Analytic Method for Evaluation of the Weights of a Robust Large-Scale Multilayer Neural Network

Mikael Fridenfalk
Department of Game Design
Uppsala University
Visby, Sweden
mikael.fridenfalk@speldesign.uu.se

***Abstract*—The multilayer feedforward neural network is presently one of the most popular computational methods in computer science. However, the current method for the evaluation of its weights is performed by a relatively slow iterative method known as backpropagation. According to previous research on a large-scale neural network with many hidden nodes, attempts to use an analytic method for the evaluation of the weights by the linear least square method showed to accelerate the evaluation process significantly. Nevertheless, the evaluated network showed in preliminary tests to fail in robustness compared to well-trained networks by backpropagation, thus resembling overtrained networks. This paper presents the design and verification of a new method that solves the robustness issues for such a neural network, along with MATLAB code for the verification of key experiments.**

Keywords–analytic; big data; FNN; large-scale; least square method; multilayer; neural network; robust; sigmoid.

I. Introduction

As an extension of an earlier work presented in ADVCOMP 2014 [1], this paper reconfirms the initial inference by the presentation of a new layer of experiments. As a brief introduction, the artificial neural network constitutes one of the most useful and popular computational methods in computer science. The most well-known category is the multilayer Feedforward Neural Network, in this paper abbreviated as FNN, where the weights are estimated by an iterative training method called backpropagation [2], [3]. Although backpropagation is relatively fast for small networks, it is rather slow for large ones, given the computational power of modern computers [4], [5]. To accelerate the training speed of FNNs, many approaches have been suggested based on the least square method [6]. Although the presentation on the implementation, as well as of the data on the robustness of these methods may be improved, the application of the least square method seems to be a promising path to investigate [7], [8].

What we presume to be required for a new method to replace backpropagation in such networks, is not only that it is efficient, but also that it is superior compared to existing methods and is easy to understand and implement. Therefore, the goal of this work has been to investigate the possibility to find a *robust analytic solution* (i.e., with good generalization abilities compared with a well-trained network using backpropagation, but without any iterations involved), for the weights of an FNN, which is easily understood and may be implemented relatively effortlessly, using a mathematical application such as MATLAB [9].

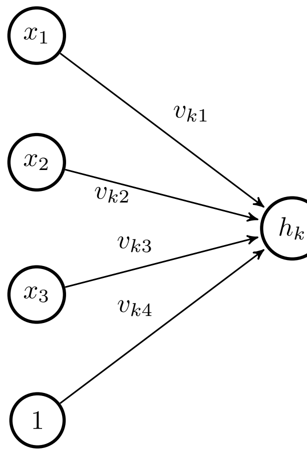

Figure 1. An example with three input nodes ($M = 3$), $h_k = S(\mathbf{v}_k\mathbf{u}) = S(v_{k1}x_1 + v_{k2}x_2 + v_{k3}x_3 + v_{k4})$, using a sigmoid activation function S.

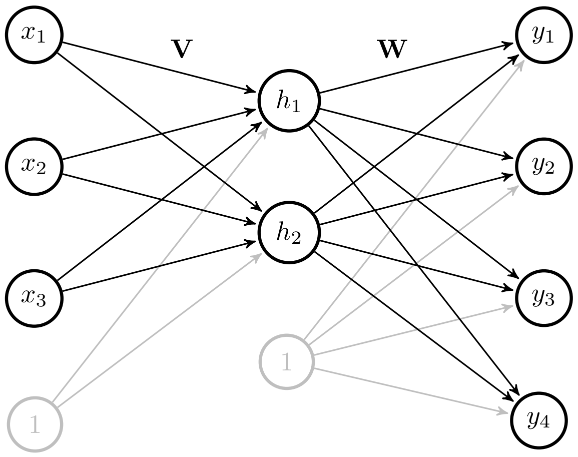

Figure 2. A vectorized model of a standard FNN with a single hidden layer, in this example with $M = 3$ input nodes, $H = 2$ hidden nodes, $K = 4$ output nodes and the weight matrices $\mathbf{V}$ and $\mathbf{W}$, using a sigmoid activation function for the output of each hidden node. In this model, the biases for the hidden layer and the output layer correspond to column $M + 1$ in $\mathbf{V}$ versus column $H + 1$ in $\mathbf{W}$.

II. Overview

As an overview of the main structure of this paper, in Section III, the history of artificial neural networks is briefly reiterated. In Sections IV-V a recap is made of the theory behind the fundamentals of the analytic method presented in this paper. In Section VI, the derivation of an upgrade is reiterated for the improvement of the robustness of the proposed analytic method. In Section VII, the experimental setup is briefly described for the experiments presented in this paper, using a mathematical engine based on C++. In the results sections, Section VIII presents experiments for the evaluation of the original analytic solution we proposed in [1], in this paper labeled as the *initial experiments*. Section IX presents an evaluation of an upgrade of the original proposal, denoted as *diagonal reinforcement* [10], and labeled as the *primary experiments*. Section X presents a MATLAB-based version of the experiments presented in Figures 4-9, thereby additionally validating the experiments presented in Section IX, thus fulfilling the initial goal of this project, as formulated in [1], and the introduction section.

III. Background

The history of artificial neural networks is considered to have started in 1943 [11]. The 1950s and 1960s is often regarded as a golden age for neural networks, marked among other things by the development of the first successful neurocomputer, Mark I Perceptron [12]. However, towards the end of the 1960s, this age was turned into an ice age after the criticism of the perceptron model [13]. In the following decades the neural networks research slowly recovered by the introduction of multilayer networks and the introduction of backpropagation [12], [14]. Backpropagation is a slow iterative method for the evaluation of the weights of multilayer neural networks. During the last decades a large number of neural networks have been suggested, but presently the evaluation of the weights of the most well-known and widely used category, namely the FNN, is still based on backpropagation, which so far has made many mathematicians to avoid this field, since an analytic method for the evaluation of the weights of robust FNNs is considered today to be missing.

