Euro. Jnl of Applied Mathematics: page 1 of 37
doi:10.1017/S0956792517000146

Solving polynomial systems using a fast adaptive back propagation-type neural network algorithm†

K. GOULIANAS[1], A. MARGARIS[2], I. REFANIDIS[3] and K. DIAMANTARAS[1]

[1]*ATEI of Thessaloniki, Department of Information Technology, GR 574 00, Thessaloniki, Greece*
emails: gouliana@it.teithe.gr, kdiamant@it.teithe.gr
[2]*TEI of Thessaly, Department of Computer Science and Engineering, GR 411 10, Larissa, Greece*
email: amarg@uom.gr
[3]*Department of Applied Informatics, University of Macedonia, GR 540 06, Thessaloniki, Greece*
email: yrefanid@uom.gr

(*Received 28 November 2016; revised 26 May 2017; accepted 27 May 2017*)

This paper proposes a neural network architecture for solving systems of non-linear equations. A back propagation algorithm is applied to solve the problem, using an adaptive learning rate procedure, based on the minimization of the mean squared error function defined by the system, as well as the network activation function, which can be linear or non-linear. The results obtained are compared with some of the standard global optimization techniques that are used for solving non-linear equations systems. The method was tested with some well-known and difficult applications (such as Gauss–Legendre 2-point formula for numerical integration, chemical equilibrium application, kinematic application, neuropsychology application, combustion application and interval arithmetic benchmark) in order to evaluate the performance of the new approach. Empirical results reveal that the proposed method is characterized by fast convergence and is able to deal with high-dimensional equations systems.

Key words: Neural networks, polynomial systems, numerical analysis

1 Introduction and review of previous papers

Polynomial systems of equations are of major interest and they are heavily used in any discipline of sciences such as mathematics, physics and engineering. The last decades a lot of algorithms have been developed for solving such systems (see, for example, [14] and [15] as well as [24], [35] and [34]). According to [21], the approaches for solving polynomial systems of equations can be classified in three main categories as follows:

(1) Symbolic methods that stem from algebraic geometry and are able to perform variable elimination. However, the currently available methods are efficient only for sets of low-degree systems of polynomials.

† The research of K. Goulianas, A. Margaris, I. Refanidis, K. Diamantaras and T. Papadimitriou has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) – Research Funding Program: THALES, Investing in knowledge society through the European Social Fund.

(2) Numerical methods that are mainly based on iterative procedures. These methods are suitable for local analysis only and perform well only if the initial guess is good enough, a condition that generally is rather difficult to satisfy.

(3) Geometric methods such as subdivision-based algorithms for intersection and ray tracing of curves and surfaces. They are characterized by slow convergence and they have limited applications.

Since polynomial systems can be considered as generalizations of systems of linear equations, it is tempting to attempt to solve them, using the well-known iterative procedures of numerical linear algebra, after their appropriate modification. The methods described by Ortega and Rheinboldt [30, 31] are examples of such an approach. On the other hand, the so-called continuation methods [23] begin with a starting system with a known solution that it is gradually transformed to the non-linear system to be solved. This is a multistage process, in which, at each stage, the current system is solved by Newton-type methods to identify a starting point for the next stage system, since, as the system changes, the solution is moved on a path that joins a solution of the starting system with the solution of the desired system.

There are two other efficient approaches for solving polynomial systems of equations as well as non-linear algebraic systems of any type, based on the use of neural networks and genetic algorithms. Since the proposed method is a neural based one, the neural-based methods are described in a lot of detail, while the description of the genetic algorithm approaches is shorter enough. The main motivation for using neural networks in an attempt to solve non-linear algebraic systems, is that neural networks are universal approximators in the sense that they have the ability to simulate any function of any type with a predefined degree of accuracy. Towards this direction, Nguyen [28] proposed a neural network architecture capable of implementing the well-known Newton–Raphson algorithm for solving multivariate non-linear systems using the iterative equation $\boldsymbol{x}^p = \boldsymbol{x}^{p-1} + \Delta\boldsymbol{x}^p$ in the pth step with $\boldsymbol{J}(\boldsymbol{x}^{p-1})\Delta\boldsymbol{x}^p = -f(\boldsymbol{x}^{p-1})$ where $\boldsymbol{x}$ is the solution vector of the appropriate dimensions, and $\boldsymbol{J}$ is the Jacobian matrix. Nguyen defined the quadratic form

$$E^p = [\boldsymbol{J}(\boldsymbol{x}^{p-1})\Delta\boldsymbol{x}^p + f(\boldsymbol{x}^{p-1})]^t[\boldsymbol{J}(\boldsymbol{x}^{p-1})\Delta\boldsymbol{x}^p + f(\boldsymbol{x})^{p-1}]$$

characterized by a zero minimal value (the notation $\boldsymbol{A}^t$ represents the transpose of the matrix $\boldsymbol{A}$) and a gradient in the form

$$\frac{\partial E^p}{\partial \Delta\boldsymbol{x}^p} = 2[\boldsymbol{J}(\boldsymbol{x}^{p-1})]^t J(\boldsymbol{x}^{p-1})\Delta\boldsymbol{x}^p + 2[\boldsymbol{J}(\boldsymbol{x}^{p-1})^t]f(\boldsymbol{x}^{p-1}).$$

The proposed neural network that deals with this situation uses $\Delta\boldsymbol{x}^p$ as the output vector and satisfy the equation:

$$\frac{d\Delta\boldsymbol{x}^p}{dt} = k\frac{\partial E^p}{\partial \Delta\boldsymbol{x}^p},$$

where k is a negative scalar constant. The stability of this system is guaranteed by the fact that its Hessian matrix is a positive definite matrix. The proposed structure is a feed-forward two-layered neural network in which the weighting coefficients are the

elements of the transpose of the Jacobian matrix. The output of this network implements the vector-matrix product $[\boldsymbol{J}(\boldsymbol{x}^{p-1})]^t J(\boldsymbol{x}^{p-1})\Delta \boldsymbol{x}^p$. Each neuron is associated with a linear processing function while the total number of processing nodes is equal to the dimension N of the non-linear system.

On the other hand, Mathia & Saeks [25] solved non-linear equations using recurrent neural networks (and more specifically linear Hopfield networks) in conjunction with multilayered perceptrons that are trained first. The multilayered perceptrons they used, are composed of an input layer of m neurons, a hidden layer of q neurons and an output layer of n neurons with the matrix $\boldsymbol{W}_1$ to contain the synaptic weights between the input and the hidden layer and the matrix $\boldsymbol{W}_2$ to contain the synaptic weights between the hidden and the output layer. The neurons of the hidden layer use the sigmoidal function, while the neurons of the output layer are linear units. This Multi-Layered Perceptron (MLP) can be viewed as a parameterized non-linear mapping in the form:

$$f : R^m \to R^n \quad \text{with} \quad \boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{W}_2 \cdot \boldsymbol{g}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{w}_b) = \boldsymbol{W}_2 \cdot \boldsymbol{g}(\boldsymbol{\alpha}),$$

where $\boldsymbol{\alpha} = \boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{w}_b$ and

$$\boldsymbol{g} : R^q \to R^q \quad \text{with} \quad \boldsymbol{z} = \boldsymbol{g}(\boldsymbol{\alpha}) = [g(\alpha_1), g(\alpha_2), \ldots, g(\alpha_q)]^T$$

is the hidden layer representative function. On the other hand, the linear Hopfield network has the well-known recurrent structure with a constant input $\boldsymbol{x}$ and state $\boldsymbol{y}$ and in this work it uses linear activation functions instead of the commonly used non-linear functions (such as the step of sigmoidal function). Mathia and Saeks used this structure to implement the Newton method defined above that guarantees a quadratic convergence, given a sufficiently accurate initial condition $\boldsymbol{x}^0$ and a non-singular Jacobian matrix $\boldsymbol{J}$ for all the iteration steps. In this approach, the Jacobian of the MLP is obtained via the chain rule:

$$\boldsymbol{J} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}} \cdot \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\alpha}} \cdot \frac{\partial \boldsymbol{\alpha}}{\partial \boldsymbol{x}} = \boldsymbol{W}_2 \cdot [\boldsymbol{G}(\boldsymbol{\alpha}(\boldsymbol{I} - \boldsymbol{G}(\boldsymbol{\alpha})))]_{\boldsymbol{\alpha}=\boldsymbol{\alpha}_0} \cdot \boldsymbol{W}_1.$$

It is clear that since this method requires the inversion of the Jacobian matrix, this matrix must be a square matrix and therefore only the case $m = n$ is supported by this architecture. Another prerequisite for the method to work is that the number of hidden neurons must be greater than the number of input and output neurons. Based on all these evidences, Mathia and Saeks defined the recurrent network as

$$\boldsymbol{x}^{p+1} = \boldsymbol{x}^p - \alpha(\boldsymbol{W}_2(\boldsymbol{G}_p(\boldsymbol{I} - \boldsymbol{G}_p))\boldsymbol{W}_1)^{-1} \times (\boldsymbol{W}_2\boldsymbol{g}(\boldsymbol{W}_1\boldsymbol{x}^p + \boldsymbol{w}_b) - \boldsymbol{y}),$$

where $\boldsymbol{G}_p = \boldsymbol{G}(\boldsymbol{\alpha}_p)$ is a constant diagonal matrix. The solution of the non-linear equation can now be estimated via an iterative procedure that terminates when a predefined tolerance value is reached by the network.

To evaluate the performance of the proposed method for various system dimensions, it is compared to four well-known methods found in the literature. The first one of them is the Newton's method [4] that allows the approximation of the function $\boldsymbol{F}(\boldsymbol{x})$ by its first order Taylor expansion in the neighbourhood of a point $\boldsymbol{x} = (x_1, x_2, x_3, \ldots, x_n)^T \in R^n$.

This is an iterative method that uses as input an initial guess

$$\boldsymbol{x}_0 = [x_1(0), x_2(0), x_3(0), \ldots, x_n(0)]^T \tag{1.1}$$

and generates a vector sequence $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k, \ldots\}$, with the vector $\boldsymbol{x}_k$ associated with the kth iteration of the algorithm, to be estimated as

$$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} - \boldsymbol{J}(\boldsymbol{x}_{k-1})^{-1}\boldsymbol{F}(\boldsymbol{x}_{k-1}), \tag{1.2}$$

where $\boldsymbol{J}(\boldsymbol{x}_k) \in R^{n\times n}$ is the Jacobian matrix of the function $\boldsymbol{F} = (f_1, f_2, \ldots, f_n)^T$ estimated at the vector $\boldsymbol{x}_k$. Note, that even though the method converges fast to the solution provided that the initial guess (namely the starting point $\boldsymbol{x}_0$) is a good one, it is not considered as an efficient algorithm, since it requires in each step the evaluation (or approximation) of n^2 partial derivatives and the solution of an $n \times n$ linear system. A performance evaluation of the Newton's method as well as other similar direct methods, shows that these methods are impractical for large-scale problems, due to the large computational cost and memory requirements. In this work, besides the classical Newton's method, the fixed Newton's method was also used. As it is well known, the difference between these variations is the fact that in the fixed method the matrix of derivatives is not updated during iterations, and therefore the algorithm uses always the derivative matrix associated with the initial condition $\boldsymbol{x}_0$.

An improvement to the classical Newton's method can be found in the work of Broyden [2] (see also [3] as well as [11] for the description of the secant method, another well-known method of solution), in which the computation at each step is reduced significantly, without a major decrease of the convergence speed; however, a good initial guess is still required. This prerequisite is not necessary in the well-known steepest descent method, which unfortunately does not give a rapidly convergence sequence of vectors towards the system solution. The Broyden's methods used in this work are the following:

- Broyden method 1. This method allows the update of the Jacobian approximation B_i during the step i in a stable and efficient way and is related to the equation

$$B_i = B_{i-1} + \frac{(\Delta_i - B_{i-1}\delta_i)\delta_i^T}{\delta_i^T \Delta_i}, \tag{1.3}$$

  where i and $i-1$ are the current and the previous steps of iterative algorithm, and furthermore we define, $\Delta_i = f(x_i) - f(x_{i-1})$ and $\delta_i = x_i - x_{i-1}$.
- Broyden method 2. This method allows the elimination of the requirement of a linear solver to compute the step direction and is related to the equation

$$B_i = B_{i-1} + \frac{(\delta_i - B_{i-1}\Delta_i)\delta_i^T B_{i-1}}{\delta_i^T B_{i-1}\Delta}, \tag{1.4}$$

  with the parameters of this equation to be defined as previously.

The last described neural network approach used for solving non-linear algebraic equations, is the modified Hopfield network model of Mishra and Kalra [27] that can solve a non-linear system of equations with n unknowns. This neural network is composed of n

product units whose outputs are linearly summed via synaptic weights. The state equation of each one of those neurons is formed by applying the well-known Kirchhoff's law in the electric circuit that represents the network and it is proven to have the form

$$C_i \frac{d\varphi^{-1}(x_i)}{dt} + \frac{\varphi^{-1}(x_i)}{R_i} = \sum_j w_{ij} f_{ij}(x_1, x_2, x_3, \ldots, x_n) + I_i$$

$(i, j) \in [1, 2, \ldots, n]$, where $\varphi(x) = 1/(1 - e^{-x})$ [1] [5] [8] [9] [16] [17] is the sigmoidal activation function of the n neurons. The energy function of the modified Hopfield network is defined as

$$E = \sum_i \int_0^{x_i} \frac{\varphi^{-1}(s)}{R_i} ds - \sum_i \sum_j w_{ij} f_{ij}(x_1, x_2, x_3, \ldots, x_n) x_i - \sum_i I_i x_i$$

and it is a bounded function since it can be proven that it satisfies the inequality $E'(t) \leqslant 0$ where $E'(t)$ is the derivative of the energy function $E(t)$.

In order to use this model for solving non-linear algebraic systems, this energy function is defined as the sum of the squares of the differences between the left- and the right-hand side of each equation, while the number n of neurons is equal to the number of the unknowns of the non-linear algebraic system under consideration. It can be proven that this system is stable in the Lyapunov sense and it is able to identify the solution vector of the above system of equations.

On the other hand, in the genetic algorithm approach, a population of candidate solutions (known as individuals or phenotypes) to an optimization problem is evolved towards better solutions. Each candidate solution is associated with a set of properties (known as chromosomes or genotypes) that can be mutated and altered. In applying this technique, a population of individuals is randomly selected and evolved in time-forming generations, whereas a fitness value is estimated for each one of them. In the next step, the fittest individuals are selected via a stochastic process and their genomes are modified to form a new population that is used in the next generation of the algorithm. The procedure is terminated when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached.

