JOURNAL OF SCIENCE AND ARTS

A NUMERICAL SOLUTION FOR AN INITIAL VALUE PROBLEM FOR ODE USING ARTIFICIAL NEURAL NETWORK

EMIL LUNGU
Valahia University of Târgovişte, Bd. Unirii 18, 130056 Târgovişte, Romania,
e-mail: *emil.lungu@valahia.ro*

Abstract: *Artificial Neural Networks (ANN) proved to be good tools for functions approximation. Starting from this idea the article presents a method for approximating the solution of an initial value problem for ordinary differential equation (ODE) using ANN. The performance function associated to the neural network is built in such a way that the training process to produce at the output of the network a function that satisfies the differential equation on the training set and also the initial condition.*

Keywords: artificial neural networks, performance and approximation functions, lsqnonlinear

1 Introduction

Research in the field of artificial neural networks has increased in the last years since their various applications. Among the large number of application we may recall the hand-writing recognition, voice recognition, human face recognition in people crowd. Also, neural networks have important applications in medicine, economy and meteorology (eg. short term predictions).

The beginnings of research in ANN domain date since 1943 when Warren Mc Culloch and Walter Pitts presented the first model of artificial neurons. Since that time the ANN domain has developed continuously, different structure models and training algorithms being proposed.

An artificial neural network is an attempt at imitating the nervous system present in the biological organism. In essence an ANN consists from many processing units which are interconnected in a complex network designed for transporting the information from the inputs to some outputs. In analogy with a real biological neuron, each unit, known as artificial neuron is characterized by a number of inputs that bring informations, and only one output which may be distributed to other neurons. Each input is affected by multiplication with a quantity called weight. Also, we may consider an extra input for each neuron having the value equal to 1. Its weight is called bias.

Figure (1) represents an artificial neuron. Its inputs are denoted by $in_k, k = 1, n$ and their correspondent weights are denoted by $w_k, k = 1, n$. The weighted inputs represent impulses which added together activate the neuron to transmit the information further. The activation function, also called transfer function is chosen accordingly with the problem to be solved. There are some standard choices like linear transfer function, log-sigmoid, positive linear transfer function, tan-sigmoid, radial basis function, triangular basis function. For our purpose we use log-sigmoid and linear transfer functions. In mathematical form the output

Paper presented at The VI-th International Conference on Nolinear Analysis and Applied Mathematics (ICNAAM), Târgovişte, 21-22 nov, 2008

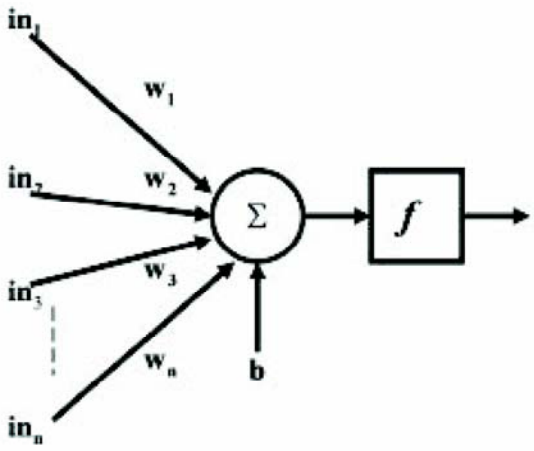


Figure 1: Artificial neural network

of a neuron has the folowing form:

$$a = f(\sum_{k=1}^{n} w_k \cdot in_k + b) \tag{1}$$

A typical interconnecting model of the neurons is the feed forward network. In this kind of structure the neurons are grouped in a sequence of layers and there are allowed only forward connections among neurons on consecutively layers. The first layer is called input layer and the last is named as output layer. The layers in between are called hidden layers. Information flows in such a network from the input layer to the output layer passing only once through each neuron. Neurons situated on the input layer do not have any activation function associated. Their role is only to distribute the information to the neurons on the first hidden layer.

An usual training process of an ANN suppose to update the weights such that for the training input data to have at the output of the network some specific values. In fact it is built a performance function that measures how well the outputs of the network correspond to the expected values. The training process is nothing else than an optimization process which update the weights of the neural network such that the difference between the expected values and the obtained ones to be as small as possible. In our case we construct the performance function in such a way that the differential equation to be satisfied in a set of points used for training and also the initial condition to be accomplished.

2 Description of the method

One of the most important application of neural networks is the functions approximation. It is well known that feedforward neural networks have capacity of approximating continuous functions to any desired accuracy using only few hidden layers. Once a neural network has been trained it also has good extrapolating properties.