IV. Related Work

In a previous work [1], an analytic solution was proposed for the evaluation of the weights of a textbook FNN. This solution was found to be significantly much faster, and for $H = N - 1$ (where H denotes the number of hidden nodes and N, the number of training points), more accurate than solutions provided by backpropagation, but at the same time significantly less robust (e.g., more noise sensitive) compared to a well-trained network using backpropagation, why the analytic solution was in this context considered to lack robustness for direct use.

However, further experiments showed that even small measures, such as an increase in the input range of the network by the duplication of the training set, with the addition of perturbation, led to significant improvement of the robustness of the evaluated network. As a systematic attempt to address the issue of robustness, this paper presents the derivation, implementation and further verification of the new method proposed in [10], based on the expansion of the training set of an FNN, with addition of perturbation, but in practice without any impact on the execution speed of the original method introduced in [1].

V. An Analytic Solution

In [1], a textbook FNN is vectorized based on a sigmoid activation function $S(t) = 1/(1 + e^{-t})$. The weights $\mathbf{V}$ and $\mathbf{W}$ of such system (often denoted as W_{IH} versus W_{HO}), may be represented by Figures 1-2. In this representation, defined here as the normal form, the output of the network may be expressed as:

$$\mathbf{y} = \mathbf{W}\mathbf{h} = \mathbf{W}\left[\frac{S(\mathbf{V}\mathbf{u})}{1}\right],\ \mathbf{u} = \begin{bmatrix}\mathbf{x}\\1\end{bmatrix} \tag{1}$$

where $\mathbf{x} = [x_1\ x_2\ \ldots\ x_M]^T$ denotes the input signals, $\mathbf{y} = [y_1\ y_2\ \ldots\ y_K]^T$ the output signals, and S, an element-wise sigmoid function. In this paper, a winner-take-all classification model is used, where the final output of the network is the selection of the output node that has the highest value. Since the sigmoid function is constantly increasing and identical for each output node, it can be omitted from the output layer, as $\max(\mathbf{y})$ results in the same node selection as $\max(S(\mathbf{y}))$. Further on, presuming that the training set is highly fragmented (the input-output relations in the training sets were in our experiments established by a random number generator), the number of hidden nodes is in many experiments set to $H = N - 1$. Defining a batch of input signals, e.g., a training set, the input matrix $\mathbf{U}$ may be expressed as:

$$\mathbf{U} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N}\\ x_{21} & x_{22} & \cdots & x_{2N}\\ \vdots & \vdots & \ddots & \vdots\\ x_{M1} & x_{M2} & \cdots & x_{MN}\\ 1 & 1 & \cdots & 1 \end{bmatrix} \tag{2}$$

where column vector i in $\mathbf{U}$, corresponds to training point i, column vector i in $\mathbf{Y}_0$ (target output value) and in $\mathbf{Y}$ (actual output value). Further, defining $\mathbf{H}$ of size $N \times N$, as the batch values for the hidden layer, given a training set of input and output values and $M^+ = M + 1$, the following relations hold:

$$\mathbf{U} = \left[\frac{\mathbf{X}}{\mathbf{1}^T}\right] : [M^+ \times N] \tag{3}$$

$$\mathbf{H} = \left[\frac{S(\mathbf{V}\mathbf{U})}{\mathbf{1}^T}\right] : [N \times N] \tag{4}$$

$$\mathbf{Y} = \mathbf{W}\mathbf{H} : [K \times N] \tag{5}$$

To evaluate the weights of this network analytically, we need to evaluate the target values (points) of $\mathbf{H}_0$ for the hidden layer. In this context, the initial assumption is that any point is feasible, as long as it is unique for each training set. Therefore, in this model, $\mathbf{H}_0$ is composed of random numbers. Thus, the following evaluation scheme is suggested for the analytic solution of the weights of such network:

$$\mathbf{V}^T = (\mathbf{U}\mathbf{U}^T)^{-1}\mathbf{U}\mathbf{H}_0^T : [M^+ \times H] \tag{6}$$

$$\mathbf{W}^T = (\mathbf{H}\mathbf{H}^T)^{-1}\mathbf{H}\mathbf{Y}_0^T : [N \times K] \tag{7}$$

where a linear least square solution is used for the evaluation of each network weight matrix. Such equation is nominally expressed as:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{8}$$

with the least square solution [6]:

$$\mathbf{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} \tag{9}$$

Since the mathematical expressions for the analytic solution of the weights of a neural network may be difficult to follow, an attempt has been made in Figure 3 to visualize the matrix operations involved. While a nonlinear activation function (such as the sigmoid function) is vital for the success of such network, the inclusion of a bias is not essential. It is for instance possible to omit the biases and to replace $\mathbf{H}_0$ with an identity matrix $\mathbf{I}$. Such a configuration would instead yield the following formula for the evaluation of $\mathbf{V}$ and $\mathbf{H}$ (where $\mathbf{UI}$ can further be simplified as $\mathbf{U}$):

$$\mathbf{V}^T = (\mathbf{U}\mathbf{U}^T)^{-1}\mathbf{UI} : [M^+ \times N] \tag{10}$$

$$\mathbf{H} = S(\mathbf{VU}) : [H \times N] \tag{11}$$

VI. Diagonal Reinforcement

To recap the theory on diagonal reinforcement, as proposed in [10], we expand the input training set $\mathbf{U}$ in (2), by the addition of perturbation to the input signal, given the definition $\mathbf{\Theta} = \mathbf{U}\mathbf{U}^T$, with $\theta_{M^+M^+} = N$ in:

$$\mathbf{\Theta} = \begin{bmatrix} \theta_{11} & \theta_{12} & \dots & \theta_{1M} & \theta_{1M^+} \\ \theta_{21} & \theta_{22} & \dots & \theta_{2M} & \theta_{2M^+} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \theta_{M1} & \theta_{M2} & \dots & \theta_{MM} & \theta_{MM^+} \\ \theta_{M+1} & \theta_{M+2} & \dots & \theta_{M+M} & N \end{bmatrix} \tag{12}$$