The application of genetic algorithm for solving non-linear algebraic systems allows the simultaneous consideration of multiple solutions due to their population approach [26] (see also [12] and [18]). Furthermore, since they do not use at all derivative information, they do not tend to get caught in local minima. However, they do not always converge to the true minimum, and in fact, their strength is that they rapidly converge to near optimal solutions. The most commonly used fitness function has the form $g = \max\{\text{abs}(f_i)\}$ $(i = 1, 2, 3, \ldots, n)$ where $\max\{\text{abs}(f_i)\}$ is the maximum absolute value of the individual equations of the system and n is the number of the system equations. The hybrid approach of Karr *et al.* [18] attempt to deal with the weakness of the classical Newton–Raphson method according to which when the initial guess $\boldsymbol{x}^0$ is not close to the root (or, even worse, it is completely unknown) the method does not behave well. In this case, a genetic algorithm can be used to locate regions in which roots of the system of equations are likely to exist. The fitness function is the function g defined above, while, the coding

schema for string representation is the well-known concatenated binary linearly mapped coding scheme.

The last work cited in this paper is the work of Grossan and Abraham [14] (see also [15]) that deal the system of non-linear equations using a modified line search approach and a multiobjective evolutionary algorithm that transfers the system of equations into an optimization problem.

Returning to the proposed neural-based non-linear system solver, the main idea is to construct a neural network-like architecture that can reproduce or implement the structure of the system of polynomials to be solved by assigning the values of the polynomial coefficients as well as the components of their solution $(x_1, x_2, x_3, \ldots, x_n)$ to fixed and variable synaptic weights in an appropriate way. If this network is trained such that its output vector is the null vector, then, by construction, the values of the variable synaptic weights will be the components of one of the system roots. This idea together with representative examples is explained completely in [22] (see also [13]).

2 Problem formulation

As it is well known from non-linear algebra, the structure of a typical non-linear algebraic system of n equations with n unknowns has the form:

$$\begin{aligned} &f_1(x_1, x_2, x_3, \ldots, x_n) = 0, \\ &f_2(x_1, x_2, x_3, \ldots, x_n) = 0, \\ &f_3(x_1, x_2, x_3, \ldots, x_n) = 0, \\ &\ldots\ldots\ldots\ldots\ldots\ldots \\ &f_n(x_1, x_2, x_3, \ldots, x_n) = 0, \end{aligned} \tag{2.1}$$

or, using vector notation, $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$, where

$$\boldsymbol{F} = (f_1, f_2, f_3, \ldots, f_n)^T \tag{2.2}$$

is a vector of non-linear functions

$$f_i(\boldsymbol{x}) = f_i(x_1, x_2, x_3, \ldots, x_n), \tag{2.3}$$

each one of them being defined in the vector space

$$\Omega = \prod_{i=1}^{n} \{\alpha_i, \beta_i\} \subset \mathbb{R}^n \tag{2.4}$$

of all real-valued continuous functions and

$$\boldsymbol{x} = (x_1, x_2, x_3, \ldots, x_n)^T \tag{2.5}$$

is the solution vector of the system. For non-linear polynomial systems, this system of equations can be written as

$$\begin{aligned}
f_1(x) &= a_{11}x_1^{e_{11}^1}x_2^{e_{21}^1}\cdots x_n^{e_{n1}^1} + a_{12}x_1^{e_{12}^1}x_2^{e_{22}^1}\cdots x_n^{e_{n2}^1}\cdots \\
&\quad + a_{1k_1}x_1^{e_{1k_1}^1}x_2^{e_{2k_1}^1}\cdots, x_n^{e_{nk_1}^1} - \beta_1 = 0, \\
f_2(x) &= a_{21}x_1^{e_{11}^2}x_2^{e_{21}^2}\cdots x_n^{e_{n1}^2} + a_{22}x_1^{e_{12}^2}x_2^{e_{22}^2}\cdots x_n^{e_{n2}^2}\cdots \\
&\quad + a_{2k_2}x_1^{e_{1k_2}^2}x_2^{e_{2k_2}^2}\cdots x_n^{e_{nk_2}^2} - \beta_2 = 0, \\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
f_n(x) &= a_{n1}x_1^{e_{11}^n}x_2^{e_{21}^n}\cdots x_n^{e_{n1}^n} + a_{n2}x_1^{e_{12}^n}x_2^{e_{22}^n}\cdots x_n^{e_{n2}^n}\cdots \\
&\quad + a_{nk_n}x_1^{e_{1k_n}^n}x_2^{e_{2k_n}^n}\cdots x_n^{e_{nk_n}^n} - \beta_n = 0,
\end{aligned}$$

or equivalently,

$$\begin{aligned}
f_1(x) &= a_{11}\prod_{\ell=1}^{n}x_\ell^{e_{\ell 1}^1} + a_{12}\prod_{\ell=1}^{n}x_\ell^{e_{\ell 2}^1},\ldots,+a_{1k_1}\prod_{\ell=1}^{n}x_\ell^{e_{\ell k_1}^1} = \sum_{j=1}^{k_1}\left(a_{1j}\prod_{\ell=1}^{n}x_\ell^{e_{\ell j}^1}\right) - \beta_1 = 0, \\
f_2(x) &= a_{21}\prod_{\ell=1}^{n}x_\ell^{e_{\ell 1}^2} + a_{22}\prod_{\ell=1}^{n}x_\ell^{e_{\ell 2}^2},\ldots,+a_{2k_2}\prod_{\ell=1}^{n}x_\ell^{e_{\ell k_2}^2} = \sum_{j=1}^{k_2}\left(a_{2j}\prod_{\ell=1}^{n}x_\ell^{e_{\ell j}^2}\right) - \beta_2 = 0, \\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
f_n(x &= a_{n1}\prod_{\ell=1}^{n}x_\ell^{e_{\ell 1}^n} + a_{n2}\prod_{\ell=1}^{n}x_\ell^{e_{\ell 2}^n},\ldots,+a_{nk_n}\prod_{\ell=1}^{n}x_\ell^{e_{\ell k_n}^n} = \sum_{j=1}^{k_n}\left(a_{nj}\prod_{\ell=1}^{n}x_\ell^{e_{\ell j}^n}\right) - \beta_n = 0,
\end{aligned}$$

or in a more compact form:

$$f_i(x) = \sum_{j=1}^{k_i}\left(a_{ij}\prod_{\ell=1}^{n}x_\ell^{e_{\ell j}^i}\right) - \beta_i = 0 \qquad (i = 1, 2, \ldots, n), \tag{2.6}$$

where in every exponent e_{jl}^i the superscript i denotes the equation, the first subscript j denotes the factor of the summation in equation i and the second subscript ℓ denotes the corresponding unknown x.

3 The architecture of the proposed neural non-linear system solver

The architecture of the neural network-like architecture that can solve a complete $n \times n$ system of polynomial equations, is characterized by four layers with the following structure:

- Layer 0 is the single input layer. This layer does not used at all to the back propagation training and it simply participates to the variable weight synapses whose values after training are the components of the system roots. In other words, in this procedure, there is not a training set since the simple input is always the value of unity while the associated desired output is the zero vector of n elements.

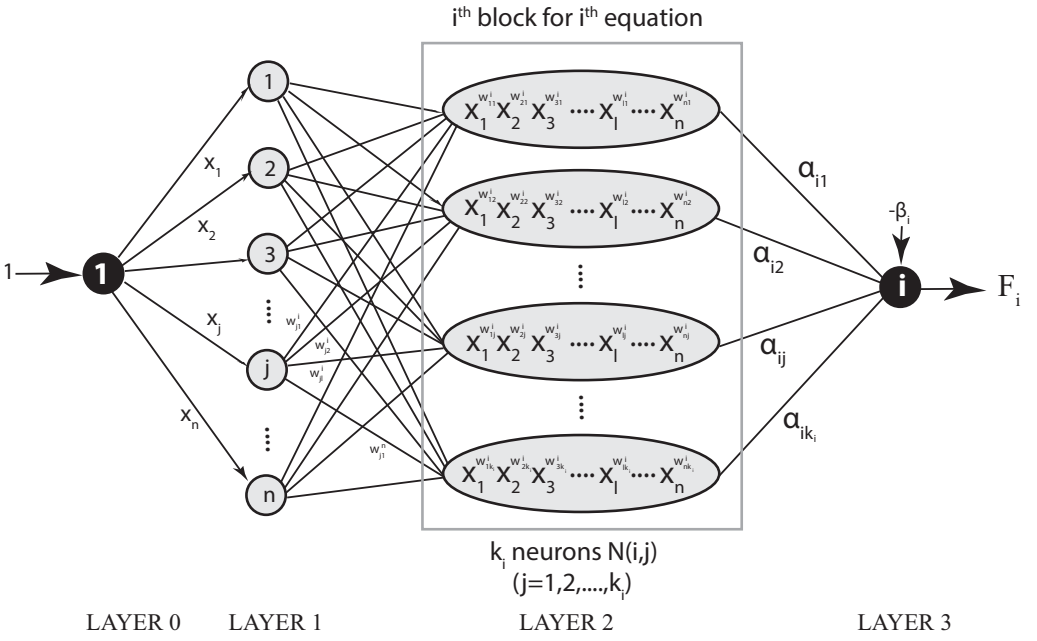

FIGURE 1. The structure of the ith block of Layer 2 and its connections with the Layers 1 and 3.

- Layer 1 contains n neurons each one of them is connected to the single input unit of the first layer. As it has been mentioned, the weights of the n synapses defined in this way, are the only variable weights of the network. During network training, their values are updated according to the equations of the back propagation algorithm and after a successful training these weights contain the n components of a system root

$$\boldsymbol{r} = (x_1, x_2, x_3, \ldots, x_n). \tag{3.1}$$

- Layer 2 is composed of n blocks of $\prod \boldsymbol{x}^{\boldsymbol{w}}$ neurons with the ℓth block containing k_ℓ neurons, namely, one neuron for each one of the k_ℓ products of powers of the x's associated with the ℓth equation of the non-linear system. The neurons of this layer, as well as the activation functions associated with them, are therefore described using the double index notation (ℓ, j) [for values $(\ell = 1, 2, \ldots, n)$ and $(j = 1, 2, \ldots, k_\ell)$]. The structure of the ith block of this layer as well as its connections with the Layers 1 and 3 are shown in Figure 1.

  In order to describe the Layer 2 neurons, the shorthand notation $N(i, j)$ is used, based on the fact that the total output of the jth neuron of the ith block is estimated as

$$x_1^{w_{j1}^i} x_2^{w_{j2}^i} x_3^{w_{j3}^i} \cdots x_\ell^{w_{j\ell}^i} \cdots x_n^{w_{jn}^i} = \prod_{\mu=1}^{n} x_\mu^{w_{j\mu}^i} = \boldsymbol{x}^{\boldsymbol{w}_{ij}}, \tag{3.2}$$

$(i = 1, 2, \ldots, n,\ \ j = 1, 2, \ldots, k_n)$. In the above expression, $\boldsymbol{x}^{\boldsymbol{w}_{ij}}$ is just a symbolic notation and does not describe some valid vector operation. To describe the fact that these neurons implement the product of the $\boldsymbol{x}$ components, each one of them is raised to

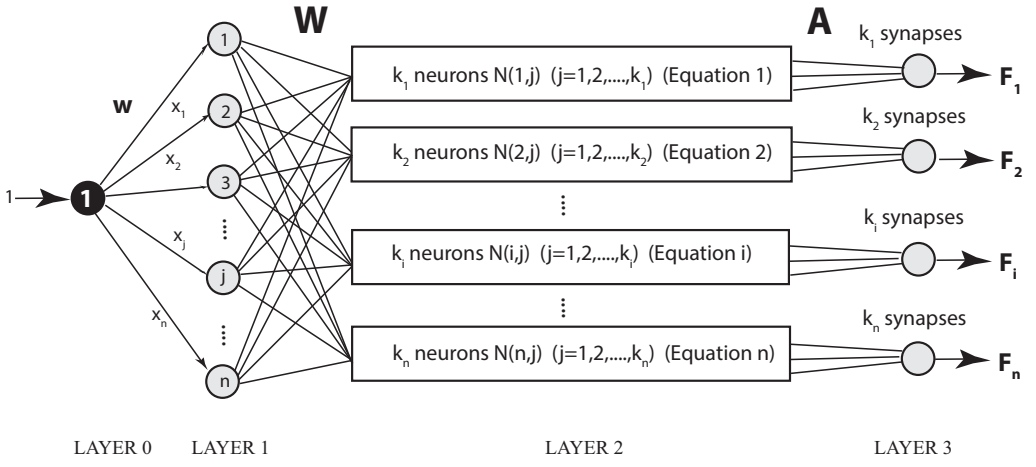

FIGURE 2. The complete architecture of the polynomial non-linear system solver.

the power of the corresponding weight component, we characterize these neurons as Π-power neurons.

- Layer 3 contains an output neuron for each equation, namely, a total number of n neurons that use the identity function $y = f(x) = x$ or the hyperbolic tangent function [6, 7, 10, 32] $y = f(x) = \tanh(x)$ as the activation function.

The complete neural network-like architecture is shown in Figure 2. Note, that each neuron n_i $(i = 1, 2, \ldots, n)$ of the output layer is associated with a bias unit with the weight of this bias synapse to has the value $-\beta_i$ where β_i is the fixed term of the ith non-linear polynomial equation. This bias synapse is not shown in the figure.

The matrix $\boldsymbol{w} = [w_1, w_2, \ldots, w_n] = [x_1, x_2, \ldots, x_n]$ connecting Layers 0 and 1 is the only variable weight matrix, whose elements (after the successful training of the network) are the components of one of the system roots, or in a mathematical notation $w_i = x_i$ $(i = 1, 2, \ldots, n)$.

The matrix $\boldsymbol{W}$ connecting Layers 1 and 2 is composed of n rows with the ith row to be associated with the variable x_i $(i = 1, 2, \ldots, n)$. The values of this row are the weights of the synapses joining the ith neuron of the second layer with the complete set of neurons of the third layer. There is a total number of $k = k_1 + k_2 + \cdots + k_n$ neurons in this layer and therefore the dimensions of the matrix $\boldsymbol{W}$ are $n \times k = n \times (k_1 + k_2 + \cdots + k_n)$. The values of these weights are the exponents of the x's in every term of each equation. Therefore, we have

$$w_{jl}^{i} = e_{jl}^{i} \quad \begin{cases} i = 1, 2 \ldots, n \\ j = 1, 2, \ldots k_i \\ l = 1, 2, \ldots, n. \end{cases} \tag{3.3}$$

From the programming point of view, it is not difficult to note that the matrix $\boldsymbol{W}$ is a two-dimensional matrix in which the weight $w_{\ell j}^{i}$ is associated with the cell $\boldsymbol{W}[\ell][\sigma + j]$, where $\sigma = k_1 + k_2 + \cdots + k_{i-1}$.

