“Parallel Training Considered Harmful?”:
Comparing Series-Parallel and Parallel Feedforward
Network Training

Antônio H. Ribeiro*
Graduate Program in Electrical Engineering at Universidade Federal de Minas Gerais (UFMG) - Av. Antônio Carlos 6627, 31270-901, Belo Horizonte, MG, Brazil

Luis A. Aguirre*
Department of Electronic Engineering at Universidade Federal de Minas Gerais (UFMG) - Av. Antônio Carlos 6627, 31270-901, Belo Horizonte, MG, Brazil

Abstract
Neural network models for dynamic systems can be trained either in parallel or in series-parallel configurations. Influenced by early arguments, several papers justify the choice of series-parallel rather than parallel configuration claiming it has a lower computational cost, better stability properties during training and provides more accurate results. Other published results, on the other hand, defend parallel training as being more robust and capable of yielding more accurate long-term predictions. The main contribution of this paper is to present a study comparing both methods under the same unified framework. We focus on three aspects: i) robustness of the estimation in the presence of noise; ii) computational cost; and, iii) convergence. A unifying mathematical framework and simulation studies show situations where each training method provides better validation results, being parallel training better in what is believed to be more realistic scenarios. An example using measured data seems to reinforce such claim. We also show, with a novel complexity analysis and numerical examples, that both methods have similar computational cost, being series series-parallel training, however, more amenable to parallelization. Some informal discussion about stability and convergence properties is presented and explored in the examples.

Keywords: Neural network, parallel training, series-parallel training, system identification, output error models

*This work has been supported by the Brazilian agencies CAPES, CNPq and FAPEMIG.
Email addresses: antonio-ribeiro@ufmg.br (Antônio H. Ribeiro), aguirre@ufmg.br (Luis A. Aguirre)
URL: antoniore92.github.io (Antônio H. Ribeiro)
1. Introduction

Neural networks are widely used and studied for modeling nonlinear dynamic systems [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. In the seminal paper by Narendra and Parthasarathy [1] a distinction is made between two training methods: series-parallel and parallel. In the series-parallel configuration, the weights are estimated minimizing the one-step-ahead prediction error, while, in the parallel configuration, the free-run simulation error is minimized.

In [1], the series-parallel configuration is said to be preferred to parallel configuration for three main reasons, latter mentioned in several other works: i) all signals generated in the identification procedure for series-parallel configuration are bounded, while this is not guaranteed for the parallel configuration [2, 3]; ii) for the parallel configuration a modified version of backpropagation is needed, resulting in a greater computational cost, while the standard backpropagation can be used for series-parallel configuration [4, 5, 6]; and, iii) assuming that the error tends asymptotically to small values the simulated output is not so different from the actual one and, therefore, the results obtained from two configurations would not be significantly different [11, 12, 13]. An additional reason that also appears sometimes in the literature is that: iv) the series-parallel training provides better results because of more accurate inputs to the neural network during training [5, 6, 7, 8, 9, 10].

Results presented in other papers, however, show the strengths of parallel training: according to [14, 15] neural network trained in parallel yield more accurate long-term predictions than the ones trained in series-parallel; [16] shows for diverse types of models, including neural networks, that the parallel training can be more robust than series-parallel training in the presence of some kinds of noise; and, in [17] neural network parallel models presents better validation results than series-parallel models for modeling a boiler unit. Furthermore, the free-run simulation error minimization (used in parallel training) seems to be successful when dealing with other types of models: some state-of-the-art structure selection techniques for polynomial models are based on it [18, 19, 20, 21, 22]; and, in [23] it provided the best results when estimating the parameters of a battery.

The main contribution of this paper is to compare these two training methods. We focus on three aspects: i) robustness of the estimation in the presence of noise; ii) computational cost; and, iii) convergence. Our findings suggest that parallel training may provide the most accurate models under some common circumstances. Furthermore, its computational cost is not significantly different than series-parallel training. We believe this to be relevant because it contradicts some of the frequently cited reasons for using series-parallel rather than parallel training.

The rest of the paper is organized as follows: Section 2 presents both training modes as prediction error methods [24] and Section 3 formulates neural network training in both configurations as a nonlinear least-squares problem. It is a secondary contribution of this paper to present both training methods under the same framework and the comparison in subsequent sections is built on top
of this formulation. Section 4 presents a complexity analysis comparing the methods and Section 5 discusses signal unboundedness and the possibility of convergence to “bad” local solutions. In Section 6, numerical examples with measured and simulated data investigate the effect of the noise in the estimation and the running time of both methods. Final comments and future work ideas are presented in Section 7.