Further, an extended matrix $\tilde{\mathbf{U}}$ is introduced, where:

$$\tilde{\mathbf{U}} = \begin{bmatrix} \tilde{\mathbf{U}}_1 & \tilde{\mathbf{U}}_2 & \dots & \tilde{\mathbf{U}}_N \end{bmatrix} \tag{13}$$

with:

$$\mathbf{U}_j = \begin{bmatrix} u_{1j}+\Delta & u_{1j}-\Delta & u_{1j} & u_{1j} & \cdots & u_{1j} & u_{1j} \\ u_{2j} & u_{2j} & u_{2j}+\Delta & u_{2j}-\Delta & \cdots & u_{2j} & u_{2j} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ u_{Mj} & u_{Mj} & u_{Mj} & u_{Mj} & \cdots & u_{Mj}+\Delta & u_{Mj}-\Delta \\ 1 & 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix} \tag{14}$$

and where Δ is defined as the amplitude of the perturbation. Thus, for the right hand side of (8), $\tilde{\mathbf{\Theta}} = \tilde{\mathbf{U}}\tilde{\mathbf{U}}^T$, or more explicitly:

$$\tilde{\mathbf{\Theta}} = 2M \begin{bmatrix} d_1 & \theta_{12} & \dots & \theta_{1M} & \theta_{1M^+} \\ \theta_{21} & d_2 & \dots & \theta_{2M} & \theta_{2M^+} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \theta_{M1} & \theta_{M2} & \dots & d_M & \theta_{MM^+} \\ \theta_{M+1} & \theta_{M+2} & \dots & \theta_{M+M} & N \end{bmatrix} \tag{15}$$

with $d_i = \theta_{ii} + \alpha$, where $\alpha = N\Delta^2/M$, or:

$$\tilde{\mathbf{\Theta}} = 2M\left[\mathbf{\Theta} + \text{diag}(\alpha, \alpha, \dots, \alpha, 0)\right] \tag{16}$$

where diag$(d_1, d_2, \dots, d_{M^+})$ denotes a diagonal matrix of size $M^+ \times M^+$ (where $M^+ = M+1$), with the diagonal elements $d_1, d_2, \dots, d_{M^+}$. Similarly, for the left hand side of (8), $\mathbf{\Psi}$ and $\mathbf{\Lambda}$ are defined as:

$$\mathbf{H}_0^T = \mathbf{\Psi} = \begin{bmatrix} \psi_{11} & \psi_{12} & \dots & \psi_{1H} \\ \psi_{21} & \psi_{22} & \dots & \psi_{2H} \\ \vdots & \vdots & \ddots & \vdots \\ \psi_{N1} & \psi_{N2} & \dots & \psi_{NH} \end{bmatrix} \tag{17}$$

$$\mathbf{\Lambda} = \mathbf{U}\mathbf{H}_0^T = \begin{bmatrix} \lambda_{11} & \lambda_{12} & \dots & \lambda_{1H} \\ \lambda_{21} & \lambda_{22} & \dots & \lambda_{2H} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{M+1} & \lambda_{M+2} & \dots & \lambda_{M+H} \end{bmatrix} \tag{18}$$

and thereby, $\mathbf{\Psi}$ and $\mathbf{\Psi}_j$ as:

$$\mathbf{\Psi} = \begin{bmatrix} \mathbf{\Psi}_1 \\ \mathbf{\Psi}_2 \\ \vdots \\ \mathbf{\Psi}_N \end{bmatrix} \tag{19}$$

$$\mathbf{\Psi}_j = \begin{bmatrix} \psi_{j1} & \psi_{j2} & \dots & \psi_{jH} \\ \psi_{j1} & \psi_{j2} & \dots & \psi_{jH} \\ \vdots & \vdots & \ddots & \vdots \\ \psi_{j1} & \psi_{j2} & \dots & \psi_{jH} \end{bmatrix} \tag{20}$$

with $\tilde{\mathbf{\Lambda}} = \tilde{\mathbf{U}}\mathbf{\Psi}$:

$$\tilde{\mathbf{\Lambda}} = 2M\mathbf{\Lambda} \tag{21}$$

This transforms (8) into:

$$\tilde{\mathbf{U}}\tilde{\mathbf{U}}^T\mathbf{X} = \tilde{\mathbf{U}}\mathbf{\Psi} \tag{22}$$

or:

$$\tilde{\mathbf{\Theta}}\mathbf{X} = \tilde{\mathbf{\Lambda}} \tag{23}$$

Thus:

$$2M\left[\mathbf{\Theta} + \text{diag}(\alpha, \alpha, \dots, \alpha, 0)\right]\mathbf{X} = 2M\mathbf{\Lambda} \tag{24}$$

Given the matrix equation:

$$\mathbf{AX} = \mathbf{B} \tag{25}$$

since, given a scalar $c \in \mathbb{R}$:

$$c \cdot (\mathbf{AX}) = (c \cdot \mathbf{A})\mathbf{X} = c \cdot \mathbf{B} \tag{26}$$

thereby:

$$\left[\mathbf{\Theta} + \text{diag}(\alpha, \alpha, \dots, \alpha, 0)\right]\mathbf{X} = \mathbf{\Lambda} \tag{27}$$

This yields thus, the final expression:

$$\left[\mathbf{U}\mathbf{U}^T + \text{diag}(\alpha, \alpha, \dots, \alpha, 0)\right]\mathbf{X} = \mathbf{U}\mathbf{H}_0^T \tag{28}$$

Hence, the expansion of $\mathbf{U}$ into a perturbation matrix $\tilde{\mathbf{U}}$ of size $M^+ \times 2MN$, and similarly of $\mathbf{H}_0^T$ into a matrix $\mathbf{\Psi}$ of size $2MN \times H$, is according to (28) equivalent to the reinforcement of the diagonal elements of the square matrix $\mathbf{\Theta} = \mathbf{U}\mathbf{U}^T$, by the addition of a scalar $\alpha = N\Delta^2/M$ to each diagonal element, except for the last one, which as a consequence of the use of bias in the network, is left intact.