Starting from this idea we try to obtain an approximation of the solution of an initial problem as the output of a feedforward neural network. The weights will be updated step by step in a training process that involve a certain performance function.

In [1] it is proposed a method that construct the approximate solution of an initial problem as a sum of two terms, one that satisfies the initial condition and another that is obtained as the output of a neural network and that satisfies the differential equation in a set of points.

In this article we do not split the solution in two terms. The initial condition is also part of the performance function. Also the training process for updating the weights does not

involve the partial derivative of the performance function in respect to the weights of the neural network. Numerical examples will show how the method works.

For the sake of an easy introduction we consider the following initial problem:

$$\begin{cases} \frac{dy}{dx}(x) = g(x,y), & \forall x \in [a,b] \\ y(a) = y_a \end{cases} \tag{2}$$

but the method allows also an implicit form of the differential equation (that is we may also have $F(x,y,y') = 0$). As in [1] we will consider a feedforward neural network with only one hidden layer, and the input and output layers will have only one neuron. The activation function for the neurons in the hidden layer will be chosen the sigmoid function that is defined by

$$f(x) = \frac{1}{1+e^{-x}}$$

while the transfer function for the output will be the linear function $f(x) = x$.

Figure (2) presents the structure of the neural network.

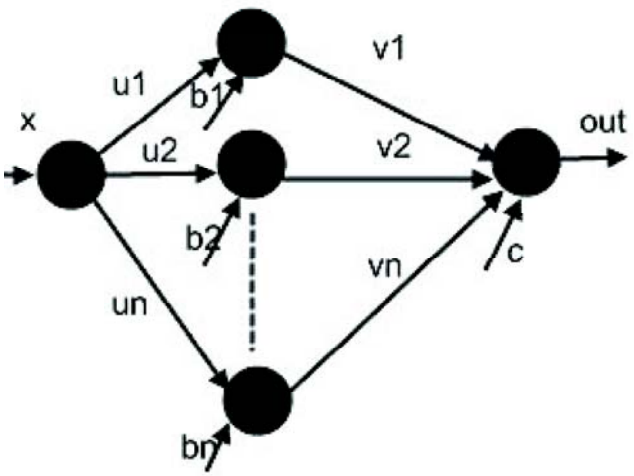


Figure 2: Artificial neural network

Taking into account the way a single neuron works we see easily that the output of our neural network is given explicitly by the formula

$$out(x) = \sum_{i=1}^{N} v_i \cdot f(u_i x + b_i) + c$$

where N is the number of hidden neurons , u_i is the weight of the connection between the input neuron and the $i-th$ hidden neuron,b_i is the corresponding bias, v_i is the weight of the connection between $i-th$ hidden neuron and the output neuron, and c is the bias of the output neuron.

Consider now a set of points $\{x_k \mid k = 1,m\} \subset [a,b]$ and construct the performance function as

$$E(\underline{w}) = \sum_{k=1}^{m} \left(\frac{\mathrm{d}out}{\mathrm{d}x}(x_k) - f(x_k, out(x_k)) \right)^2 + (out(a) - y_a)^2$$

where $\underline{w} = (u_1, \ldots, u_N, v_1, \ldots, v_N, b_1, \ldots, b_N, c)$

To compute the performance function we need to compute the network output and the derivative of this output with respect to the input in the set of considered points. Here the values of the network weights are considered fixed until they are updated.

In this simple case

$$\frac{\mathrm{d}out}{\mathrm{d}x}(x) = \sum_{i=1}^{N} v_i \cdot u_i \cdot f'(u_i x + b_i) = \sum_{i=1}^{N} v_i \cdot u_i \cdot f(u_i x + b_i)(1 - f(u_i x + b_i))$$

but it gets more complicated in the case of more hidden layers. Even then, taking into account the formula (1) rewritten as $a(x) = f(\sum_{k=1}^{n} w_k \cdot in_k(x) + b)$ for a certain neuron and observing that

$$a'(x) = \left(\sum_{k=1}^{n} w_k \cdot in'_k(x)\right) \cdot f'\left(\sum_{k=1}^{n} w_k \cdot in_k(x) + b\right)$$

we are still able to compute the derivative of the network output the same way we compute the network output. The calculus is performed sequentially from the input layer to the output layer.