Finally, the matrix $\boldsymbol{A}$ connecting Layers 2 and 3 has dimensions $k \times n = (k_1 + k_2 + \cdots + k_n) \times n$ with non-zero elements in the connections between the k_i neurons of the ith block of Layer 2 with the ith neuron of Layer 3. The values of this matrix are the coefficients $\alpha's$ of the polynomial system of equations. The parameter value α_{ij} is stored in the cell $\boldsymbol{A}[i][\sigma + j]$ where the σ parameter is estimated as before. Regarding the network architecture, the values of the ith column of the $\boldsymbol{A}$ matrix are the fixed weights of the input synapses of the ith output neuron ($i = 1, 2, \ldots, n$). In mathematical language, the coefficient matrix $\boldsymbol{A}$ is defined as

$$\boldsymbol{A} = \begin{cases} \boldsymbol{A}^i_{ji} = \alpha_{ij} i = 1, 2, \ldots, n, \;\; j = 1, 2, \ldots, k_i \\ \\ \boldsymbol{A}^k_{ji} = 0 \;\; k = 1, 2, \ldots, n, \;\; k \neq i. \end{cases} \tag{3.4}$$

Since the unique neuron of the first layer does not participate in the calculations, it is not included in the index notation and therefore if we use the symbol u to describe the neuron input and the symbol v to describe the neuron output, the symbols $u1$ and $v1$ are associated with the n neurons of the second layer, the symbols $u2$ and $v2$ are associated with the k neurons of the third layer and the symbols $u3$ and $v3$ are associated with the n neurons of the fourth (output) layer. These symbols are accompanied by additional indices that identify a specific neuron inside a layer and this notation is used throughout the remaining part of the paper.

Example 3.1 *To understand the architecture of the neural network-like system and the way it is formed, consider a polynomial system of two equations with two unknowns* ($n = 2$) *in which the first equation contains two terms* ($k_1 = 2$) *and the second equation contains three terms* ($k_2 = 3$). *Using the above notation, this system is expressed as*

$$\begin{aligned} f_1(x_1, x_2) &= \alpha_{11} x_1^{e^1_{11}} x_2^{e^1_{21}} + \alpha_{12} x_1^{e^1_{12}} x_2^{e^1_{22}} - \beta_1, \\ f_2(x_1, x_2) &= \alpha_{21} x_1^{e^2_{11}} x_2^{e^2_{21}} + \alpha_{22} x_1^{e^2_{12}} x_2^{e^2_{22}} + \alpha_{23} x_1^{e^2_{13}} x_2^{e^2_{23}} - \beta_2, \end{aligned}$$

and its solution is the vector $\boldsymbol{x} = (x_1, x_2)^T$. *The final form of the system depends on the values of the above parameters. For example, using the exponents* $e^1_{11} = 1$, $e^1_{21} = e^1_{22} = e^2_{21} = e^2_{13} = 2$, $e^1_{12} = e^2_{11} = 3$, $e^2_{12} = e^2_{22} = e^2_{23} = 4$, *the coefficients* $\alpha_{11} = 1$, $\alpha_{12} = -3$, $\alpha_{21} = \alpha_{22} = 2$, $\alpha_{23} = -4$ *and the fixed terms* $\beta_1 = 5$ *and* $\beta_2 = -3$, *the system is*

$$\begin{aligned} f_1(x_1, x_2) &= x_1 x_2^2 - 3x_1^3 x_2^2 - 5, \\ f_2(x) &= 2x_1^3 x_2^2 + 2x_1^4 x_2^4 - 4x_1^2 x_2^4 + 3. \end{aligned}$$

The neural network that solves this example system is shown in Figure 3.

Example 3.2 *The neural network architecture for the system of polynomials*

$$\begin{aligned} f_1(x_1, x_2) &= 0.25x_1^2 + x_2^2 - 1 = 0, \\ f_2(x_1, x_2) &= x_1^2 - 2x_1 + x_2^2 = 0, \end{aligned}$$

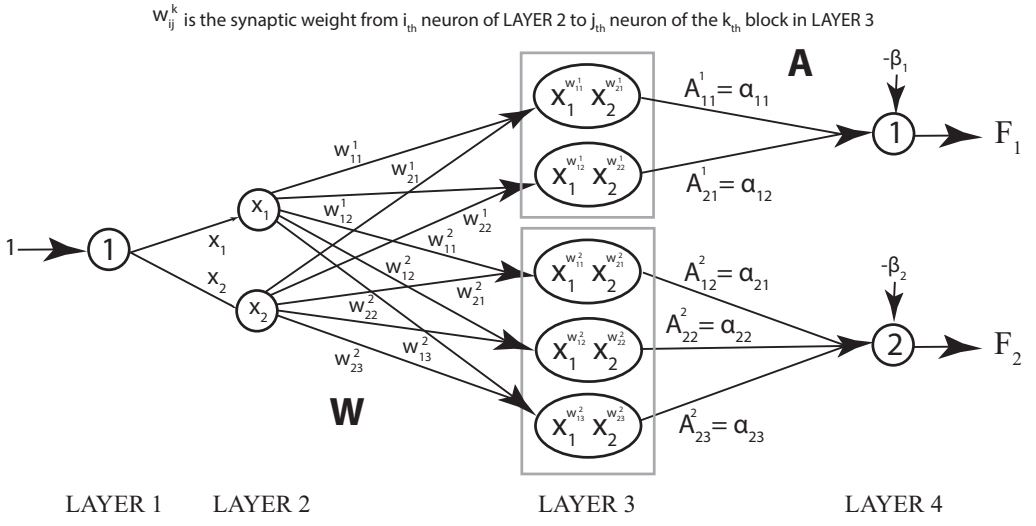

FIGURE 3. The structure of neural solver for Example 3.1.

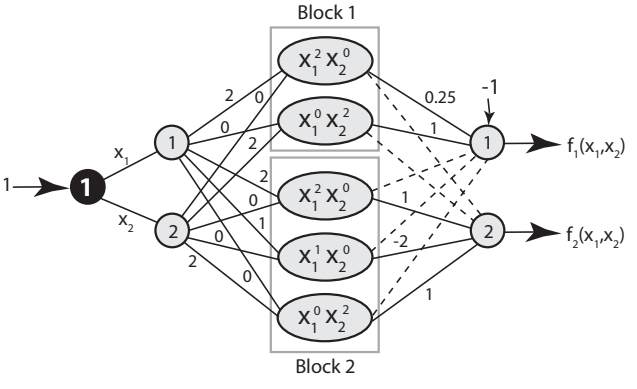

FIGURE 4. The structure of neural solver for Example 3.2. The dashed lines represent synapses with zero weights.

is shown in Figure 4, while, the contents of the matrices $\boldsymbol{W}$ *and* $\boldsymbol{A}$ *are the following:*

$$\boldsymbol{W} = \begin{pmatrix} w_{11}^1 & w_{12}^1 & w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^1 & w_{22}^1 & w_{21}^2 & w_{22}^2 & w_{23}^2 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 2 & 1 & 0 \\ 0 & 2 & 0 & 0 & 2 \end{pmatrix} \quad \textit{and}$$

$$\boldsymbol{A} = \begin{pmatrix} \alpha_{11} & \alpha_{21} \\ \alpha_{12} & \alpha_{22} \\ \alpha_{13} & \alpha_{23} \\ \alpha_{14} & \alpha_{24} \\ \alpha_{15} & \alpha_{25} \end{pmatrix} = \begin{pmatrix} 0.25 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & -2 \\ 0 & 1 \end{pmatrix}.$$

3.1 Deriving the back propagation equations

At this point, we can proceed to the development of the back propagation equations that implement the three well-known stages of this algorithm, namely the forward pass, the backward pass associated with the estimation of the delta parameter and the second forward pass related to the weight adaptation process. In a more detailed description, these stages are performed as follows:

3.1.1 *Forward pass*

The inputs and the outputs of the neurons during the forward pass stage, are computed as follows:

LAYER 1 $u1_i = w_i = x_1$, $v1_i = u1_i = x_i$ $i = 1, 2, \ldots, n$.

LAYER 2 The inputs to the neurons of Layer 2 are

$$u2_j^i = \prod_{\ell=1}^{n} v1_\ell^{w_{\ell j}^i} = \prod_{\ell=1}^{n} x_\ell^{e_{\ell j}^i}, \tag{3.5}$$

while their associated outputs have the form $v2_j^i = u2_j^i$ $(i = 1, 2, \ldots, n,\ \ j = 1, 2, \ldots, k_i)$.

LAYER 3 The inputs of the Layer 3 neurons are

$$\begin{aligned} u3_i &= \sum_{j=1}^{k_i} v2_j^i A_{ji}^i - \beta_i = \sum_{j=1}^{k_i} \alpha_{ij} \prod_{\ell=1}^{n} x_\ell^{e_{\ell j}^i} - \beta_i \\ &= F_i(\boldsymbol{x}) = F_i(x_1, x_2, \ldots, x_n) \end{aligned} \tag{3.6}$$

for the values $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, k_i$. Regarding the output $v3_i$, it depends on the activation function associated with the output neurons. In this paper, the simulator uses two different functions of interest, the identity function $y = f(x) = x$ (Case I) and the hyperbolic tangent function $y = f(x) = \tanh(x)$ (Case II). The corresponding output of the Layer 3 neurons, is therefore defined as

$$v3_i = \begin{cases} u3_i & i = 1, 2, \ldots, n \qquad \text{(Case I)} \\ \tanh(u3_i) & i = 1, 2, \ldots, n \qquad \text{(Case II).} \end{cases} \tag{3.7}$$

3.1.2 *Backward pass – estimation of δ parameters*

In this phase of back propagation algorithm, the values of δ parameters are estimated. Using the notation $\delta 1$, $\delta 2$ and $\delta 3$ to denote these parameters for Layer 1, Layer2 and Layer3 neurons, their estimation equation are the following:

- Case I (Identity function):

$$\begin{aligned}
&\delta 3_i = -F_i(\boldsymbol{x}) \qquad (i = 1, 2, \ldots, n), \\
&\delta 2_j^{i(x_k)} = \delta 3_i \boldsymbol{A}_{ji}^i \frac{\partial v 2_j^i}{\partial x_k} = -F_i(\boldsymbol{x}) \alpha_{ij} e_{kj}^i x_k^{e_{kj}^i - 1} \prod_{\substack{\ell=1 \\ \ell \neq k}}^{n} x_\ell^{e_{\ell j}^i}, \\
&\text{for the values } i, k = 1, 2, \ldots, n \text{ and } j = 1, 2, \ldots, k_i
\end{aligned} \tag{3.8}$$

$$\delta 1_k = \sum_{i=1}^{n} \sum_{j=1}^{k_i} \delta 2_j^{i(x_k)} = -\sum_{i=1}^{n} F_i(\boldsymbol{x}) \sum_{j=1}^{k_i} \alpha_{ij} e_{kj}^i x_k^{e_{kj}^i - 1} \prod_{\substack{\ell=1 \\ \ell \neq k}}^{n} x_\ell^{e_{\ell j}^i}$$

$(k = 1, 2, \ldots, n)$. The parameter $\delta 2_j^{i(x_k)}$ is associated with the unknown x_k $(k = 1, 2, \ldots, n)$.

- Case II (Hyperbolic tangent function):

$$\begin{aligned}
&\delta 3_i = -v 3_i (1 - v 3_i^2) = -\tanh[F_i(\boldsymbol{x})] \Big(1 - \tanh^2[F_i(\boldsymbol{x})] \Big) \qquad (i = 1, 2, \ldots, n), \\
&\delta 2_j^{i(x_k)} = \delta 3_i \boldsymbol{A}_{ji}^i \frac{\partial v 2_j^i}{\partial x_k} = -\tanh[F_i(\boldsymbol{x})] \Big(1 - \tanh^2[F_i(\boldsymbol{x})] \Big) \times \alpha_{ij} e_{kj}^i x_k^{e_{kj}^i - 1} \prod_{\substack{\ell=1 \\ \ell \neq k}}^{n} x_\ell^{e_{\ell j}^i},
\end{aligned}$$

for the values $i, k = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, k_i$.

$$\begin{aligned}
\delta 1_k &= \sum_{i=1}^{n} \tanh[F_i(\boldsymbol{x})] \Big(1 - \tanh^2[F_i(\boldsymbol{x})] \Big) \sum_{j=1}^{k_i} \delta 2_j^{i(x_k)} = -\sum_{i=1}^{n} \sum_{j=1}^{k_i} \alpha_{ij} e_{kj}^i x_k^{e_{kj}^i - 1} \prod_{\substack{\ell=1 \\ \ell \neq k}}^{n} x_\ell^{e_{\ell j}^i} \\
&= -\sum_{i=1}^{n} \tanh[F_i(\boldsymbol{x})] \Big(1 - \tanh^2[F_i(\boldsymbol{x})] \Big) \times \sum_{j=1}^{k_i} \alpha_{ij} e_{kj}^i x_k^{e_{kj}^i - 1} \prod_{\substack{\ell=1 \\ \ell \neq k}}^{n} x_\ell^{e_{\ell j}^i}
\end{aligned}$$

$(k = 1, 2, \ldots, n)$. The parameter $\delta 2_j^{i(x_k)}$ is associated with the unknown x_k $(k = 1, 2, \ldots, n)$.

4 Convergence analysis and update of the synaptic weights

- Case I (Identity function): We define the energy function as

$$E(\boldsymbol{x}) = \frac{1}{2} \sum_{i=1}^{n} (d_i - v 3_i)^2 = \frac{1}{2} \sum_{i=1}^{n} (0 - v 3_i)^2 = \frac{1}{2} \sum_{i=1}^{n} (v 3_i)^2 = \frac{1}{2} \sum_{i=1}^{n} [F_i(\boldsymbol{x})]^2,$$

[where $d_i = 0$ $(i = 1, 2, \ldots, n)$ is the desired output of the ith neuron of the output layer and $v3_i$ $(i = 1, 2, \ldots, n)$ is the corresponding real output] and therefore we have

$$\frac{\partial E(\boldsymbol{x})}{\partial x_k} = \sum_{i=1}^{n} F_i(\boldsymbol{x}) \frac{\partial F_i(\boldsymbol{x})}{\partial x_k} = \sum_{i=1}^{n} F_i(\boldsymbol{x}) \sum_{j=1}^{k_i} \alpha_{ij} e_{kj}^{i} x_k^{e_{kj}^{i}-1} \times \prod_{\substack{\ell=1 \\ \ell \neq i}}^{n} x_\ell^{e_{\ell j}^{i}} = -\delta 1^k$$

$(k = 1, 2, \ldots, n)$. By applying the weight update equation of the back propagation, we get the following expression:

$$x_k^{m+1} = x_k^m + \beta(m) \frac{\partial E(\boldsymbol{x})}{\partial x_k} = x_k^m - \beta(m) \delta 1^k \tag{4.1}$$

$(k = 1, 2, \ldots, n)$, where β is the learning rate of the back propagation algorithm and x_k^m and x_k^{m+1} are the values of the synaptic weight $w_k = x_k$ during the mth and $(m+1)$th iterations, respectively.