VII. Experimental Setup

The experiments presented in Sections VIII-IX, are based on a minimal mathematical engine that was developed in C++, with the capability to solve $\mathbf{X}$ in a linear matrix equation system of the form $\mathbf{AX} = \mathbf{B}$, where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{X}$ denote matrices of appropriate sizes, since it is computationally more efficient to solve a linear equation system directly, than by matrix inversion. In this system, the column vectors of $\mathbf{X}$ are evaluated using a single Gauss-Jordan elimination cycle [6], where each column vector $\mathbf{x}_i$ in $\mathbf{X}$ corresponds to the column vector $\mathbf{b}_i$ in $\mathbf{B}$. Backpropagation was in these experiments, for high execution speed (and a fair comparison with the new methods), also implemented in C++, using the code

$$\underbrace{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}}_{\mathbf{V}^T} = \underbrace{\left(\begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} * & * & 1 \\ * & * & 1 \\ * & * & 1 \\ * & * & 1 \\ * & * & 1 \\ * & * & 1 \end{bmatrix} \right)^{-1}}_{\underbrace{\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}}_{(\mathbf{UU}^T)^{-1}}} \underbrace{\left[\begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix} \right]}_{\underbrace{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}}_{\mathbf{UH}_0^T}}$$

$$\underbrace{\begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}}_{\mathbf{H}} = \left[\frac{S\left(\underbrace{\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}}_{\mathbf{V}} \underbrace{\begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}}_{\mathbf{U}} \right)}{\underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}}_{\mathbf{1}^T}} \right]$$

$$\underbrace{\begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix}}_{\mathbf{W}^T} = \underbrace{\left(\begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} * & * & * & * & * & 1 \\ * & * & * & * & * & 1 \\ * & * & * & * & * & 1 \\ * & * & * & * & * & 1 \\ * & * & * & * & * & 1 \\ * & * & * & * & * & 1 \end{bmatrix} \right)^{-1}}_{(\mathbf{HH}^T)^{-1}} \underbrace{\left[\underbrace{\begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}}_{\mathbf{H}} \underbrace{\begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix}}_{\mathbf{Y}_0^T} \right]}_{\underbrace{\begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix}}_{\mathbf{HY}_0^T}}$$

$$\underbrace{\begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{bmatrix}}_{\mathbf{Y}} = \underbrace{\begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{bmatrix}}_{\mathbf{W}} \underbrace{\begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}}_{\mathbf{H}}$$

Figure 3. A visual representation of the evaluation of weights $\mathbf{V}$ and $\mathbf{W}$ by the analytic method presented in the initial work [1], and the actual output $\mathbf{Y}$, in this example as a function of six training points, $N = 6$, the training input and output sets $\mathbf{U}$ (for a simplified notation, whenever the relation $\mathbf{U} = \mathbf{U}_0$ is implicit) and $\mathbf{Y}_0$, with two inputs, $M = 2$, four outputs, $K = 4$, and five hidden nodes, $H = N - 1 = 5$. In this figure, an asterisk denotes a floating-point number. To facilitate bias values, certain matrix elements are set to one.

presented in [15] as a reference. To measure the efficiency of the new method in [1], compared with the standard method (backpropagation), the standard textbook definition for the mean-squared error was used:

$$\epsilon = \frac{1}{2N} \sum_{i}^{K} \sum_{j}^{N} (a_{ij} - y_{ij})^2 \tag{29}$$

where a_{ij} denotes an element in $\mathbf{Y}_0$ (target value), and y_{ij} the corresponding element in $\mathbf{Y}$ (actual value). Although the new method, as in [1], does not intrinsically benefit from such definition (since there is no need here for the differentiation of the mean-squared error, which however, is essential for backpropagation), to simplify comparison in Tables I-V, the same definition of the mean-squared error was also applied to the new method.

VIII. Initial Experiments

The experimental results presented in this section, as shown in Tables I-V, are based on ten individual experiments for each parameter setting using different random seeds, where $\bar{t}$ denotes average execution time, using a single CPU core on a modern laptop computer (the same computer was used for all experiments presented in this paper), $\bar{\epsilon}$, the average value of the mean-squared errors, and $\tilde{\epsilon}$, the median value. The success rate, $\bar{s}$, is similarly based on an average value. Since the variation of the results is large between the experiments, the average values are in general larger than the median values. If the number of experiments per parameter setting is increased, the average value tends to increase as well.

On a note of preliminary experiments with respect to robustness, regarding the generalization abilities of the network, the original method (i.e., without the application of diagonal reinforcement) showed to steeply lose accuracy with the addition of noise to the input values, compared with a network trained by backpropagation. This shows that although the results seem to be in order according to Tables I-V, the original method lacks robustness for direct use. However, further experiments showed that even small measures, such as an increase in the input range of the network by the duplication

Table I. Initial experiments (Tables I-V) – Backpropagation with $H = N - 1$ and average success rate $\bar{s}$.

M	N	H	K	Iterations	$\bar{t}$	$\bar{\epsilon}$	$\tilde{\epsilon}$	$\bar{s}$ (%)
5	20	19	5	10^4	46.6 ms	0.0671	0.0620	93.0
5	20	19	5	10^6	4.39 s	0.0175	$4.64 \cdot 10^{-5}$	96.5
10	50	49	10	10^4	182 ms	0.116	0.114	83.2
10	50	49	10	10^6	18.1 s	0.0680	0.0650	86.4
20	50	49	20	10^4	333 ms	0.0394	0.0392	94.6
20	50	49	20	10^6	33.3 s	0.0170	0.0200	96.6
40	100	99	40	10^4	1.27 s	0.0671	0.0670	88.9
40	100	99	40	10^6	127 s	0.0180	0.0200	96.4

Table II. New method with $H = N - 1$.

