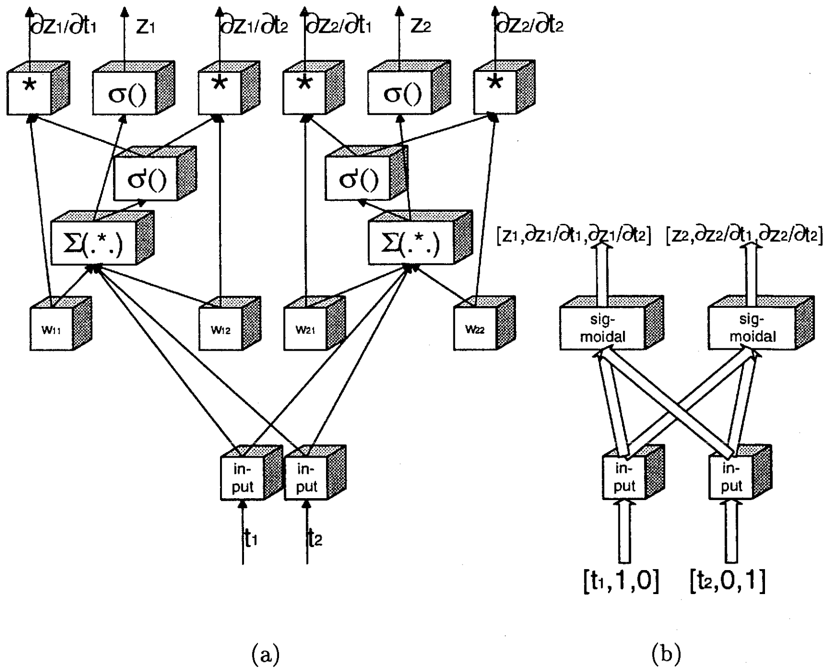

Fig. 1. Neural networks computing derivatives: (a) a network using scalar signals (Hornik *et al.*, 1990), (b) a network using aggregated signals. The label 'sigmoidal' on a neuron means that it has a sigmoidal activation function, but having aggregate inputs, it calculates its outputs using formulae (8)–(11), and back-propagates the gradient using (14), (15) and (18). The neurons labelled as 'input' perform an identical mapping. Neurons labelled as $w_{i,j}$ are the only ones which are allowed to contain weights (exactly one each).

Using one of the networks described in this subsection together with the formula of the target function defined as in (7), we obtain a method for solving differential equations. In contrast to the SA method, it is not only able to approximate solutions for non-trained parameters, but it solves the equation as well. Let us call this method the SGA method (Solutions Generating then Approximating method).

If the equation and the initial condition are not dependent on any parameter, the SGA method solves only the differential equation and we will call this reduced form of the SGA method the SG method.

4. Numerical Example

One of the considered differential equations was the heat equation

$$\frac{\partial y}{\partial t_1} - \frac{\partial^2 y}{\partial t_2^2} = \varphi(t_1, t_2) \tag{19}$$

for $\varphi(t_1, t_2) \equiv 0$. The equation was reduced to the system of equations with first-order derivatives by using the substitutions $y_1 = y$ and $y_2 = \partial y / \partial t_2$:

$$\begin{cases} \dfrac{\partial y_1}{\partial t_1} - \dfrac{\partial y_2}{\partial t_2} = 0 \\ y_2 - \dfrac{\partial y_1}{\partial t_2} = 0 \end{cases} \tag{20}$$

The system of equations was considered in the domains $t = [t_1, t_2]^T \in D_t = \langle 0, 1 \rangle^2$, $y = [y_1, y_2]^T \in D = \mathbb{R}^2$. Our system was combined with boundary conditions dependent on a parameter $a \in D_a = \langle 0, 1 \rangle$:

$$\bigwedge_{t_2 \in \langle 0,1 \rangle} y_1(0, t_2) = 0, \qquad \bigwedge_{t_1 \in \langle 0,1 \rangle} y_1(t_1, 0) = a t_1, \qquad \bigwedge_{t_1 \in \langle 0,1 \rangle} y_1(t_1, 1) = 0 \tag{21}$$

Equation (19) was solved using the method of straight lines. The results were used to build a training sequence $\{([t_1^{[\nu]}, t_2^{[\nu]}, a^{[\nu]}]^T, \tilde{y}^{[\nu]}\}_{\nu=1,\ldots,495}$ for the SA method. The input patterns $[t_1^{[\nu]}, t_2^{[\nu]}, a^{[\nu]}]^T$, $\nu = 1, \ldots, 495$ were taken from $\{0, 1/10, \ldots, 1\} \times \{0, 1/8, \ldots, 1\} \times \{1/5, 2/5, \ldots, 1\}$. The trained perceptron had one hidden layer which contained ten sigmoidal neurons and its output layer contained one linear neuron. All the signals were scalars.

The system (20), (21) was used to train a network with aggregated signals (SGA method). The second network had three inputs, ten hidden sigmoidal and two linear output elements. The dimension of signals was three. The sampling sequence was like the learning sequence in the former case. The terms lying on the boundary for which conditions (21) could be computed created the learning subsequence.

Finally, the third network with the architecture like that of the second one was trained with the SA method by using a learning sequence extended by the values of derivatives $\partial y_1 / \partial t_1$ and $\partial y_2 / \partial t_2$.

All the networks were tested on the sampling sequence containing 990 terms: $[t_1^{[\nu]}, t_2^{[\nu]}, a^{[\nu]}]^T \in \{0, 1/10, \ldots, 1\} \times \{0, 1/8, \ldots, 1\} \times \{1/10, 2/10, \ldots, 1\}$ for $\nu = 1, \ldots, 990$.