- Case II (Hyperbolic tangent function): Working in the same way, this time we have

$$E(\boldsymbol{x}) = \frac{1}{2} \sum_{i=1}^{n} (d_i - v3_i)^2 = \frac{1}{2} \sum_{i=1}^{n} (0 - v3_i)^2 = \frac{1}{2} \sum_{i=1}^{n} (v3_i)^2 = \frac{1}{2} \sum_{i=1}^{n} \tanh^2[F_i(\boldsymbol{x})].$$

In this case, the partial derivative of the mean square error with respect to x_k has the form:

$$\begin{aligned} \frac{\partial E(\boldsymbol{x})}{\partial x_k} &= \sum_{i=1}^{n} \tanh[F_i(\boldsymbol{x})] \frac{\partial \tanh[F_i(\boldsymbol{x})]}{\partial x_k} = \sum_{i=1}^{n} \tanh[F_i(\boldsymbol{x})] \Big(1 - \tanh^2[F_i(\boldsymbol{x})] \Big) \frac{\partial F_i(\boldsymbol{x})}{\partial x_k} \\ &= \sum_{i=1}^{n} \tanh[F_i(\boldsymbol{x})] \Big(1 - \tanh^2[F_i(\boldsymbol{x})] \Big) \times \sum_{j=1}^{k_i} \alpha_{ij} e_{kj}^{i} x_k^{e_{kj}^{i}-1} \prod_{\substack{\ell=1 \\ \ell \neq k}}^{n} x_\ell^{e_{\ell j}^{i}} = -\delta 1^k \end{aligned}$$

$(k = 1, 2, \ldots, n)$ and the weight adaptation equation is given again by the expression:

$$x_k^{m+1} = x_k^m + \beta(m) \frac{\partial E(\boldsymbol{x})}{\partial x_k} = x_k^m - \beta(m) \delta 1^k \tag{4.2}$$

for the values $k = 1, 2, \ldots, n$.

5 The case of adaptive learning rate

The adaptive learning rate is one of the most interesting features of the proposed simulator since it allows to each neuron of Layer 1 to be trained with its own learning rate value $\beta(k)$ $(k = 1, 2, \ldots, n)$. However, the values of these individual learning rates must allow the algorithm to converge, and in the next section the required convergence conditions are established.

The energy function associated with the mth iteration is defined as

$$E^m(\boldsymbol{x}) = \frac{1}{2} \sum_{i=1}^{n} [F_i^m(\boldsymbol{x})]^2, \tag{5.1}$$

and therefore the difference between the energies associated with the mth and $(m+1)$th iterations has the form:

$$\Delta E^m(\boldsymbol{x}) = E^{m+1}(\boldsymbol{x}) - E^m(\boldsymbol{x}) = \frac{1}{2}\sum_{i=1}^{n}[F_i^{m+1}(\boldsymbol{x})]^2 - \frac{1}{2}\sum_{i=1}^{n}[F_i^m(\boldsymbol{x})]^2$$
$$= \frac{1}{2}\sum_{i=1}^{n}\left\{\Delta F_i^m(\boldsymbol{x})\left[\Delta F_i^m(\boldsymbol{x}) + 2F_i^m(\boldsymbol{x})\right]\right\}, \tag{5.2}$$

as it can be easily be proven via simple mathematical operations. From the weight update equation of the back propagation algorithm, it can be easily seen that

$$\Delta x_k = -\beta(k)\frac{\partial E^m(\boldsymbol{x})}{\partial x_k} = -\beta(k)\sum_{\ell=1}^{n} F_\ell^m(\boldsymbol{x})\frac{\partial F_\ell^m(\boldsymbol{x})}{\partial x_k}, \tag{5.3}$$

and therefore,

$$\Delta F_i^m(\boldsymbol{x}) = \frac{\partial F_i^m(\boldsymbol{x})}{\partial x_k}\Delta x_k = -\beta(k)\frac{\partial F_i^m(\boldsymbol{x})}{\partial x_k}\sum_{\ell=1}^{n} F_\ell^m(\boldsymbol{x})\frac{\partial F_\ell^m(\boldsymbol{x})}{\partial x_k}$$

$(k = 1, 2, \ldots, n)$. Using this expression in the equation of $\Delta E^m(\boldsymbol{x})$, after the appropriate mathematical manipulations gets the form:

$$\Delta E^m(\boldsymbol{x}) = \frac{1}{2}\beta(k)\left(\sum_{\ell=1}^{n} F_\ell^m(\boldsymbol{x})\frac{\partial F_\ell^m(\boldsymbol{x})}{\partial x_k}\right)^2 \times \left[\beta(k)\sum_{i=1}^{n}\left(\frac{\partial F_i^m(\boldsymbol{x})}{\partial x_k}\right)^2 - 2\right].$$

The convergence condition of the back propagation algorithm is expressed as $\Delta E^m(\boldsymbol{x}) < 0$, an inequality that leads directly to the required criterion of convergence

$$\beta(k) < \frac{2}{\sum_{i=1}^{n}\left(\frac{\partial F_i^m(\boldsymbol{x})}{\partial x_k}\right)^2}. \tag{5.4}$$

Defining the adaptive learning rate parameter (ALRP) μ, this expression is written as

$$\beta(k) = \frac{\mu}{\|\boldsymbol{C}_k^m(\boldsymbol{J})\|^2}, \tag{5.5}$$

where $\boldsymbol{C}_k^m(\boldsymbol{J})$ is the kth column of the Jacobian matrix:

$$\boldsymbol{J} = \begin{pmatrix} \partial F_1/\partial x_1 & \partial F_1/\partial x_2 & \ldots & \partial F_1/\partial x_n \\ \partial F_2/\partial x_1 & \partial F_2/\partial x_2 & \ldots & \partial F_2/\partial x_n \\ \vdots & \ddots & \ddots & \vdots \\ \partial F_n/\partial x_1 & \partial F_n/\partial x_2 & \ldots & \partial F_n/\partial x_n \end{pmatrix}, \tag{5.6}$$

for the mth iteration. Using this notation, the back propagation algorithm converges for ALRP values $\mu < 2$.

The above description is associated with the Case I that uses the identity function, while, for Case II (hyperbolic tangent function), the appropriate equations are

$$E^m(\boldsymbol{x}) = \frac{1}{2}\sum_{i=1}^{n}(0 - v_3^{i,m})^2 = \frac{1}{2}\sum_{i=1}^{n}[v_3^{i,m}]^2 = \frac{1}{2}\sum_{i=1}^{n}\tanh^2[\boldsymbol{F}_i^m(\boldsymbol{x})],$$

$$\Delta E^m(\boldsymbol{x}) = \frac{1}{2}\beta(j)\left(\sum_{\ell=1}^{n}\tanh[\boldsymbol{F}_\ell^m(\boldsymbol{x})]\frac{\partial\tanh[\boldsymbol{F}_\ell^m(\boldsymbol{x})]}{\partial x_j}\right)^2 \times\left[\beta(j)\sum_{i=1}^{n}\left(\frac{\partial\tanh[\boldsymbol{F}_i^m(\boldsymbol{x})]}{\partial x_j}\right)^2 - 2\right], \tag{5.7}$$

and by performing a lengthy (but quite similar analysis) the convergence criterion is proven to have the form:

$$\beta(j) < \frac{2}{\sum_{i=1}^{n}\left(\frac{\partial\tanh[\boldsymbol{F}_i^m(\boldsymbol{x})]}{\partial x_j}\right)^2} = \frac{2}{\sum_{i=1}^{n}\left(\frac{\partial\tanh[\boldsymbol{F}_i^m(\boldsymbol{x})]}{\partial \boldsymbol{F}_i^m(\boldsymbol{x})}\times\frac{\partial \boldsymbol{F}_i^m(\boldsymbol{x})}{\partial x_j}\right)^2}$$

$$= \frac{2}{\sum_{i=1}^{n}\left[\left(1-\tanh^2[\boldsymbol{F}_i^m(\boldsymbol{x})]\right)\times\frac{\partial \boldsymbol{F}_i^m(\boldsymbol{x})}{\partial x_j}\right]^2}\frac{2}{\sum_{i=1}^{n}\left[\left(1-[v_3^{\ell}]^2\right)\times\frac{\partial \boldsymbol{F}_i^m(\boldsymbol{x})}{\partial x_j}\right]^2},$$

with the μ parameter to be defined accordingly.

6 Experimental results

To examine and evaluate the validity, accuracy and performance of the proposed neural solver, selected polynomial algebraic systems were solved and the simulation results were compared against those obtained by other methods. In these simulations, all the three modes of training were examined, namely as follows:

- Linear Adaptive Learning Rate (LALR) that uses as activation function the identity function $y = x$ and minimizes the sum $\sum_i \boldsymbol{F}_i(\boldsymbol{x})$.
- Non-linear Adaptive Learning Rate (NLALR) that uses as activation function the hyperbolic tangent function $y = \tanh(x)$ and minimizes the sum $\sum_i \tanh(\boldsymbol{F}_i)(\boldsymbol{x})$.
- Non-linear Modified Adaptive Learning Rate (NLMALR) that uses as activation function the hyperbolic tangent function $y = \tanh(x)$ and minimizes the sum $\sum_i \boldsymbol{F}_i(\boldsymbol{x})$.

Even though in the theoretical analysis and the construction of the associated equations the classical back propagation algorithm was used, the simulations showed that the execution time can be further decreased if in each cycle the synaptic weights were updated one after the other and the new output values were used as input parameters in the corrections associated with the next weight adaptation. To evaluate the accuracy of the results and to perform a valid comparison with the results found in the literature, different tolerance values in the form $10^{-\text{tol}}$ were used, with a value of tol = 12 to give an accuracy of six decimal digits. Therefore, the tolerance value used in each example has been selected

such that the achieved accuracy to be the same with the literature in order to make comparisons.

Since the convergence condition of the proposed neural solver has the form $\mu < 2$, where μ is the ALRP, the performed simulations used the values $\mu = 0.1 - 1.9$ with a variation step equal to 0.1 (in some cases, the value $\mu = 2.0$ was also tested). The maximum allowed number of iterations was set to $N = 1000$, while, the vector of initial conditions

$$[x_1(0), x_2(0), \ldots, x_n(0)],$$

and the n-dimensional search region $-\alpha \leqslant x_i \leqslant \alpha$ was a function of the dimension of the problem n. The main graphical representation of the results shows the variation of the minimum and the mean iteration number with respect to the value of the ALRP μ, while, the remaining results are shown in a tabulated form. In these results, the parameter SR describes the success rate, namely the percentage of the tested systems (i.e. initial condition combinations) that converged to some of the system roots, while $SR(i)$ is the success rate associated with the ith root.

After the description of the experimental conditions, let us now present eight example systems as well as the experimental results emerged for each one of them. These example systems are polynomial systems of n equations with n unknowns with increasing size $n = 2, 3, 4, 5, 6, 8, 10$. The example systems with dimensions $n = 2$ and $n = 3$ have been borrowed from [22] and [13], respectively in order to compare the proposed method with the neural methods presented there, while, the remaining examples are real engineering applications found in the literature, and more specifically a Gauss–Legendre 2-point formula for numerical integration ($n = 4$), a chemical equilibrium application ($n = 5$), a neurophysiology application ($n = 6$), a kinematic application ($n = 8$), a combustion application ($n = 10$) and an interval arithmetic application ($n = 10$). Note, that of course the proposed neural solver can handle any dimension and the maximum dimension $d = 10$ has been selected in accordance with the studied literature (all the studied simulations were applied for solving systems up to this dimension).

In the following presentation, the roots of the example systems are identified and compared with the roots estimated by the other methods.

Example 1 *Consider the following polynomial system of two equations with two unknowns* x_1, x_2 *defined as*

$$F_1(x_1, x_2) = 0.25x_1^2 + x_2^2 - 1 = 0$$
$$F_2(x_1, x_2) = (x_1 - 1)^2 + x_2^2 - 1 = 0$$

(this is the Example 2 from [22] – see also [19]). The system has two distinct roots with values

$$ROOT1\ (x_1, y_1) = \left(\frac{2}{3}, +\frac{2\sqrt{2}}{3}\right) = (0.6666666, +0.942809)$$
$$ROOT2\ (x_2, y_2) = \left(\frac{2}{3}, -\frac{2\sqrt{2}}{3}\right) = (0.6666666, -0.942809)$$