2. Unifying Framework

Consider the following data set: \( Z = \{ (u[k], y[k]), k = 1, 2, \ldots, N \} \), containing a sequence of sampled inputs-output pairs. Here \( u[k] \in \mathbb{R}^{N_u} \) and \( y[k] \in \mathbb{R}^{N_y} \) are vectors containing all the inputs and outputs of interest at instant \( k \). The output \( y[k] \) is correlated with its own past values \( y[k-1], y[k-2], \ldots \), and with past input values \( u[k-\tau_d], u[k-\tau_d-1], u[k-\tau_d-2], \ldots \). The integer \( \tau_d \geq 0 \) is the input-output delay (when changes in the input take some time to affect the output).

This paper is focused on trying to find a difference equation model:

\[
y[k] = F(y[k-1], \ldots, y[k-n_y], u[k-\tau_d], \ldots, u[k-n_u]; \Theta),
\]

that best describes the data in \( Z \). The model is described by: the parameterized nonlinear function \( F \) (e.g. a neural network); a parameter vector \( \Theta \in \mathbb{R}^{N_\Theta} \); the maximum input and output lags \( n_y \) and \( n_u \); and, the input-output delay \( \tau_d \). It is implicit here the assumption that a finite number of past terms can be used to describe the output.

From now on, the simplified notation:

\[
y[k] = \begin{bmatrix} y[k-1] & \ldots & y[k-n_y] & u[k-\tau_d] & \ldots & u[k-n_u] \end{bmatrix}^T,
\]

will be used, hence, Equation (1) can be rewritten as:

\[
y[k] = F(y[k], u[k]; \Theta).
\]

In this section we present parallel and series-parallel training in the prediction error methods framework. Prediction error methods were initially developed for linear systems [24]. Although there are works that extend the results to nonlinear models [25, 26], the authors are not aware of a mathematical derivation in an entirely nonlinear setup, as the one presented here, being described anywhere else.

2.1. Output Error vs Equation Error

To study the previously described problem it is assumed that for a given input sequence \( u[k] \) and a set of initial conditions \( y_0 \), the output was generated by a “true system”, described by the following equations:

\[
y^*[k] = F^*(y^*[k-1], \ldots, y^*[k-n_y], u[k-\tau_d], \ldots, u[k-n_u]; \Theta^*) + v[k],
\]

\[
y[k] = y^*[k] + w[k],
\]

where \( F^* \) and \( \Theta^* \) are the “true” function and parameter vector that describe the system; \( v[k] \in \mathbb{R}^{N_y} \) and \( w[k] \in \mathbb{R}^{N_y} \) are random variable vectors, that cause
the deviation of the deterministic model from its true value; \( u[k] \) and \( y[k] \) are the measured input and output vectors; and, \( y^*[k] \) is the output vector without the effect of the output error.

The random variable \( v[k] \) affects the system dynamics and is called *equation error*, while the random variable \( w[k] \) only affects the measured values and is called *output error*.

### 2.2. Free-run Simulation and One-step-ahead Prediction

**Definition 1** *(One-step-ahead prediction)*. For a given function \( F \), parameter vector \( \Theta \) and dataset \( Z \), the one-step-ahead prediction is defined as:

\[
\hat{y}_1[k] = F(y[k-1], \ldots, y[k-n_y], u[k-\tau_d], \ldots, u[k-n_u]; \Theta),
\]

(3)

**Definition 2** *(Free-run simulation)*. For a given function \( F \), parameter vector \( \Theta \), dataset \( Z \), and a set of initial conditions \( \{y_0[k]\}_{k=1}^{n_y} \), the free-run simulation is defined using the recursive formula:

\[
\hat{y}_s[k] = \begin{cases} 
    y_0[k], & 1 \leq n < n_y; \\
    F(\hat{y}_s[k-1], \ldots, \hat{y}_s[k-n_y], u[k-\tau_d], \ldots, u[k-n_u]; \Theta), & n \geq n_y,
\end{cases}
\]

(4)

The vector of initial conditions is defined as \( y_0 = [y_0[1]^T, \ldots, y_0[n_y-1]^T]^T \).