The results are presented in Table 1 and Figs. 2 and 3.

5. Conclusions

The paper shows how to obtain, using a feedforward nonlinear perceptron, the values of the solution to a given differential equation which can contain parameters for arbitrarily chosen arguments provided that they belong to a polyhedron stretched on the arguments of the training sequence.

Two ways of reaching this goal are described: one is to teach the network some solutions (and possibly their derivatives) for several parameters (the SA method), the other is to force the network to find the solution (the SG and SGA methods).

At present, we do not know the conditions which must be satisfied by the differential equation for the SGA method to be convergent to the solution of such an

Table 1. Summary of training. The predicate symbol '$\sim$' means here 'desired to approximate.'

	Classical perceptron after learning the solution (SA method)	Perceptron with aggregated signals forced to find the solution (SGA method)	Perc. with aggr. sig. after learning the solution and its derivatives (SA method)
Network's input	3 scalars: $x_1^{(0)} = t_1$, $x_2^{(0)} = t_2$, $x_3^{(0)} = a$	3 triples: $\boldsymbol{x}_1^{(0)} = [t_1, 1, 0]^T$, $\boldsymbol{x}_2^{(0)} = [t_2, 0, 1]^T$, $\boldsymbol{x}_3^{(0)} = [a, 0, 0]^T$	
Hidden layer	1 hidden layer consisting of 10 sigmoidal neural elements		
Network's output	1 scalar: $x_1^{(2)} \sim y$	2 triples: $\boldsymbol{x}_1^{(2)} \sim [y_1,\, \partial y_1/t_1,\, \partial y_1/t_2]^T$, $\boldsymbol{x}_2^{(2)} \sim [y_2,\, \partial y_2/t_1,\, \partial y_2/t_2]^T$	
Training seq. length	495	495	495
Training epochs	10 000	10 000	10 000
$s^{[10000]}$	0.024102	0.101338	0.118913
Test sequence length	990	990	990
Average squared error	0.009523	0.033279	0.019719
Maximal error	0.069093	0.193763	0.098182

equation. However, numerical examples show that the method finds a solution with the precision comparable to that of the SA method. In the future, it is necessary to derive conditions under which the sequence of mappings generated during training by the SGA method is convergent and, moreover, it has the solution to the differential equation as its limit.

The disadvantages of all the described neural methods are a long training time and problems with obtaining a satisfactory accuracy.

The following advantages are implied by the neural approach to the problem:

- a fast and parallelisable formula used by the perceptron to compute its outputs,
- a possibility of concurrent calculation of the values for several arguments presented parallelly to several neural networks having the same structure and weights,
- a possibility of obtaining solution values for a parameterized equation (or system) for a parameter for which the equation was not previously solved,
- when modelling a real process which changes its characteristics during its lifetime, the network can be trained between operation times to adjust its mapping to the process.

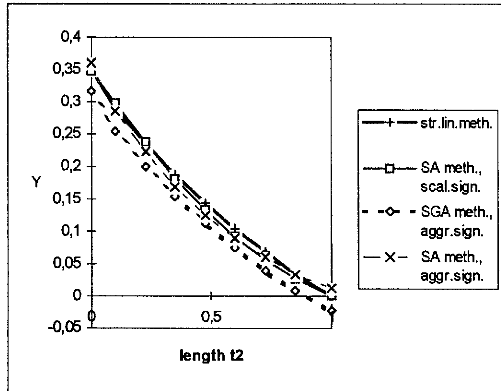

Fig. 2. Comparison of the trained mappings and the solution to the heat equation, $t_1 = 0.7$ (the parameter $a = 0.5$ was not in the training sequence).

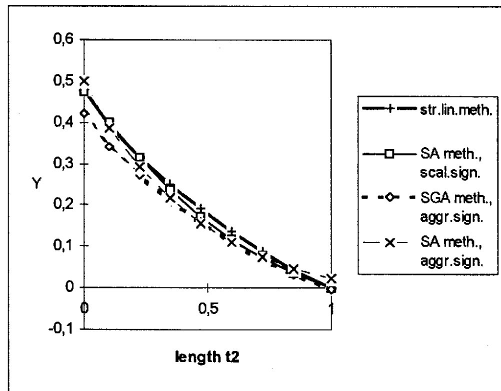

Fig. 3. Comparison of the trained mappings and the solution to the heat equation, $t_1 = 0.6$ (the parameter $a = 0.8$ was not in the training sequence).

References

Cybenko G. (1989): *Approximation by superpositions of a sigmoidal function.* — Math. Contr. Sign. Syst., Vol.2, No.4, pp.303–314.

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

Nerrand O., Roussel-Ragot P., Urbani D., Personnaz L. and Dreyfus G. (1994): *Training recurrent neural networks: Why and how? An illustration in dynamical process modelling.* — IEEE Trans. Neural Networks, Vol.5, No.2, pp.178–184.

Nikliborc W. (1951): *Differential Equations, Part I.* — Mathematical Monographies, Vol.XXV, Polish Math. Soc., Warsaw-Wroclaw (in Polish).

Szczepaniak P.S. and Lis B. (1994a): *Solution of differential equations using feed-forward artificial neural network.* — Proc. 6th Int. Congress *Computational and Applied Mathematics ICCAM*, Leuven, Belgium,

Szczepaniak P.S. and Lis B. (1994b): *Neural modelling of dynamic process.* — Proc. of the Workshop *Qualitative and Quantitative Approaches to Model-Based Diagnosis* of the 2nd Int. Conf. *Intelligent Systems Engineering*, Hamburg, Germany, pp.144–149.

Received: 7 February 1997