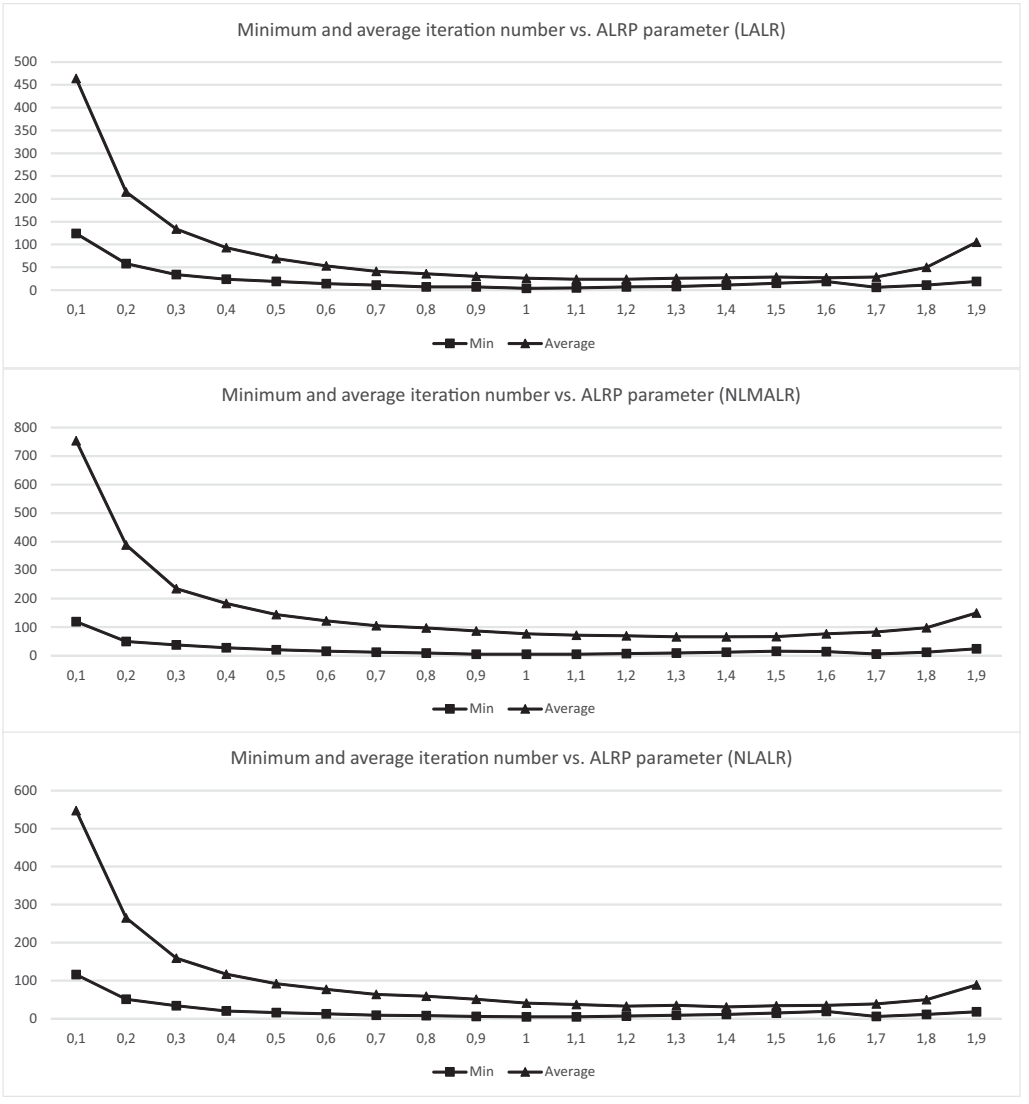

FIGURE 5. The variation of the overall minimum and the average iteration number with respect to the value of the ALRP for the Example System 1.

as well as a double root with value $ROOT3\,(x_3, y_3) = (2, 0)$. *To study the system, the network ran 1,681 times with the synaptic weights to vary in the interval* $-2 \leqslant x_1, x_2 \leqslant 2$ *with a variation step* $h = \Delta x_1 = \Delta x_2 = 0.1$ *and a tolerance value* $tol = 12$. *The variation of the overall (i.e. regardless or the root) minimum and the average iteration number for the LALR, NLMALR and NLALR modes of operation, are shown in Figure 5.*

Table 1 shows the simulation results for the Example System 1 and for the best run, namely for the minimum iteration number for Root i ($i = 1, 2, 3$) and for the modes LALR, NLMALR and NLALR. The columns of this table from left to right are the

Table 1. *The experimental results for the Example System 1*

μ	Root	MIN	AVG	SR	$x_1(0)$	$x_2(0)$	x_1	x_2	F_1	F_2
						LALR				
1.00	1	04	10	0.41	0.80	−1.00	0.666665677	−0.942808954	6.89×10^{-7}	-6.89×10^{-7}
1.80	2	11	33	0.34	−1.10	−1.90	2.000000356	−0.000149853	5.72×10^{-7}	9.63×10^{-7}
1.00	3	04	09	0.41	0.80	1.00	0.666665677	0.942808954	-7.65×10^{-7}	7.65×10^{-7}
						NLMALR				
1.00	1	05	70	0.33	0.40	−1.00	0.666666429	−0.942809021	-1.38×10^{-7}	1.38×10^{-7}
1.00	2	05	70	0.33	0.40	1.00	0.666666429	0.942809021	-2.97×10^{-7}	2.97×10^{-7}
1.70	3	06	18	0.14	1.00	−0.70	1.999999176	−0.000794747	-0.87×10^{-7}	-5.89×10^{-7}
						NLALR				
1.10	1	05	27	0.38	0.60	−0.90	0.666665202	−0.942809110	0.24×10^{-7}	5.40×10^{-7}
1.10	2	05	27	0.38	0.60	0.90	0.666665202	0.942809110	2.20×10^{-7}	-1.11×10^{-7}
1.60	3	19	29	0.13	1.80	−0.40	1.999998527	−0.001361467	4.40×10^{-7}	-1.26×10^{-8}

Table 2. *A comparison of the results emerged by applying the method described in [13] and the proposed method for the Example System 2. In the same table, the simulation results associated with the same initial values used in [13] are also shown*

				Run with initial values as in [13]				
	Method of ref. [13]	Proposed method	ALRP value		Method of ref. [13]	Proposed method	ALRP value	Method used
ROOT1	0304	25	1.2	ROOT1	0304	37	1.1	NLALR
ROOT2	0734	11	1.1	ROOT2	0734	26	1.3	NLALR
ROOT3	1992	23	1.5	ROOT3	1992	79	1.8	LALR
ROOT4	1930	15	1.4	ROOT4	1930	75	1.8	LARL

value of μ parameter, the root Id, the minimum (MIN) and the average (AVG) iteration number, the success rate (SR), the initial conditions $x_1(0)$ and $x_2(0)$, the identified root components x_1 and x_2 and the values of the functions $F_1(x_1, x_2)$ and $F_2(x_1, x_2)$ estimated for the identified root. In this simulation, the absolute error value was of the order of 10^{-6} since the used tolerance value was tol = 12. The simulator was also tested using the initial values $x_1(0) = 0.7$ and $x_2(0) = 1.3$ that according to the literature (see [22]) resulted to a minimum number of iteration with a value equal to 345. The simulation converged to exactly the same root but in only six to eight iterations a fact that is associated with the adaptive learning rate feature.

Example 2 *Consider the following polynomial system of three equations with three unknowns x_1, x_2, x_3 defined as*

$$F_1(x_1, x_2, x_3) = x_1^3 + x_2^3 + x_3^3 - 3x_1^2 - 2x_2^2 - 2x_3^2 + 2x_1 + 2x_2 + x_3 = 0,$$
$$F_2(x_1, x_2, x_3) = x_1^3 + x_2^3 + x_3^3 - x_1^2 - 5x_2^2 - x_3^2 + 8x_2 - 4 = 0,$$
$$F_3(x_1, x_2, x_3) = x_1^3 + x_2^3 + x_3^3 - 4x_1^2 - 4x_2^2 - 5x_3^2 + 4x_1 + 5x_2 + 8x_3 - 6 = 0$$

(this is the Example 2 from [13]). This system has four distinct roots with values:

$$ROOT1\ (x_1, x_2, x_3) = (1.428042, 0.384132, 1.383866),$$
$$ROOT2\ (x_1, x_2, x_3) = (1.107007, 1.008547, 0.568966),$$
$$ROOT3\ (x_1, x_2, x_3) = (-0.020060, 0.856390, 1.143860),$$
$$ROOT4\ (x_1, x_2, x_3) = (0, 1, 1).$$

The neural solver was able to identify all the four roots with an absolute error of 10^{-6} due to the used tolerance value of tol = 13. The root estimation procedure was performed in the search region $[-3, 3]$ with a variation step of $\Delta x_1 = \Delta x_2 = 0.2$ and all the three modes of operation (LALR, NLMALR, NLALR) were tested. The variation of the minimum and the average iteration number for each one of the four roots are shown in Figure 6. A comparison of the proposed method to the one presented in [13] gave to the results of Table 2 that reveal the superiority of the proposed method over the method

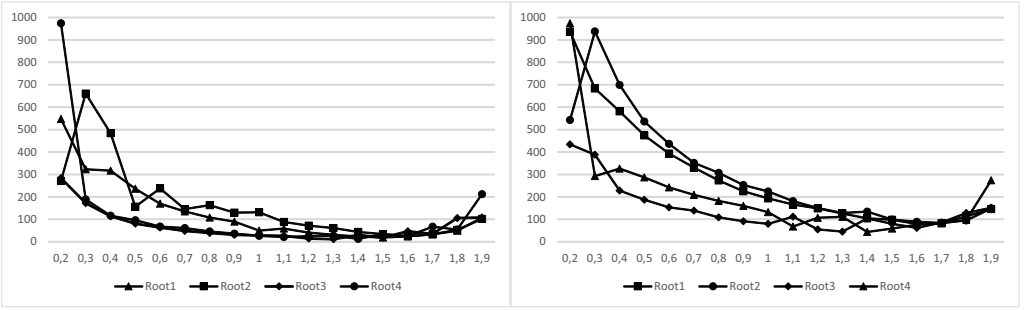

FIGURE 6. The variation of the minimum and the average iteration number with respect to the value of the ALRP for the four roots of the Example System 2 and for the NLALR mode of operation.

described in [13] where the learning rate value was not adaptive but always had the same fixed value. In this table, the first column includes the results reported in [13], while the second and the third column include the minimum iteration number and the associated ALRP for the proposed method.

Example 3 (***n* = 4** – Gauss–Legendre 2-point formula for numerical integration) *The next example is from the field of numerical integration, where the Gauss–Legendre N-point iteration formula for* $n = 2$*, results in the following non-linear algebraic system of four equations with four unknowns* (x_1, x_2, x_3, x_4).

$$F_1(x_1, x_2, x_3, x_4) = x_3 + x_4 = 0,$$
$$F_2(x_1, x_2, x_3, x_4) = x_1 x_3 + x_2 x_4 = 0,$$

$$F_3(x_1, x_2, x_3, x_4) = x_1^2 x_3 + x_2^2 x_4 - (2/3) = 0,$$
$$F_4(x_1, x_2, x_3, x_4) = x_1^3 x_3 + x_2^3 x_4 = 0$$

(see [12] and also [26]). The system has two symmetric roots in the search region $-1.5 \leqslant x_i \leqslant 1.5$ $(i = 1, 2, 3, 4)$ *whose values were identified by the neural network as*

$$\begin{aligned} ROOT1\ (x_1, x_2, x_3, x_4) = \\ (+0.5773502692, -0.5773502692, \\ +1.0000000000, +1.0000000000) \\ ROOT2\ (x_1, x_2, x_3, x_4) = \\ (-0.5773502692, +0.5773502692, \\ +1.0000000000, +1.0000000000), \end{aligned}$$

using a variation step $\Delta x_i = 0.3$ $(i = 1, 2, 3, 4)$ *and a tolerance value of* $tol = 2$*. Karr et al.* [18] *finds the one of the roots, while El-Emary and El-Kareem find the other root. The proposed neural solver was able to identify both roots after* 34 *iterations. The variation*

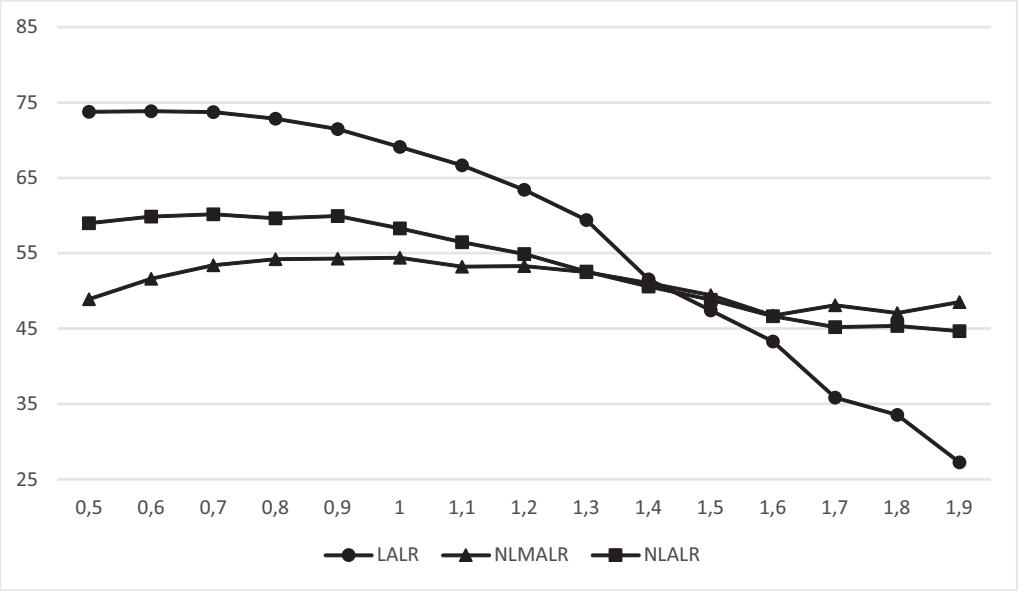

FIGURE 7. The variation of the overall success rate for the Example System 3 and for the interval $0.5 \leqslant \mu \leqslant 1.9$.

of the overall success rate with respect to the ALRP in the interval $0.5 \leqslant \mu \leqslant 1.9$ *for the operation modes LALR, NLMALR and NLALR in shown in Figure 7.*

Example 4 (***n* = 5** – Chemical equilibrium application) *The next system is associated with a chemical engineering application. It has five equations with five unknowns* $(x_1, x_2, x_3, x_4, x_5)$ *and it is defined as*

$$\begin{aligned}
F_1 &= F_1(x_1, x_2, x_3, x_4, x_5) = x_1x_2 + x_1 - 3x_5 = 0,\\
F_2 &= F_2(x_1, x_2, x_3, x_4, x_5) = 2x_1x_2 + x_1 + x_2x_3^2 + R_8x_2\\
&\quad - Rx_5 + 2R_{10}x_2^2 + R_7x_2x_3 + R_9x_2x_4 = 0,\\
F_3 &= F_3(x_1, x_2, x_3, x_4, x_5) = 2x_2x_3^2 + 2R_5x_3^2 - 8x_5\\
&\quad + R_6x_3 + R_7x_2x_3 = 0,
\end{aligned}$$

$$\begin{aligned}
F_4 &= F_4(x_1, x_2, x_3, x_4, x_5) = R_9x_2x_4 + 2x_4^2 - 4Rx_5 = 0,\\
F_5 &= F_5(x_1, x_2, x_3, x_4, x_5) = x_1(x_2 + 1) + R_{10}x_2^2\\
&\quad + x_2x_3^2 + R_8x_2 + R_5x_3^2 + x_4^2 - 1 + R_6x_3\\
&\quad + R_7x_2x_3 + R_9x_2x_4 = 0,
\end{aligned}$$

where the constants that appear in the above equations are defined as $R = 10$, $R_5 = 0.193$,

Table 3. *Simulation results for the Example System 4 for values* $x_i(0) = \mathbf{10}$ $(i = 1, 2, 3, 4, 5)$, $h = 20$ *and* $tol = 3$