With the above introduction the training process is nothing else than a minimization problem. We have to find the vector $\underline{w}$ that is a minimum point for the performance function. For this purpose we used "lsqnonlin" Matlab function that require the function to minimize and an initial approximation $\underline{w}^{(0)}$.

3 Details of implementation

The solution proposed in this article was entirely implemented in Matlab. Function "lsqnonlin" is optimized to find the minimum of a function of the form $F(\underline{w}) = \sum_{k=1}^{p} f_k^2(\underline{w})$. It is recommended to implement a vectorial function of the form $G(\underline{w}) = (f_1(\underline{w}), \ldots, f_p(\underline{w}))$ instead of providing the scalar function F given above.

In our case function G has the following form

$$G(\underline{w}) = \begin{pmatrix} \frac{\mathrm{d}out}{\mathrm{d}x}(x_1) - f(x_k, out(x_1) \\ \frac{\mathrm{d}out}{\mathrm{d}x}(x_2) - f(x_k, out(x_2) \\ \vdots \\ \frac{\mathrm{d}out}{\mathrm{d}x}(x_m) - f(x_k, out(x_m) \\ out(a) - y_a \end{pmatrix}$$

As we have already said the function "lsqnonlin" requires an initial approximation to start the calculus. To speed up the calculus we have computed a rough approximation of the minimum using a genetic algorithm. In Matlab this is done using the function "ga".

Remark 3.1. *The proposed solution do not compute the gradient of the performance function with respect to the weights. This will result in a slower convergence of the process. Providing the jacobian of the function G to "lsqnonlin" will improves the speed of convergence at an extra cost of computing this jacobian.*

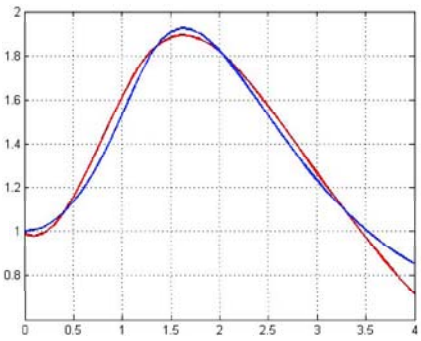
4 Numerical results

Example 1 *In this example we consider the nonlinear initial value problem*

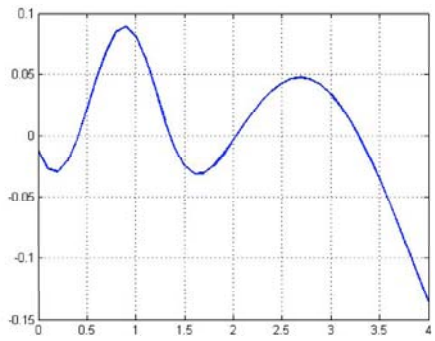
$$\begin{cases} y'(x) = sin(x \cdot y(x)), \quad x \in [0, 3] \\ y(0) = 1 \end{cases}$$

To apply our method we considered a hidden layer with 10 neurons and the performance function was constructed to satisfy the differential equation in 10 equally spaced points in the problem domain and also to satisfy the initial condition . The first approximation of the

network weights was obtained by a genetic algorithm. For this approximation the performance function had the value equal to 0.2223. After using "lsqnonlin" the performance function got the value $4.1956 \cdot 10^{-4}$. *Figure (3a)) shows the solution obtained with our method after applying the genetic algorithm (red color) and also the solution obtained with Matlab function "ode45" that implements the adaptive Runge-Kutta method. In the figure on the right right it is represented the difference between the two solutions.*



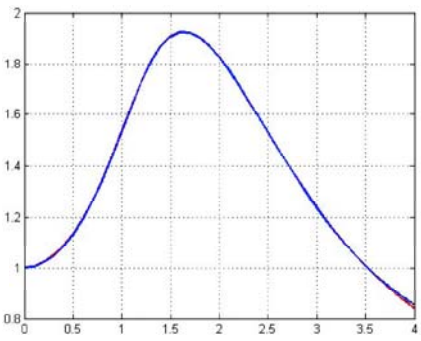
(a) Neural network and Runge-Kutta solutions



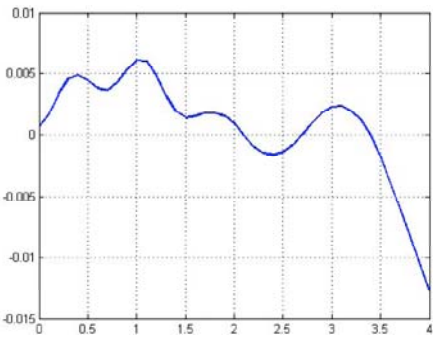
(b) Error

Figure 3: Comparison between neural network and Runge-Kutta solutions for the initial approximation obtained with genetic algorithm

Starting from the above solution and applying "lsqnonlin" we obtain the results in the figure (4).