M	N	H	K	$\bar{t}$	$\bar{\epsilon}$	$\tilde{\epsilon}$	$\bar{s}$ (%)
5	20	19	5	332 μs	$3.34 \cdot 10^{-8}$	$2.00 \cdot 10^{-13}$	100.0
10	50	49	10	3.74 ms	$6.99 \cdot 10^{-9}$	$2.26 \cdot 10^{-12}$	100.0
20	50	49	20	4.79 ms	$2.68 \cdot 10^{-13}$	$5.93 \cdot 10^{-16}$	100.0
40	100	99	40	36.7 ms	$3.93 \cdot 10^{-12}$	$2.33 \cdot 10^{-13}$	100.0

Table III. Backpropagation with $H < N - 1$ and 10^4 iterations.

M	N	H	K	$\bar{t}$	$\bar{\epsilon}$	$\tilde{\epsilon}$	$\bar{s}$ (%)
5	20	5	5	14.7 ms	0.130	0.125	86.5
10	50	10	10	43.1 ms	0.227	0.237	71.6
20	50	20	20	144 ms	0.0624	0.0599	94.2
40	100	40	40	531 ms	0.115	0.115	84.8

Table IV. New method with $H < N - 1$.

M	N	H	K	$\bar{t}$	$\bar{\epsilon}$	$\tilde{\epsilon}$	$\bar{s}$ (%)
5	20	5	5	59 μs	0.281	0.289	58.5
10	50	10	10	370 μs	0.350	0.350	50.0
20	50	20	20	1.30 ms	0.279	0.277	91.8
40	100	40	40	9.24 ms	0.290	0.290	98.5

Table V. Selective use of bias for the new method, with $M = 40$, $N = 100$, $H = 99$, and $K = 40$.

H_b	Y_b	$\bar{t}$	$\bar{\epsilon}$	$\tilde{\epsilon}$	$\bar{s}$ (%)
No	No	35.7 ms	0.00514	0.00513	100.0
No	Yes	36.4 ms	$1.35 \cdot 10^{-12}$	$1.91 \cdot 10^{-14}$	100.0
Yes	No	36.2 ms	0.00544	0.00534	100.0

Table VI. Average execution time, $\bar{t}_{\text{bp}}$ (backprop.), $\bar{t}_{\text{new}}$ (initial), $\bar{t}_{\text{new+}}$ (primary), $\bar{t}_{\text{new*}}$ (verification).

Figure	M	N	H	K	$\bar{t}_{\text{bp}}$	$\bar{t}_{\text{new}}$	$\bar{t}_{\text{new+}}$	$\bar{t}_{\text{new*}}$
4	10	25	24	10	92.5 ms	688 μs	668 μs	326 μs
5	20	50	49	20	332 ms	4.84 ms	4.86 ms	559 μs
6	40	100	99	40	1.27 s	37.0 ms	37.1 ms	1.47 ms
7	10	25	10	10	43.3 ms	212 μs	202 μs	175 μs
8	20	50	20	20	144 ms	1.33 ms	1.31 ms	306 μs
9	40	100	40	40	535 ms	9.35 ms	9.38 ms	631 μs

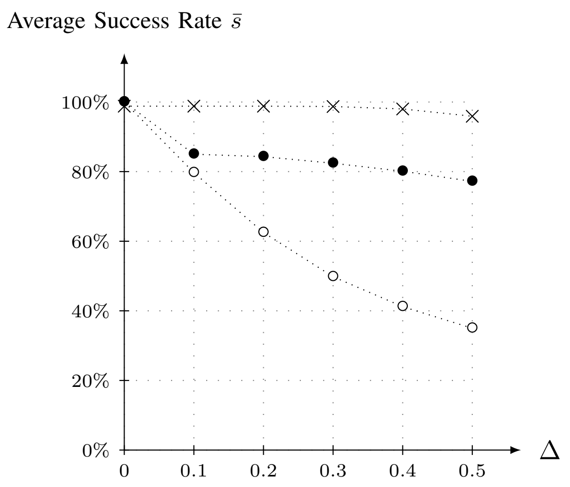

Figure 4. Backpropagation (×), initial analytic method (○), versus new method using diagonal reinforcement (●), with $M = 10$ (input nodes), $N = 25$ (training points), $H = 24$ (hidden nodes), and $K = 10$ (output nodes).

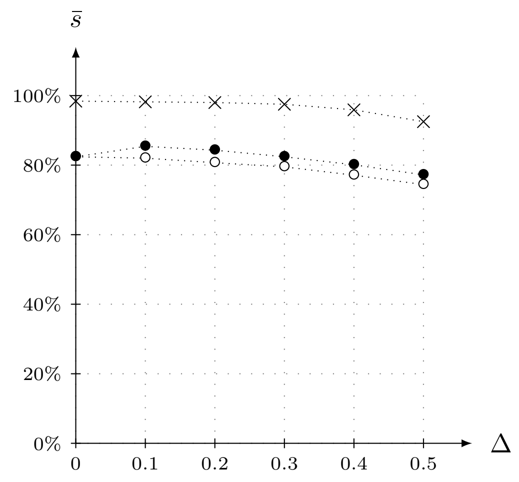

Figure 7. $M = 10$, $N = 25$, $H = 10$, and $K = 10$.

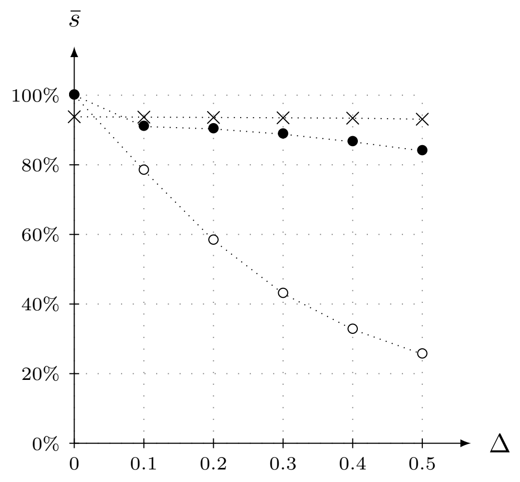

Figure 5. $M = 20$, $N = 50$, $H = 49$, and $K = 20$.