ALRP value	Roots found	Roots in domain	Overall average	Overall min	Success rate	Mean global absolute error
1.7	05	05	42.40	13.00	0.16	4.667735931086521e−003
1.6	04	04	10.00	10.00	0.13	4.957275434429678e−003
1.5	11	11	15.09	06.00	0.34	4.674821343136327e−003
1.4	18	18	10.17	08.00	0.56	4.825119557276585e−003
1.3	23	23	09.83	07.00	0.72	3.926034096798575e−003
1.2	23	23	15.57	12.00	0.72	4.544813622758976e−003
1.1	20	20	15.85	10.00	0.63	3.661480777692344e−003
1.0	27	27	13.44	06.00	0.84	4.515696344648852e−003
0.9	22	22	15.91	06.00	0.69	4.561322590387505e−003
0.8	26	26	24.38	07.00	0.81	4.701823281735112e−003
0.7	15	15	38.73	19.00	0.47	4.714094850138384e−003
0.6	18	18	36.72	15.00	0.56	5.046436895669526e−003
0.5	27	27	47.96	10.00	0.84	5.433914833493907e−003

and

$$R_6 = \frac{0.002597}{\sqrt{40}},\quad R_7 = \frac{0.003448}{\sqrt{40}},\quad R_8 = \frac{0.00001799}{40}$$
$$R_9 = \frac{0.0002155}{\sqrt{40}},\quad R_{10} = \frac{0.00003846}{40}$$

(this is the Example 5 from [29], see also [14]). This system has a lot of roots. The method of Oliveira and Petraglia [29] was able to identify seven roots, while the method of Grosan et al. [14] was able to identify eight roots; however, no root values are reported. To deal with this system, the neural network run with initial conditions

$$[x_1(0), x_2(0), x_3(0), x_4(0), x_5(0)] = (\pm 10, \pm 10, \pm 10, \pm 10, \pm 10),$$

with a variation step $h = \Delta x_i = 20$ *(this mean that each* x_i $(i = 1, 2, 3, 4, 5)$ *got the values* ± 10 *only, leading thus to a number of* $2^5 = 32$ *different examined combinations). The algorithm worked in LALR, NLMALR and NLALR modes of operation with a lot of tolerance values. The minimum number of iterations was* 6 *for* $tol = 3$, $7 - 9$ *for* $tol = 4$ *and* 32 *for* $tol = 5$ *and for the value* $tol = 5$ *regarding the global absolute error, the accuracy of the results was superior with respect to [29] and [14]. Restricting to this case, as an example, the results for the NLALR method are shown in Table 3. The values of the 18 roots associated with the value* $\mu = 1.4$ *(see the fourth row of Table 3) are shown in Table 4 while the variation of the overall success rate for the three modes of operations and for the values* $0.5 \leqslant \mu \leqslant 1.5$ *are shown in Figure 8. It is interesting to note that the roots of this system are located to the edges of hypercubes (i.e. they are combinations of the same values) as it is shown from Table 4 for the roots 4–5, 6–7, 8–9, 10–11, 15–16 and 17–18.*

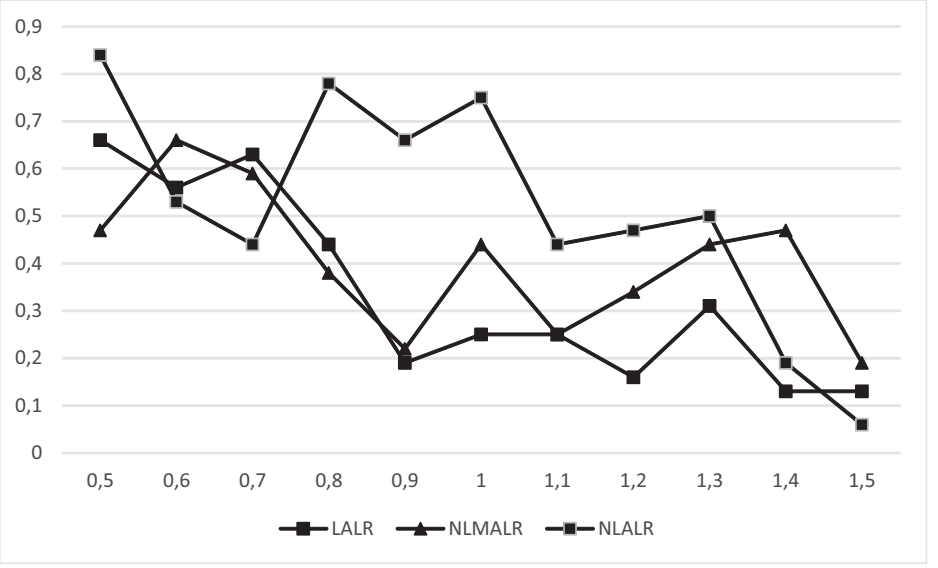

FIGURE 8. The variation of the overall success rate for the Example System 4 and for the interval $0.5 \leqslant \mu \leqslant 1.5$.

Example 5 (***n* = 6** – Neurophysiology application) *The next system is associated with a neurophysiology application and has six equations with six unknowns* $(x_1, x_2, x_3, x_4, x_5, x_6)$. *This system is defined as*

$$\begin{aligned}
F_1(x_1, x_2, x_3, x_4, x_5, x_6) &= x_1^2 + x_3^2 - 1 = 0,\\
F_2(x_1, x_2, x_3, x_4, x_5, x_6) &= x_2^2 + x_4^2 - 1 = 0,\\
F_3(x_1, x_2, x_3, x_4, x_5, x_6) &= x_3^3 x_5 + x_4^3 x_6 = 0,\\
F_4(x_1, x_2, x_3, x_4, x_5, x_6) &= x_1^3 x_5 + x_2^3 x_6 = 0,\\
F_5(x_1, x_2, x_3, x_4, x_5, x_6) &= x_1 x_3^2 x_5 + x_2 x_4^2 x_6 = 0,\\
F_6(x_1, x_2, x_3, x_4, x_5, x_6) &= x_1^2 x_3 x_5 + x_2^2 x_4 x_6 = 0
\end{aligned}$$

(this is the Example 4 from [29], see also [14]). It should be noted that for the sake of simplicity each equation of the system was divided by the value $d = 20,000$, *namely the maximum value associated with each equation for the used initial condition vector*

$$[x_1(0), x_2(0), x_3(0), x_4(0), x_5(0), x_6(0)] = (10, 10, 10, 10, 10, 10).$$

The neural simulator was tested using the tolerance values $tol = 14, 18, 32, 34$ *for ALRP values* $0.5 \leqslant \mu \leqslant 1.9$ *and the simulation results are shown in Table 5. It seems that the simulation results are better compared with the results of [14] and [29]. From this table, it seems that even though all algorithms returned the same root number with the same overall success rates, the LALR is more efficient since it identified 40 roots compared with the 24 roots returned by NLMALR and NLALR approaches. In Table 5, RN is the number of*

Table 4. *The 18 roots associated with the Example System 4 for the value* $\mu = 1.4$ *and for tolerance* $tol = 3$

x_1	x_2	x_3	x_4	x_5
−8.131333851870817e−004	−1.359533054784222e−003	−3.043719899204538e−005	4.182264603098331e−004	−7.029763068624972e−004
−7.746229657550767e−001	−1.519421526245464e−001	1.861206258742570e+000	9.177552312059069e−001	3.221672948059449e−002
−5.793502427881108e−001	−1.322929368195001e−001	4.218103393494799e−002	1.052065811201743e+000	4.340930082972737e−002
−4.955870029772600e+000	−6.784417604683974e−001	3.603146139289912e−001	−1.882818726796256e+000	1.457606883269279e−001
−4.954087905517680e+000	−6.773965057935943e−001	3.692163362821110e−001	1.883937591760401e+000	1.458004260363130e−001
−4.933324914871970e+000	−6.699956824121340e−001	−4.463898191353368e−001	−1.891693688010017e+000	1.459005546635799e−001
−4.928264906444314e+000	−6.688533415397449e−001	−4.607875174935332e−001	1.892870052852373e+000	1.458758936635606e−001
−2.749629205285697e−002	−1.023261282935850e−001	−3.198255492481633e+000	1.340087431906510e+000	6.440088142444400e−002
−2.740159347608984e−002	−1.023306605405972e−001	−3.196740946194353e+000	−1.340156000532087e+000	6.440850897664324e−002
−2.458212775145130e−002	−1.032750164465344e−001	3.151732708157127e+000	1.342498465185122e+000	6.453547830892163e−002
−2.448700949731841e−002	−1.032824590508319e−001	3.150217790553944e+000	−1.342567308161016e+000	6.454285159083822e−002
2.611354755474082e+000	−1.785135899776621e−001	−2.184966293949471e+000	−1.239191396818034e+000	5.904950768151784e−002
3.813176992591856e−001	−1.498564688007109e−001	−2.063322258845801e−001	1.427034028611167e+000	7.486215398549007e−002
7.512486994650907e−001	−1.740472622603657e−001	−3.796605107689919e−001	−1.620346847485962e+000	9.535327523184987e−002
9.315460552903154e−002	−1.655643999796043e−001	−1.660532451153788e+000	1.426360715943424e+000	7.137201951550734e−002
9.317736452012326e−002	−1.655621961588012e−001	−1.660123725453574e+000	−1.426349728978280e+000	7.137298019670738e−002
9.650676476805797e−002	−1.658822323018465e−001	1.642568797109839e+000	1.426033418063847e+000	7.139772385244506e−002
9.652972769904977e−002	−1.658802004587660e−001	1.642162371946303e+000	−1.426022339901355e+000	7.139864953861951e−002

Table 5. *Simulation results for the Example System 5*

LALR						NLMALR						NLALR					
tol	μ	MIN	RN	SR	MGAE	tol	μ	MIN	RN	SR	MGAE	tol	μ	MIN	RN	SR	MGAE
14	1.3	008	40	75	10^{-07}	14	0.5	071	24	75	10^{-07}	14	0.7	137	24	75	10^{-07}
18	1.3	**014**	40	75	10^{-09}	18	0.5	**101**	24	75	10^{-09}	18	0.7	**188**	24	75	10^{-09}
32	1.3	030	40	75	10^{-16}	32	0.5	208	24	75	10^{-16}	32	0.7	365	24	75	10^{-16}
34	1.3	032	40	75	10^{-17}	34	0.5	223	24	75	10^{-17}	34	0.7	390	24	75	10^{-17}

identified roots for each case, while the initials MGAE stand for Mean Global Absolute Error.

The identified roots of the system for the three best cases associated with the tolerance value tol = 18 – these values are typed in bold – are the following:

$$-1.000003880464001e+000,\quad +9.999875438003126e-001,\quad -1.176766115920378e-013$$
$$+1.176761543374461e-013,\quad +1.539743797109988e+000,\quad +1.539819981135548e+000$$

$$+1.000006354559783e+000,\quad +9.999882333616720e-001,\quad +2.036905674026522e-008$$
$$+2.036958697250499e-008,\quad -8.496033838099611e-001,\quad +8.496522199402240e-001$$

$$-8.546692757403601e-001,\quad -8.546795770156993e-001,\quad +5.191637622544543e-001$$
$$+5.191619376677052e-001,\quad +3.140147740354785e-001,\quad -3.140034532894029e-001.$$

Example 6 (***n* = 8** – Kinematic application) *The next system is part of the description for a kinematic application and is composed of eight non-linear equations with eight unknowns* $\boldsymbol{x} = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ *defined as (see [29])*

$$x_j^2 + x_{j+1}^2 = 0,$$

$$\begin{aligned}\alpha_{1j}x_1x_3 + \alpha_{2j}x_1x_4 + \alpha_{3j}x_2x_3 + \alpha_{4j}x_2x_4 + \alpha_{5j}x_2x_7 +\\ \alpha_{6j}x_5x_8 + \alpha_{7j}x_6x_7 + \alpha_{8j}x_6x_8 + \alpha_{9j}x_1 + \alpha_{10j}x_2 +\\ \alpha_{11j}x_3 + \alpha_{12j}x_4 + \alpha_{13j}x_5 + \alpha_{14j}x_6 + \alpha_{15j}x_7 +\\ \alpha_{16j}x_8 + \alpha_{17j} = 0\end{aligned}$$

$(1 \leqslant j \leqslant 4)$ *with the coefficients* α_{ij} *to have the values of Table 6. Note that for sake of simplicity and better performance, each parameter value has been divided by* $d = 100$. *The neural solver was tested in the search region* $-10 \leqslant x_i \leqslant 10$ $(i = 1, 2, \ldots, 8)$ *with a variation step* $h = 20$ *and a global absolute error tolerance values* $GAE\_TOL = 8, 10, 11, 12, 15, 16$. *The best results with respect to the total number of the identified roots, are shown in Table 7. It is interesting to note that in all cases this maximum number of identified roots is associated with an ALRP value of* $\mu = 1.5$. *The variation of the average and the minimum iteration number for all roots for a tolerance value* $GAE\_TOL = 16$ *is shown in Figure 9.*

A comparison of the results emerged from the proposed neural method and the method of Oliveira and Petraglia [29] is shown in Table 8. In the method of Oliveira and Petraglia, the roots are identified with a global absolute error tolerance with values 8, 10, 11, 12, 15, 16. On the other hand, the proposed method is capable of identifying the same roots with any tolerance value up to 16. Note that the neural based solver identified six roots for this system, while the method of Oliveira and Petraglia only five roots (the Roots 2 and 3 of Table 7 in [29] are actually the same root with a different accuracy).

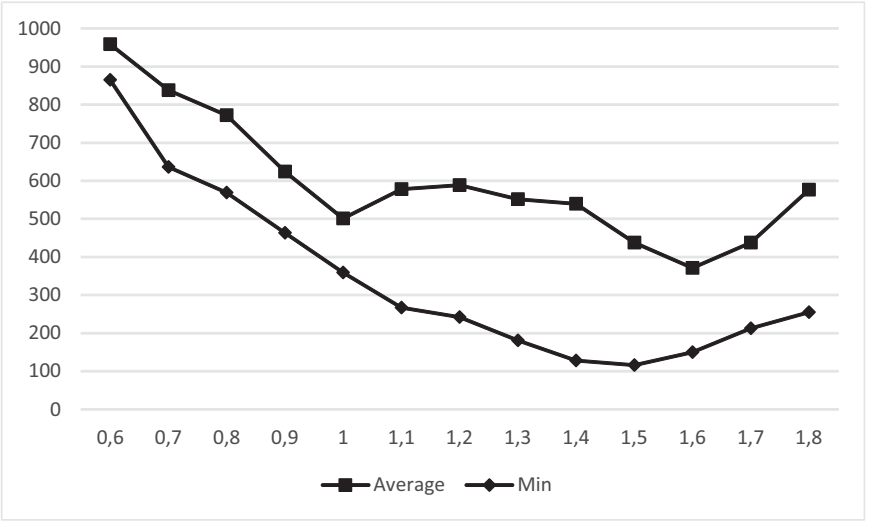

FIGURE 9. The variation of the overall minimum and average number of iterations for a tolerance GAE_TOL = 16 and for $0.6 \leqslant \mu \leqslant 1.8$ for the Example System 6.