### 2.3. Optimal Predictor

If the measured values of \( y \) and \( u \) are known at all instants previous to \( k \), the optimal prediction of \( y[k] \) is the following conditional expectation:  

\[
\hat{y}^*[k] = E\left\{y[k] \mid y[k], u[k] \right\}
\]

where \( \hat{y}^*[k] \) denotes the optimal prediction and \( E\{\cdot\} \) indicates the mathematical expectation.

Consider the following situations:

**Situation 1** *(White equation error)*. The sequence of equation errors \( \{v[k]\} \) is a white noise process and the output error is zero \((w[k] = 0)\).

**Situation 2** *(White output error)*. The sequence of output errors \( \{w[k]\} \) is a white noise process and the equation error is zero \((v[k] = 0)\).

The next two lemmas give the optimal prediction \( \hat{y}^*[k] \) for the two situations above:

**Lemma 1.** If Situation 1 holds and the function and parameter vector matches the true ones \((F = F^* \text{ and } \Theta = \Theta^*)\), then the one-step-ahead prediction is equal to the optimal prediction \((\hat{y}_1[k] = \hat{y}^*[k])\).

---

1The prediction is optimal in the sense that the expected squared prediction error is minimized \([27, p.18, \text{Sec. 2.4}]\).
Proof. Since the output error is zero, it follows that \( y[k] = y^*[k] \) and therefore Equation (2) reduces to \( y[k] = F^*(\bar{y}[k], u[k]; \Theta^*) + v[k] \).

And, because \( v[k] \) has zero mean\(^2\), it follows that:

\[
\hat{y}_s[k] = E\{y[k] \mid \bar{y}[k], u[k]\} = F^*(\bar{y}[k], u[k]; \Theta^*) = \hat{y}_1[k].
\]

Lemma 2. If Situation 2 holds, and the function, parameter vector and initial conditions match the true ones \((F = F^*, \Theta = \Theta^* \text{ and } y_0 = y_0^*)\), then the free-run simulation is the optimal prediction \((\hat{y}_s[k] = \hat{y}_s[k])\).

Proof. There is no equation error and therefore:

\[
\hat{y}_s[k] = E\{y[k] \mid \bar{y}[k], u[k]\} = E\{y^*[k] + w[k] \mid \bar{y}[k], u[k]\} = y^*[k] = \hat{y}_s[k],
\]

where it was used that for matching initial conditions and parameters and in the absence of equation error, the noise-free output \( y^*[k] \) is exactly equal to the free-run simulation \((y^*[k] = \hat{y}_s[k])\).

In series-parallel configuration the parameters are estimated by minimizing the one-step-ahead error, \( e_1[k] = \hat{y}_1[k] - y[k] \), while in parallel training the free-run simulation error, \( e_s[k] = \hat{y}_s[k] - y[k] \), is the one minimized. For \( F = F^* \), both training methods minimize an error that approaches the optimal predictor error \( e^*[k] = \hat{y}_s[k] - y[k] \) as \( \Theta \to \Theta^* \). Series-parallel training does it for Situation 1, and parallel training for Situation 2. Hence, both training procedures may be regarded as prediction error methods\(^3\).

Neural networks are an appropriate choice for the function \( F \) because they are universal approximators. A neural network with as few as one hidden layer can approximate any measurable function \( F^* \) with arbitrary accuracy within a given compact set \([28]\).

Let \( e = [e[1]^T, \ldots, e[N]^T]^T \) be the prediction error vectors. In this paper we focus in the minimization of the sum of square errors \( ||e||^2 \). This loss function is optimal in the maximum likelihood sense if the residuals are considered to be Gaussian white noise \([29]\). An algorithm for minimizing the square errors is described in the following section in the context of neural network models.

3. Nonlinear Least-Squares Network Training

Unlike other machine learning applications (e.g. natural language processing and computer vision) where there are enormous datasets available to train neural network models, the datasets available for system identification are usually of

\(^2\)A white noise process has zero mean by definition.

\(^3\)In the context of predictor error methods the nomenclature NARX (nonlinear autoregressive model with exogenous input) and NOE (nonlinear output error model) is often used to refer to the models obtained using, respectively, series-parallel and parallel training.
moderate size. The available data is usually obtained through tests with limited duration because of practical and economical reasons. And, even when there is a long record of input-output data, it either does not contain meaningful dynamic behavior [30] or the system cannot be considered time-invariant over the entire record, resulting in the necessity of selecting smaller portions of this longer dataset for training.