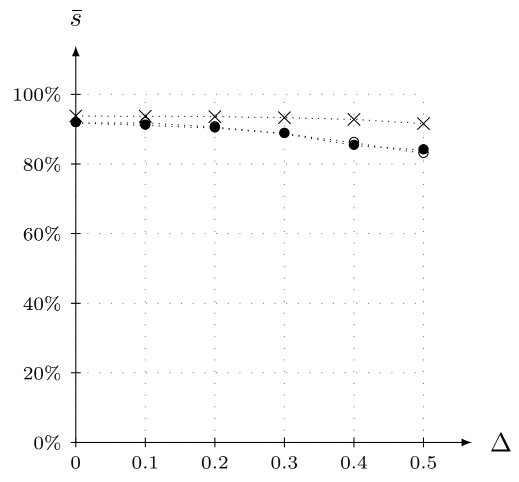

Figure 8. $M = 20$, $N = 50$, $H = 20$, and $K = 20$.

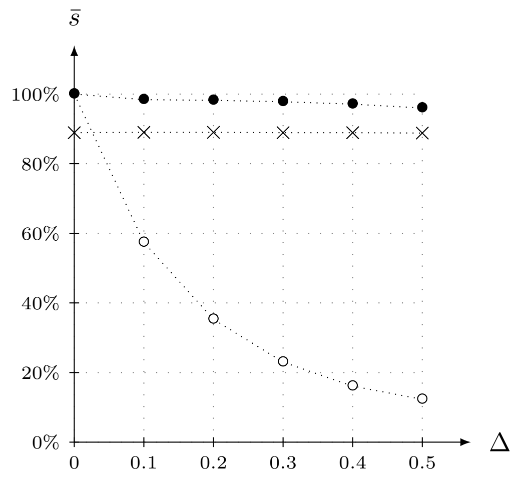

Figure 6. $M = 40$, $N = 100$, $H = 99$, and $K = 40$.

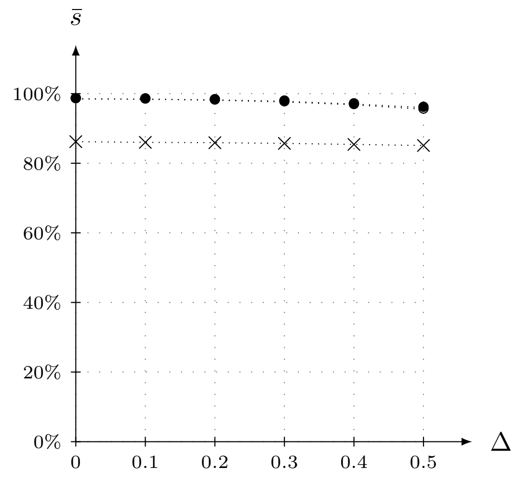

Figure 9. $M = 40$, $N = 100$, $H = 40$, and $K = 40$.

of the training set with the addition of perturbation and a more conscious design of $\mathbf{H}_0$, by for instance the clustering of the random values as a function of the output values, or for layers with few hidden nodes, the binary encoding [16] of $\mathbf{H}_0$, led to significant improvements of the robustness of the new method. Therefore, this was an indication at an early stage of the development of the method presented in this paper, that these robustness issues could be solved, and in this context, without any significant impact to the computational speed of the initial version of the new method (i.e., without diagonal reinforcement).

IX. Primary Experiments

The experimental results presented in this section, as shown in Figures 4-9, are in similarity with previous section based on a mathematical engine developed in C++, here in addition examining the effects of diagonal reinforcement, as described in Section VI, measuring average success rate, and Table VI, measuring execution speed. Each experiment is based on ten individual experiments (with different random seeds), using a single CPU core. In Table VI, $\bar{t}_{\text{bp}}$ denotes the execution time for backpropagation based on 10000 iterations, which applies to all backpropagation experiments presented in this paper. Similarly, $\bar{t}_{\text{new}}$ denotes the execution time for the original analytic method in [1], and $\bar{t}_{\text{new+}}$, the execution time for the new method, using diagonal reinforcement.

In all the experiments presented in this paper, the input values to the FNN is based on the integers $\{0, 1, 2\}$, and the output values of a binary number, $\{0, 1\}$. To avoid inconsistencies (or repetition) in any training set, no identical input vectors are permitted. For the addition of noise, a random value (with uniform distribution) in the range of $\pm\Delta$, was added to each input value. However, although according to our derivation of (28), $\alpha = N\Delta^2/M$, α had in practice to be retuned to $10^5 \cdot N\Delta^2$ for good results in the experiments in Figures 4-6 ($H = N - 1$), and to $10^4 \cdot N\Delta^2$ in Figures 7-9 (few hidden nodes).

X. Matlab Verification Experiments

This paper presents a verification of previously presented results in [10], but here based on MATLAB. As initially inferred in [1], the goal in this work is to develop a method that is efficient, superior compared to existing methods, and in addition easy to understand and implement using a mathematical application such as MATLAB.

Thus, to verify the results presented in Figures 4-9, which were based on our own computational engine developed in C++, we hereby present a MATLAB version of the same solution, as presented in Figure 10 (and verified by the auxiliary MATLAB code in Figures 11-13), corresponding to the same experiments as in Figures 4-9.

The MATLAB code in Figures 10-13, produces numerical results that coincide relatively accurately (considering the use of random numbers) with previous evaluations using the engine based on C++. The execution speed proved to be faster, as shown in Table VI, compared to the C++ engine. According to this table, while for smaller matrix sizes, the execution speed is only marginally faster for MATLAB (defined by execution time, $\bar{t}_{\text{new*}}$), in comparison with the C++ engine ($\bar{t}_{\text{new+}}$), however, for larger matrices, MATLAB is significantly faster. One reason is that while MATLAB is multithreaded, the C++ engine uses only a single thread. In addition, while the core computational engine of MATLAB is expected to be implemented by assembly code, we used regular C++ code, prioritizing code readability. MATLAB is in addition expected to use smart tricks to accelerate matrix operations, including using CPU cores very efficiently.