Example 7 ($n = 10$ – Combustion application) *Let us solve now a large system of 10 equations with 10 unknowns in the form:*

$$
\begin{aligned}
F_1(\boldsymbol{x}) &= x_2 + 2x_6 + x_9 + 2x_{10} - 10^{-5} = 0,\\
F_2(\boldsymbol{x}) &= x_3 + x_8 - 3 \times 10^{-5} = 0,\\
F_3(\boldsymbol{x}) &= x_1 + x_3 + 2x_5 + 2x_8 + x_9 + x_{10} - 5 \times 10^{-5} = 0,\\
F_4(\boldsymbol{x}) &= x_4 + 2x_7 - 10^{-5} = 0,\\
F_5(\boldsymbol{x}) &= 0.5140437 \times 10^{-7} x_5 - x_1^2 = 0,\\
F_6(\boldsymbol{x}) &= 0.1006932 \times 10^{-6} x_6 - 2x_2^2 = 0,\\
F_7(\boldsymbol{x}) &= 0.7816278 \times 10^{-15} x_7 - x_4^2 = 0,
\end{aligned}
$$

$$
\begin{aligned}
F_8(\boldsymbol{x}) &= 0.1496236 \times 10^{-8} x_8 - x_1 x_3 = 0,\\
F_9(\boldsymbol{x}) &= 0.6194411 \times 10^{-7} x_9 - x_1 x_2 = 0,\\
F_{10}(\boldsymbol{x}) &= 0.2089296 \times 10^{-14} x_{10} - x_1 x_2^2 = 0,
\end{aligned}
$$

where $\boldsymbol{x} = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10})^T$ *(this is the Example 7 from [29], see also [14]). To reproduce the experimental results of the literature, the neural solver ran in the interval* $[-20, 20]$ *and since the function* $y(x) = \tanh(x)$ *does not work correctly for large values of its argument, only the identity function* $y = x$ *was used. Furthermore, to simplify the calculations, each equation was divided with the value* $d = 8,000$*, the largest value that each equation can take, for the initial vector* $x_i(0) = 20$ $(i = 1, 2, \ldots, 10)$*. This system has a lot of roots and the cited works identify different roots with different accuracy. For example, in [14], the roots are identified with an accuracy of two decimal digits while in [29] the accuracy of estimating the absolute error was five decimal digits. The neural solver is able to*

Table 6. *The coefficients* α_{ij} $(1 \leqslant i \leqslant 17,\ 1 \leqslant j \leqslant 4)$ *for the Example System 6*

	Column 1	Column 2	Column 3	Column 4
Row 01	−0.249150680	0.125016350	−0.625550077	1.489477300
Row 02	1.609135400	−0.686607360	−0.115719920	0.230623410
Row 03	0.279423430	−0.119228120	−0.666404480	1.328107300
Row 04	1.434801600	−0.719940470	0.110362110	−0.258645030
Row 05	0.000000000	−0.432419270	0.290702030	1.165172000
Row 06	0.400263840	0.000000000	1.258776700	−0.269084940
Row 07	−0.800527680	0.000000000	−0.629388360	0.538169870
Row 08	0.000000000	−0.864838550	0.581404060	0.582585980
Row 09	0.074052388	−0.037157270	0.195946620	−0.208169850
Row 10	−0.083050031	0.035436896	−1.228034200	−0.699103170
Row 11	−0.386159610	0.085383482	0.000000000	−0.699103170
Row 12	−0.755266030	0.000000000	−0.079034221	0.357444130
Row 13	0.504201680	−0.039251967	0.026387877	1.249911700
Row 14	−1.091628700	0.000000000	−0.057131430	1.467736000
Row 15	0.000000000	−0.432419270	−1.162808100	1.165172000
Row 16	0.049207290	0.000000000	1.258776700	1.076339700
Row 17	0.049207290	0.013873010	2.162575000	−0.696868090

Table 7. *The best results regarding the maximum number of identified roots for the Example System 6*

GAE_TOL	ALRP	Roots	MIN	AVERAGE	SR	Average global absolute error
08	1.5	23	298.57	059.00	0.64	9.518700509207428e−009
10	1.5	21	313.19	074.00	0.55	9.477243149484472e−011
11	1.5	21	348.76	080.00	0.55	9.555354820055328e−012
12	1.5	20	353.05	088.00	0.50	9.307679340805897e−013
15	1.5	19	409.11	110.00	0.48	9.356039335086196e−016
16	1.5	19	438.89	116.00	0.48	8.974454951112643e−017

reach better accuracy for a tolerance value of tol = 12 *that gives an accuracy of six decimal digits. The variation of the identified root number in the search interval and the minimum iteration number for ALRP values* $0.1 \leqslant \mu \leqslant 1.8$ *and for the identify activation function are shown in Figure 10. On the other hand, Figure 11 shows the relationship between the total roots identified by the network and the number of the those roots that belong in the search interval (this means that the remaining roots are located outside this 10-dimensional interval).*

Example 8 ($\boldsymbol{n = 10}$ – Interval arithmetic application) *The last system examined here, is another system of 10 equations with 10 unknowns, defined as*

$$F_1(\boldsymbol{x}) = x_1 - 0.18324757x_3x_4x_9 - 0.25428722 = 0,$$
$$F_2(\boldsymbol{x}) = x_2 - 0.16275449x_1x_6x_{10} - 0.37842197 = 0,$$

Table 8. *A comparison between the results emerged from the proposed method and the method of Oliveira and Petraglia [29], regarding the values of the identified roots associated with the Example System 6. The notation RiN* $(i = 1, 2, 4, 5, 6)$ *describes the roots identified by the neural method while RiP is the root identified in [29]*

R1N	−6.755626600166026e−001	7.373025785871713e−001	6.755626600166033e−001	−7.373025785871703e−001
	−6.755626600166036e−001	2.678657261798752e−001	4.040958160867550e−001	−9.554465010536861e−001
R1P	−6.755626600166059e−001	7.383025785871684e−001	6.755626600166059e−001	−7.373025785871685e−001
	−6.755626600166058e−001	2.678657261798790e−001	4.040958160867596e−001	−9.5544650105368735e−001
R2N	−2.048194778674478e−001	9.787997657775097e−001	2.048194778674360e−001	9.787997657775119e−001
	−2.048194778674299e−001	2.254970000035143e−001	−3.428905960820967e−001	−1.057015814569538e+000
R2P	−2.048194778719412e−001	9.787997657768206e−001	2.048194778707990e−001	9.787997657780387e−001
	−2.048194778564528e−001	2.254969999958249e−001	−3.428905960883317e−001	−1.057015814565187e+000
R4N	6.525942183197070e−001	−7.577075862202327e−001	−6.525942183197049e−001	7.577075862202350e−001
	−6.525942183197024e−001	−2.156347473355500e−001	2.941429960251973e+000	2.219355129196244e+000
R4P	6.525942183797983e−001	−7.577075861684874e−001	−6.525942183798085e−001	7.577075861612514e−001
	−6.525942185121569e−001	−2.156347473741136e−001	2.941429959906766e+000	2.219355129326753e+000
R5N	5.900443048695306e−001	8.073708678736404e−001	−5.900443048695271e−001	8.073708678736425e−001
	−5.900443048695252e−001	9.429268443544715e−001	2.668849079627183e−001	−1.224247978413023e+000
R5P	5.900443048695251e−001	8.073708678736427e−001	−5.900443048695251e−001	8.073708678736427e−001
	−5.900443048695251e−001	9.429268443544667e−001	2.668849079627161e−001	−1.224247978413024e+000
R6N	−3.728754498882111e−001	−9.278814034512500e−001	−3.728754498882155e−001	−9.278814034512503e−001
	3.728754498882122e−001	1.927664221632472e+000	4.979007501724295e−001	−5.688985915508570e−001
R6P	−3.728754498605437e−001	−9.278814034467753e−001	−3.728754499015765e−001	9.278814034448708e−001
	3.728754499045672e−001	1.927664221593003e+000	4.979007501790993e−001	−5.688985915498652e−001

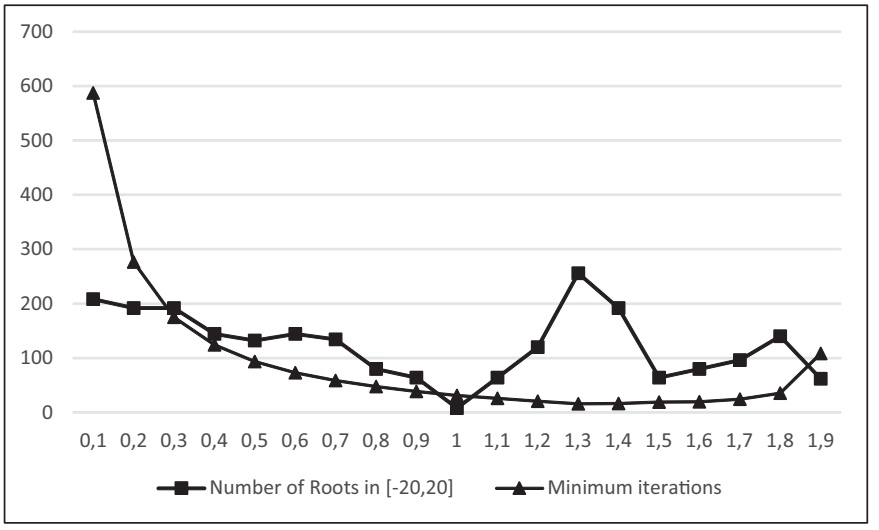

FIGURE 10. The variation of the identified root number in the search interval and the minimum iteration number for ALRP values $0.1 \leqslant \mu \leqslant 1.8$ for the identity activation function, for the Example System 7.

$$
\begin{aligned}
F_3(\boldsymbol{x}) &= x_3 - 0.16955071x_1x_2x_{10} - 0.27162577 = 0,\\
F_4(\boldsymbol{x}) &= x_4 - 0.15585316x_1x_6x_7 - 0.19807914 = 0,\\
F_5(\boldsymbol{x}) &= x_5 - 0.19950920x_3x_6x_7 - 0.44166728 = 0,\\
F_6(\boldsymbol{x}) &= x_6 - 0.18922793x_5x_8x_{10} - 0.14654113 = 0,\\
F_7(\boldsymbol{x}) &= x_7 - 0.21180486x_2x_5x_8 - 0.42937161 = 0,\\
F_8(\boldsymbol{x}) &= x_8 - 0.17081208x_1x_6x_7 - 0.07056438 = 0,\\
F_9(\boldsymbol{x}) &= x_9 - 0.19612740x_6x_8x_{10} - 0.34504906 = 0,\\
F_{10}(\boldsymbol{x}) &= x_{10} - 0.21466544x_1x_4x_8 - 0.42651102 = 0,
\end{aligned}
$$

where $\boldsymbol{x} = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10})^T$ *(this is the Example 3 from [29], see also [14]), it is associated with an interval arithmetic application, and according to [29] it has only one solution). To achieve a solution similar to the one reported in [29], a tolerance value tol* $= 29$ *has to be used with an accuracy regarding the absolute error of 14–15 decimal digits. The neural solved was ran 1,024 times working in all three modes of operation in the search interval* $-2 \leqslant x_i \leqslant 2$ $(i = 1, 2, \ldots, 10)$ *with a variation step* $h = 4$ *and for ALRP values* $0.1 \leqslant \mu \leqslant 1.9$. *The variation of the average and the minimum iteration number with respect to the value of the ALRP for tolerance tol* $= 29$ *and for the NLMALR method is shown in Figure 12. Regarding the global absolute error the cited method gave an error of* 0.00000000000000 *while the error value associated with the proposed method is* $1.526556658859590200e - 016 = 0.00000000000000$.

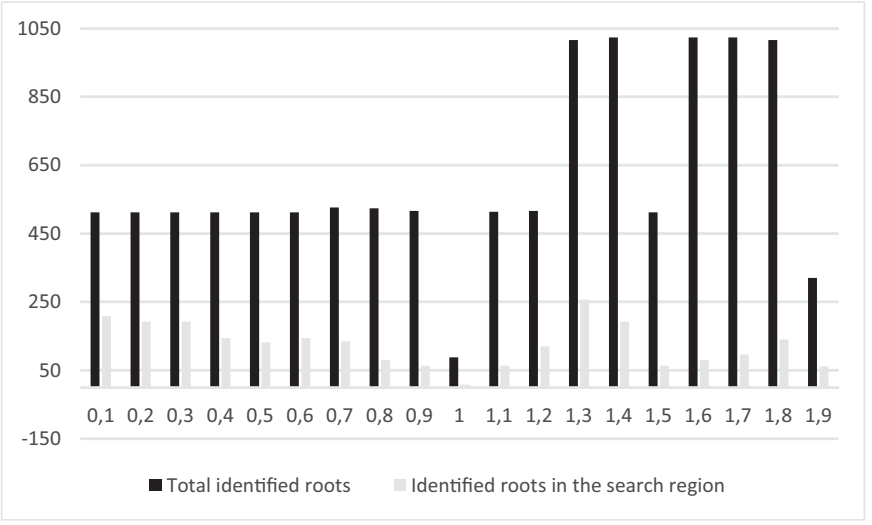

FIGURE 11. The total number of identified roots with respect to the identified roots that belong to the search interval for the Example System 7.

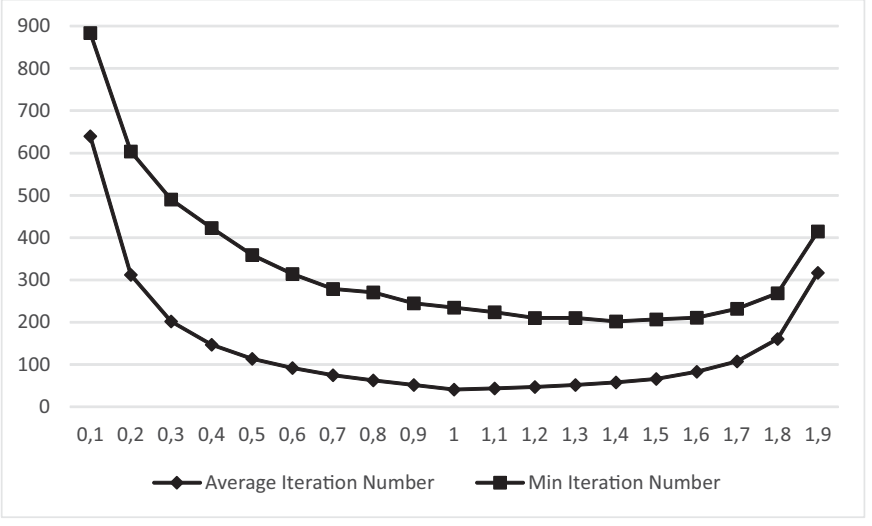

FIGURE 12. The variation of the average and the minimum iteration number with respect to the value of the ALRP (tol = 29) for the NLMALR method and for the Example System 8.