(a) Neural network and Runge-Kutta solutions



(b) Error

Figure 4: Comparison between neural network and Runge-Kutta solutions

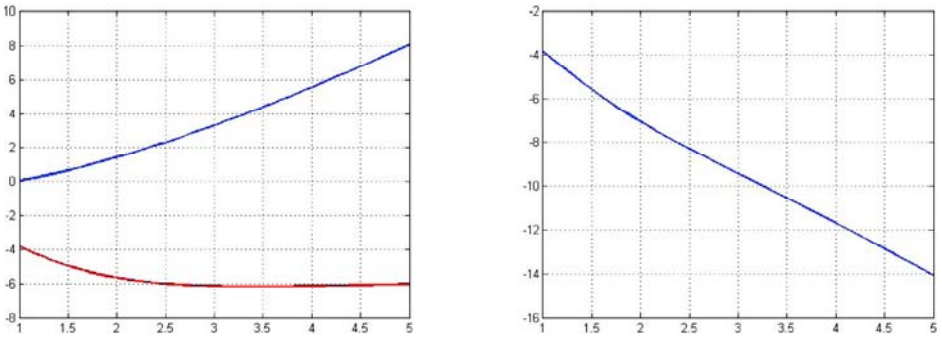
Both figures represent the solution of the equation extended to the interval $[0, 4]$. *This shows that the neural network has also good extrapolation capabilities.*

Example 2 *The presented method has the advantage that it can solve differential equation in implicit form* $F(x, y, y') = 0$ *while Runge Kutta method and others require an explicit form* $y' = f(x, y)$. *In this example we consider the following problem:*

$$\begin{cases} xy'(x) - y(x) + e^{y'(x)-1} - 2 \cdot x = 0, & x \in [1, 4] \\ y(1) = 0 \end{cases}$$

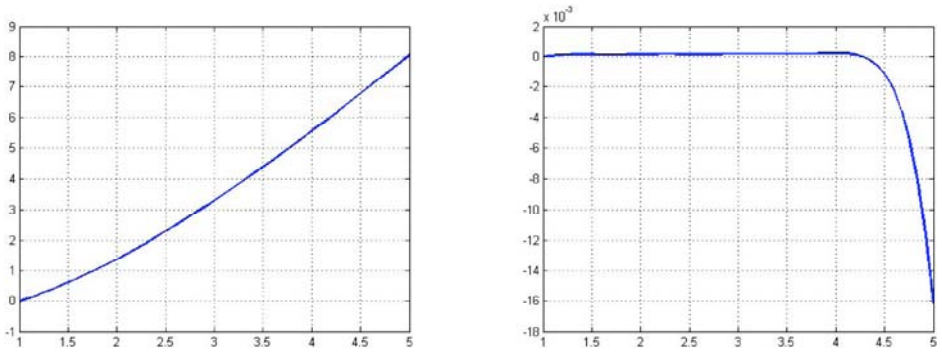
having the exact solution $y(x) = x \cdot ln(x)$ *We used the same structure of the neural network and also 10 equally spaced nodes in* $[1, 4]$. *The initial solution produces a value for the performance function equal to 18.1003. After using the optimization function "lsqnonlin"*

this value become $4.1927 \cdot 10^{-7}$. *Figures (5) and (6) are similar to those one in first example and they show again the accuracy of the numerical solution. Since the solution is represented on* $[0, 5]$ *we see that the neural network has again good extrapolation capabilities.*



(a) Neural network and exact solutions (b) Error

Figure 5: Comparison between neural network and exact solutions for the initial approximation obtained with genetic algorithm



(a) Neural network and exact solutions (b) Error

Figure 6: Comparison between neural network and exact solutions

References

[1] Lagaris, I.E., Likas, A., Fotiadis, D.I., *Artificial Neural Networks for Solving Ordinary and Partial Differential Equation*
[2] Rojas R., *Neural Networks. A Systematic Introduction*, Springer Verlag Berlin, 1996
[3] Kincaid, D., Cheney, W., *Numerical analysis*, Brooks/Cole Publishing Company, 1991

Manuscript received: 07.07.2009 / accepted: 05.10.2009