However, as a note, although MATLAB in average showed to be faster than our C++ engine, the application proved to be initially slower at a startup phase, running the MATLAB master script in Figure 13. This script is thus, for a more representative time measurement, recommended to be executed twice.

Since the MATLAB code presented in this paper constitutes the actual results of this work, we have below included a brief explanation of this code. To facilitate direct copy and paste of this code from a PDF version of this document into the suggested filenames (e.g., EvalVW.m for the code in Figure 10), the code have been placed in a single column environment and without line numbers. In addition, on two instances, a comma is placed after an end-statement, since copy and paste may place two end-statements after one another, which presently generates an error in MATLAB. The commas are not visible in the code (due to white font color), but appear after a paste operation into an m-file.

A. *EvalVW.m*

The function EvalVW constitutes the core results of this paper. The MATLAB function `rand(h,n)` creates a matrix of size $H \times N$, consisting of random numbers within the interval of 0 to 1. Regarding the evaluation of $\mathbf{Q}$, note that only the first M diagonal elements are affected by the application of diagonal reinforcement. In the evaluation of $\mathbf{W}$, since $\mathbf{HH}^T$ showed in our experiments to often be a near singular matrix, MATLAB will by default issue a warning statement at execution, unless the warning messages are temporarily turned off and on again by the commands `warning('off','all')` and `warning('on','all')`.

B. *GenTrainingSet.m*

This function calculates a training set, i.e., a complete set of input and output signals for the training of an FNN. The generation of the input matrix $\mathbf{U}_0$ is straightforward and concise, yielding unique columns of values in $\{0, 1, 2\}$, based on random numbers. As a note on the generation of $\mathbf{Y}_0$, which is also based on random numbers, this loop-based solution showed to be both straightforward and fast.

C. *Exp.m*

This inner loop experimental function is used for the complete evaluation of a single point in Figures 4-9. As shown here, N1 denotes the number of FNNs that are generated and tested, and N2, the number of input/output tests performed on each evaluated FNN. On time measurements, since we are only interested in the time it takes to evaluate the weights $\mathbf{V}$ and $\mathbf{W}$, the `tic` (reset and start) and `toc` (stop) MATLAB timer commands, are only placed around the function EvalVW.

To obtain the exact same results each time the function Exp is called, using the same input parameters, the function `rng` is used to set the seed of the random number generator in MATLAB. This is actually not necessary at this stage of the code development, but is often useful at a development stage of a system based on random numbers.

As a potential pitfall, note that the disturbance added to $\mathbf{U}$, is only allowed to affect the first M rows of this matrix (i.e., the input signals), and thus not the bias row, which by definition is always set to a column vector of ones.

The logical expression I0 == I, yields finally a value of one, for each element of I0 that is equal to the same element in I, else zero. This expression is used for the evaluation of the success rate of the network, by comparing the expected output $\mathbf{Y}_0$, with the actual output $\mathbf{Y}$.

D. NNX.m

The master script NNX.m (corresponding to the verification experiments presented in this paper) generates results that correspond to Figures 4-9, except for backpropagation, which was not implemented in MATLAB, but only in C++, since while matrix operations are efficient in MATLAB, the handling of iterations and scalars are, as a general rule, slower than C++.

XI. Discussion

The strength of an analytic method is clear in the sense that it could potentially expand the use of FNNs to a wider range of applications. However, at this stage, there are also some question marks that could require further study.

The first of these is the question of the use of diagonal reinforcement, compared with the initial method in [1], since as shown in Figures 4-9, almost the same effect can be produced by the reduction of the number of hidden nodes. From this perspective, diagonal reinforcement serves mainly to generalize the initial method as proposed in [1].

In addition, two observations that caught our attention during the experiments were that the matrix $\mathbf{HH}^T$ in many cases was nearly singular in the evaluation of $\mathbf{W}$, and that α had to be set to a very high value in comparison to $N\Delta^2/M$, to have the intended effect. Thus, although the analytic method presented in this paper seems to operate correctly for large-scale FNNs, further analysis is recommended to shed light on its inner workings.

XII. Conclusion

The method presented in this paper provides for a robust analytic solution of the weights of a large-scale FNN (i.e., as previously defined, with good generalization abilities compared with a well-trained network using backpropagation, but without any iterations involved), which is significantly faster compared with backpropagation, yet straightforward to implement, using a mathematical application such as MATLAB.

Examples of a few application areas that could benefit from this method are artificial intelligence, economics, control theory, and in general any field dealing with big data for decision making, such as cancer treatment and research.

References

[1] M. Fridenfalk, “The Development and Analysis of Analytic Method as Alternative for Backpropagation in Large-Scale Multilayer Neural Networks,” in ADVCOMP 2014: The Eighth International Conference on Advanced Engineering Computing and Applications in Sciences, Rome, Italy, August 2014, pp. 46–49.

[2] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach. 3rd ed., Prentice Hall, 2009, pp. 727–736.

[3] B. J. Wythoff, “Backpropagation Neural Networks: A Tutorial,” in Chemometrics and Intelligent Laboratory Systems, vol. 18, no. 2, 1993, pp. 115–155.

[4] P. D. Wilde, Neural Networks Models: An Analysis. Springer, 1996, pp. 35–51.

[5] R. P. W. Duin, “Learned from Neural Networks,” in ASCI 2000, Lommel, Belgium, 2000, pp. 9–13.

[6] C. H. Edwards and D. E. Penney, Elementary Linear Algebra. Prentice Hall, 1988, pp. 220–227.