6.1 Test results with iteration number and execution CPU time for large sparse non-linear systems of equations

A very crucial task in evaluating this type of arithmetic methods, is to examine their scaling capabilities and more specifically to measure the iteration number and the execution time as the dimension of the system increases. In order to compare the results of the proposed method with the ones emerged by other well-known methods, the following example systems were implemented and solved using the proposed algorithm:

- *Example 9* This system is Problem 3 in [20] and it is defined as

$$f_i(x) = x_i^2 + x_i - 2 \quad (i = 1, 2, \ldots, n)$$

with initial conditions $x_0 = (0.5, 0.5, 0.5, \ldots, 0.5)$.

- *Example 10* This system is Problem 5 in [20] and it is defined as

$$f_i(x) = x_i^2 - 4 \quad (i = 1, 2, \ldots, n)$$

with initial conditions $x_0 = (0.5, 0.5, 0.5, \ldots, 0.5)$.

- *Example 11* This system is Problem 4 in [33] and it is defined as

$$\begin{cases} f_i(x) = 1 - x_i & i = 1, 3, \ldots, n-1 \\ f_i(x) = 10(x_i - x_{i-1})^2 & i = 2, 4, \ldots, n \end{cases}$$

with initial conditions $x_0 = (-1.2, -1.2, \ldots, -1)$.

In this simulations, the proposed method identified as GBALR (Generalized Back-propagation with Adaptive Learning Rate) is used with the identity function $y = x$ as the output neurons activation function and it is based to the minimization of the quantity $\sum_i f_i^2(x)/2$ $(i = 1, 2, \ldots, n)$. Besides this base method, there are two additional variations of this approach that are also tested, namely, the variation GBALR1 that uses the function $y = \tanh x$ and minimizes the quantity $\sum_i f_i^2(x)/2$ $(i = 1, 2, \ldots, n)$ and the variation GBALR2 that uses the function $y = \tanh x$ and minimizes the quantity $\sum_i \tanh(f_i^2(x))/2$ $(i = 1, 2, \ldots, n)$. These methods are compared against the classical and the fixed Newton's methods as well the Broyden1 and Broyden2 methods described in the theoretical section.

The simulation results for the above examples are summarized in Table 9. For each example, the system dimension n was set to the values 10, 20, 50, 100, 200, 500 and 1,000. In this table, a cell with the '–' symbol means that either the algorithm could not lead to a result (i.e. it was divergent), or the maximum number of iterations (with a value equal to 500) was reached. Regarding the simulation parameters, their values are $ALRP = 0.8$ and tol = 10 for Example 9, ALRP = 0.8 and tol = 12 for Example 10, ALRP = 1.0 and tol = 12 for Example 11, ALRP = 1.0 and tol = 14 for Example 12 and ALRP = 0.8 and tol = 10 for Example 13. The main conclusions of the above experiments are the following:

- It seems that the proposed method is better than Newton's method, since the last method does not converge in the Examples (9,12,13) (in the Examples 9 and 13, it converges only for $n = 2$). Furthermore, in the Example 10, the proposed method converged after the same number of iterations but with a better CPU time.
- Fixed Newton method converges only in the Examples 10 and 13 and requires more iterations with respect to the proposed method even though the CPU time for $n = 500, 1,000$ is better than GBALR.

Table 9. *Tables for Examples 9–11 with iteration number and CPU time (in seconds) for systems with* $n = 10, 20, 50, 100, 200, 500, 1,000$, *using Newton, Fixed Newton, Broyden-1, Broyden-2 and GBALR (three variations) methods*

		Newton		Fixed Newton		Broyden-1		Broyden-2		GBALR			GBALR1			GBALR2		
	n	ITER	CPU	ITER	CPU	ITER	CPU	ITER	CPU	ITER	CPU	ALRP	ITER	CPU	ALRP	ITER	CPU	ALRP
	10	4	0.0036	21	0.0020	6	0.0012	6	0.0009	1	0.0004	0.8	3	0.0007	1.0	3	0.0004	1.0
	20	4	0.0043	22	0.0033	6	0.0009	6	0.0006	1	0.0003	0.8	4	0.0007	1.0	3	0.0009	1.0
E	50	4	0.0049	22	0.0053	6	0.0021	6	0.0009	1	0.0006	0.8	4	0.0018	1.0	4	0.0020	1.0
X	100	4	0.0082	23	0.0143	6	0.0065	6	0.0023	1	0.0019	0.8	4	0.0066	1.0	4	0.0079	1.0
	200	4	0.0886	23	0.0822	6	0.0326	6	0.0090	1	0.0089	0.8	4	0.0337	1.0	4	0.0341	1.0
9	500	4	0.8281	24	1.0735	6	0.3282	6	0.0383	1	0.1792	0.8	4	0.7020	1.0	4	0.7333	1.0
	1,000	4	7.5342	25	7.4984	6	2.5772	6	0.1681	1	1.6868	0.8	4	6.7449	1.0	4	6.8611	1.0
	10	6	0.0027	–	–	8	0.0010	8	0.0008	6	0.0008	1.0	5	0.0006	1.0	5	0.0007	1.0
	20	6	0.0047	–	–	8	0.0014	8	0.0009	6	0.0013	1.0	5	0.0009	1.0	5	0.0007	1.0
E	50	6	0.0090	–	–	8	0.0027	8	0.0013	6	0.0027	1.0	5	0.0022	1.0	5	0.0024	1.0
X	100	6	0.0253	–	–	8	0.0084	8	0.0174	6	0.0088	1.0	5	0.0072	1.0	5	0.0087	1.0
	200	6	0.0624	–	–	8	0.0397	8	0.0088	6	0.0512	1.0	5	0.0382	1.0	5	0.0417	1.0
10	500	6	1.3586	–	–	8	0.3747	8	0.0492	6	1.0863	1.0	5	0.8978	1.0	5	0.9287	1.0
	1,000	6	11.4271	–	–	8	2.8243	8	0.2233	6	10.2622	1.0	5	8.6765	1.0	5	9.0248	1.0
	10	–	–	–	–	–	–	–	–	21	0.0018	1.0	34	0.0034	1.4	–	–	–
	20	–	–	–	–	–	–	–	–	22	0.0022	1.0	48	0.0043	1.4	–	–	–
E	50	–	–	–	–	–	–	–	–	22	0.0073	1.0	49	0.0158	1.4	–	–	–
X	100	–	–	–	–	–	–	–	–	22	0.0287	1.0	49	0.0704	1.4	–	–	–
	200	–	–	–	–	–	–	–	–	22	0.1198	1.0	50	0.2656	1.4	–	–	–
11	500	–	–	–	–	–	–	–	–	23	2.0861	1.0	50	4.5391	1.4	–	–	–
	1,000	–	–	–	–	–	–	–	–	23	19.0208	1.0	51	44.7966	1.4	–	–	–

- Broyden methods converge only in the Example 9 and requires more iterations than GBALR even though it achieves better CPU times for $n = 500, 1,000$.
- In Example 9, GBALR requires only one iteration and therefore it is better than all the other methods regarding the iteration number; however, Broyden methods are superior than GBALR for large values of n.
- None of the classical methods converges in Example 11.
- A comparison between GBALR, GBALR1 and GBALR2 shows that GBALR is the superior method since it converges in all examples.

7 Conclusions

The objective of this research was the design and performance evaluation of a neural network architecture, capable of solving polynomial systems of equations. We first compared our approach with some simple systems of non-linear equations having only two or three equations that were recently used in our previous work for analysing the performance of a new proposed method. The results obtained using the proposed adaptive learning rate procedure, are very promising giving exact solutions with very fast convergence. Therefore, the proposed method can be used for solving complicated polynomial systems in applications such as numerical integration, chemical equilibrium, kinematics, neuropsychology and combustion technology.

Regarding larger system dimensions, the GBARL method is capable to solve large sparse systems. In most cases, it is superior than the classical methods with respect to iteration number but for large systems with dimensions $n \geqslant 500$ is characterized by larger CPU time.

Challenges for future research include the use of the network with other activation functions in the output layer, such as the hyperbolic tangent function, as well as the ability of the network to handle situations such that the case of multiple roots (real and complex) for the case of overdetermined and underdetermined systems.

References

[1] Barron, A. R. (1933) Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Trans. Inform. Theory* **39**(3), 930–945.
[2] Broyden, C. G. (1965) A class of methods for solving nonlinear simultaneous equations. *Math. Comput.* **19**(92), 577–593.
[3] Broyden, C. G., Dennis, J. E. & More, J. J. (1993) On the local and the superlineal convergences of quasi-Newton methods. *J. Inst. Math. Appl.* **12**(3), 223–246.
[4] Burden, R. L., Faires, J. D. & Burden, A. M. (2015) *Numerical Analysis*, ISBN 978-1305253667, Brooks Cole, Boston, USA.
[5] Cao, F. & Chen, Z. (2009) The approximation operators with sigmoidal functions. *Comput. Math. Appl.* **58**(4), 758–765.
[6] Cao, F. & Chen, Z. (2012) The construction and approximation of a class of neural networks operators with ramp functions. *J. Comput. Anal. Appl.* **14**(1), 101–112.
[7] Costarelli, D. (2015) Neural network operators: Constructive interpolation of multivariate functions. *Neural Netw.* **67**, 28–36.
[8] Costarelli, D. & Vinti, G. (2016) Max-product neural network and quasi interpolation operators activated by sigmoidal functions. *J. Approx. Theory* **209**, 1–22.

[9] Costarelli, D. & Vinti, G. (2016) Pointwise and uniform approximation by multivariate neural network operators of the max-product type. *Neural Netw.* **81**, 81–90.

[10] Costarelli, D. & Vinti, G. (2016) Approximation by max-product neural network operators of Kantorovich type. *Results Math.* **69**(3), 505–519.

[11] Dennis, J. E. & Wolkowicz, H. (1993) Least change secant method, sizing and shifting. *SIAM J. Numer. Anal.* **30**(5), 1291–1314.

[12] El-Emary, I. M. M. & El-Kareem, M. M. A. (2008) Towards using genetic algorithms for solving nonlinear equation systems. *World Appl. Sci. J.* **5**(3), 282–289.

[13] Goulianas, K., Margaris, A. & Adamopoulos, M. (2013) Finding all real roots of 3×3 nonlinear algebraic systems using neural networks. *Appl. Math. Comput.* **219**(9), 4444–4464.

[14] Grosan, C., Abraham, A. & Snasel, V. (2012) Solving polynomial systems using a modified line search approach. *Int. J. Innovative Comput. Inform. Control* **8**(1) (B), 501–526.

[15] Grosan, C. & Abraham, A. (2012) Multiple solutions of a system of nonlinear equations. *Int. J. Innovative Comput. Inform. Control* **4**(9), 2161–2170.

[16] Hahm, N. & Hong, B. I. (2016) A Note on neural network approximation with a sigmoidal function. *Appl. Math. Sci.* **10**(42), 2075–2085.

[17] Iliev, A., Kyurkchiev, N. & Markov, S. (2015) On the approximation of the cut and step functions by logistic and Gompertz functions. *BIOMATH* **4**(2), 1–12.

[18] Karr, C. L, Weck, B. & Freeman, L. M. (1998) Solution to systems of nonlinear equations via a generic algorithm. *Eng. Appl. Artif. Intell.* **11**, 369–375.

[19] Ko, T. H., Sakkalis, T. & Patrikalakis, N. M. (2004) Nonlinear polynomial systems: Multiple roots and their multiplicities. In: F. Giannini & A. Pasko (editors), *Proceedings of Shape Modelling International Conference, SMI 2004*, Genova, Italy.

[20] Mamat, M., Muhammad, K. & Waziri, M. Y. (2014) Trapezoidal Broyden's method for solving systems of nonlinear equations. *Appl. Math. Sci.* **8**(6), 251–260.

[21] Manocha, D. (1994) Solving Systems of polynomial equations. *IEEE Comput. Graph. Appl.* **14**(2), 46–55.

[22] Margaris, A. & Goulianas, K. (2012) Finding all roots of 2×2 non linear algebraic systems using back propagation neural networks. In: *Neural Computing and Applications*, Vol. 21(5), Springer-Verlag London Limited, pp. 891–904.

[23] Martinez, J. M. (1994) Algorithms for solving nonlinear systems of equations. In: *Continuous Optimization: The State of the Art*, Kluwer, pp. 81–108.

[24] Martynyuk, A. A. (2008) An exploration of polydynamics of nonlinear equations on time scales. *ICIC Exp. Lett.* **2**(2), 155–160.

[25] Mathia, K. & Saeks, R. (1995) Solving nonlinear equations using reccurent neural networks. In: *World Congress on Neural Networks (WCNN'95)*, Washington DC, pages I-76 to I-80.

[26] Mhetre, P. (2012) Genetic algorithm for linear and nonlinear equation. *Int. J. Adv. Eng. Technol.* **3**(2), 114–118.

[27] Mishra, D. & Kalra, P. (2007) Modified hopfield neural network approach for solving nonlinear algebraic equations. *Eng. Lett.* **14**(1), 135–142.

[28] Nguyen, T. T. (1993) Neural network architecture of solving nonlinear equation systems. *Electron. Lett.* **29**(16), 1403–1405.

[29] Oliveira H. A. & Petraglia A. (2013) Solving nonlinear systems of functional equations with fuzzy adaptive simulated annealing. *Appl. Soft Comput.* **13**(11), 4349–4357.

[30] Ortega, J. M. & Rheinboldt, W. C. (1970) *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York.

[31] Rheinboldt, W. C. (1974) Methods for solving systems of equations. *Reg. Conf. Ser. Appl. Math* **14**.

[32] Schmidhuber, J. (2015) Deep learning in neural networks: An overview. *Neural Netw.* **61**, 85–117.

[33] Spedicato, E. & Huang, Z. (1997) Numerical experience with Newton-like methods for nonlinear algebraic systems. *Computing* **58**(1), 69–89.
[34] Zhou, G. & Zhu, P. (2010) Iterative algorithms for solving repeated root of nonlinear equation by modified Adomian decomposition method. *ICIC Exp. Lett*. **4**(2), 595–600.
[35] Zhoug, X., Zhang, T. & Shi, Y. (2009) Oscillation and nonoscillation of neutral difference equation, with positive and negative coefficients. *Int. J. Innovative Comput. Inform. Control* **5**(5), 1329–1342.