[7] B. Widrow and M. A. Lehr, “30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation,” in Proceedings of the IEEE, vol. 78, no. 9, 1990, pp. 1415–1442.

[8] Y. Yam, “Accelerated Training Algorithm for Feedforward Neural Networks Based on Least Squares Method,” in Neural Processing Letters, vol. 2, no. 4, 1995, pp. 20–25.

[9] “MATLAB, The MathWorks, Inc.” 2015, URL: http://www.mathworks.com/ [accessed: 2015-11-24].

[10] M. Fridenfalk, “Method for Analytic Evaluation of the Weights of a Robust Large-Scale Multilayer Neural Network with Many Hidden Nodes,” in ICSEA 2014: The Proceedings of the Ninth International Conference on Software Engineering Advances, Nice, France, October 2014, pp. 374–378.

[11] W. S. McCulloch and W. Pitts, “A Logical Calculus of the Ideas Immanent in Nervous Activity,” in Bulletin of Mathematical Biophysics, vol. 5, 1943, pp. 115–133.

[12] N. Yadav, A. Yadav, and M. Kumar, “History of Neural Networks,” in An Introduction to Neural Network Methods for Differential Equations. Springer, 2015, pp. 13–15.

[13] M. Minsky and S. Papert, Perceptrons. MIT Press, Cambridge, 1969.

[14] P. J. Werbos, “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences,” Ph.D. dissertation, Harvard University, 1974.

[15] M. T. Jones, AI Application Programming. 2nd ed., Charles River, 2005, pp. 165–204.

[16] F. Gray, “Pulse Code Communication,” U.S. Patent no. 2 632 058, 1947.

```
function [V,W] = EvalVW(U0,Y0,h,d)
[mp,n] = size(U0);
H0 = rand(h,n);
Q = U0 * U0' + diag([d * ones(1,mp-1) 0]);
V = (Q\(U0 * H0'))';
H = [1./(1 + exp(-V*U0)); ones(1,n)];
warning('off','all');
W = ((H * H')\(H * Y0'))';
warning('on','all');
```

Figure 10. The results of this paper, condensed as the MATLAB function EvalVW.m.

```
function [U0,Y0] = GenTrainingSet(m,n,k)
ok = 0;
for i = 1:1000,
    UX = randi([0 2],floor(1.5*n),m);
    U0 = unique(UX,'rows','stable')';
    if size(U0,2) >= n, ok = 1; break; end
end
if ok, U0 = U0(:,1:n); U0 = [U0; ones(1,n)];
else U0 = -1; end
Y0 = zeros(k,n);
v = randi([1 k],1,n);
for i = 1:n, Y0(v(i),i) = 1; end
```

Figure 11. Generation of a training set for testing, GenTrainingSet.m.

```
function [sr,time] = Exp(m,n,h,k,d,delta)
N1 = 10; N2 = 100; srsum = 0; t = 0;
for j = 1:N1
    rng(1000*j);
    [U0,Y0] = GenTrainingSet(m,n,k);

    tic; [V,W] = EvalVW(U0,Y0,h,d); t = t + toc;

    [~,I0] = max(Y0);
    for i = 1:N2
        rng(1000*j+i);
        U = U0;
        U = U + [2 * delta * (rand(m,n) - .5); zeros(1,n)];
        H = [1./(1 + exp(-V*U)); ones(1,n)];
        Y = W * H; [~,I] = max(Y);
        srsum = srsum + 100 * sum(I == I0)/n;
    end
end
time = t/N1;
sr = srsum/(N1*N2);
```

Figure 12. Inner loop experiments Exp.m, called by NNX.m, based on 10 training sets, each performing 100 input/output tests.

```
clear, clc
m = [10 20 40]';
p = [m 2.5*m 2.5*m-1 m; m 2.5*m m m]
diagReinfRel = 10^4*[10 10 10 1 1 1];
for r = 1:6
    time = 0;
    m = p(r,1); n = p(r,2); h = p(r,3); k = p(r,4);
    dr = diagReinfRel(r) * n;
    for j = 1:2
        for i = 1:6
            delta = .1*(i-1);
            if j == 1, d = 0; else d = dr * delta * delta; end
            [sr,t] = Exp(m,n,h,k,d,delta);
            successRate(i,j) = sr;
            time = time + t;
        end
    end
    successRate
    aveTime(r) = time/12;
end
1000 * aveTime %milliseconds
```

Figure 13. The MATLAB master script NNX.m, the outer loop for the validation of the function EvalVW($\mathbf{U}_0, \mathbf{Y}_0, h, d$) in Figure 10, and the experimental results in Figures 4-9 (backpropagation excluded).

```
p =

    10    25    24    10
    20    50    49    20
    40   100    99    40
    10    25    10    10
    20    50    20    20
    40   100    40    40

ans =

  100.0000   78.7560   58.7600   46.2760   38.2480   32.4600
  100.0000   83.9840   82.4880   79.7600   76.4720   72.9680

ans =

  100.0000   68.1140   43.2180   30.1940   22.9300   18.5740
  100.0000   90.1140   88.9560   87.2680   85.0620   82.2980

ans =

  100.0000   75.1150   51.6290   35.2760   24.9340   18.6990
  100.0000   98.2980   97.9940   97.4470   96.5440   95.1400

ans =

   83.2000   82.2400   80.3280   77.6960   74.3560   70.7360
   83.2000   84.4800   82.6160   79.7920   76.4280   72.5720

ans =

   91.4000   90.4880   89.0460   86.8360   83.8920   80.4020
   91.4000   90.0060   88.9680   87.2620   85.0540   81.0420

ans =

   98.5000   98.2320   97.8330   97.1770   96.0400   94.3460
   98.5000   98.3130   98.0030   97.4440   96.5250   95.0490

ans =

    0.3257    0.5587    1.4664    0.1749    0.3056    0.6307
```

Figure 14. Results from the execution of NNX.m by MATLAB (with some line breaks removed), corresponding to the code in Figure 13, Figures 4-9 (C++), and the verification column in Table VI.