Due to the unavailability of large datasets, the use of neural networks in system identification is usually restricted to neural networks with few hundred weights. The Levenberg-Marquardt method does provide a fast convergence rate [31] and has been described as very efficient for batch training of moderate size problems [32]. Hence, it will be the method of choice for training neural networks in this paper.

This section presents the parallel and series-parallel training as nonlinear least-squares problems. Sections 3.1 and 3.2 give some background in the the Levenberg-Marquardt algorithm and in the backpropagation algorithm proposed by [32]. Series-parallel and parallel training are discussed in Sections 3.3 and 3.4. The backpropagation can be directly applied to series-parallel training, while for parallel training we introduce a new formula for computing the derivatives. This formula can be interpreted either as a variation of the dynamic backpropagation [1] adapted to compute the Jacobian instead of the gradient; or, as a specific case of real-time recurrent learning [33] with a special type of recurrent connection.

3.1. Nonlinear Least-Squares

Let $\Theta \in \mathbb{R}^{N_\Theta}$ be a vector of parameters containing all neural network weights and bias and $e(\Theta) \in \mathbb{R}^{N_e}$ an error vector. In order to estimate the parameter vector $\Theta$ the sum of square errors $V(\Theta) = \frac{1}{2} \| e(\Theta) \|^2$ is minimized. Its gradient vector and Hessian matrix may be computed as: [31, p. 246]

$$\frac{\partial V}{\partial \Theta} = \left[ \frac{\partial e}{\partial \Theta} \right]^T e(\Theta),$$

(6)

$$\frac{\partial^2 V}{\partial \Theta^2} = \left[ \frac{\partial e}{\partial \Theta} \right]^T \left[ \frac{\partial e}{\partial \Theta} \right] + \sum_{i=1}^{N_e} e_i \frac{\partial^2 e_i}{\partial \Theta^2},$$

(7)

where $\frac{\partial e}{\partial \Theta} \in \mathbb{R}^{N_e \times N_\Theta}$ is the Jacobian matrix associated with $e(\Theta)$. Non-linear least-squares algorithms usually update the solution iteratively ($\Theta^{n+1} = \Theta^n + \Delta \Theta^n$) and exploit the special structure of the gradient and Hessian of $V(\Theta)$, in order to compute the parameter update $\Delta \Theta^n$.

The Levenberg-Marquardt algorithm considers a parameter update [34]:

$$\Delta \Theta^n = - \left[ \frac{\partial e}{\partial \Theta} \right]_n^T \left[ \frac{\partial e}{\partial \Theta} \right]_n + \lambda^n D^n \right]^{-1} \left[ \frac{\partial e}{\partial \Theta} \right]_n^T e_n,$$

(8)

for which $\lambda^n$ is a non-negative scalar and $D^n$ is a non-negative diagonal matrix. Furthermore, $e_n$ and $\frac{\partial e}{\partial \Theta} \big|_n$ are the error and its Jacobian matrix evaluated at $\Theta^n$. 

6
There are different ways of updating $\lambda^n$ and $D^n$. The update strategy presented here is similar to [35]. The elements of the diagonal matrix $D^n$ are chosen equal to the elements in the diagonal of $\left[ \partial e_n / \partial e_n \right]$. And $\lambda^n$ is increased or decreased according to the agreement between the local model ($\phi_n(\Delta \Theta^n) = \frac{1}{2} \| e_n + \left[ \partial e_n / \partial \Theta \right] \Delta \Theta^n \|^2$) and the real objective function $V(\Theta_n)$. The degree of agreement is measured using the following ratio:

$$\rho_n = \frac{V(\Theta^n) - V(\Theta^n + \Delta \Theta^n)}{\phi_n(0) - \phi_n(\Delta \Theta^n)}.$$  \hspace{1cm} (9)

One iteration of the algorithm is summarized next:

**Algorithm 1 (Levenberg-Marquardt Iteration).** For a given $\Theta^n$ and $\lambda^n$:

1. Compute $e(\Theta^n)$ and $\partial e / \partial \Theta$ if not already computed.
2. $\text{diag}(D^n) \leftarrow \text{diag} \left( \left[ \partial e / \partial \Theta \right] \right)$.
3. Solve (8) and compute $\Delta \Theta^n$.
4. Compute $\rho_n$ as in (9).
5. $\lambda^{n+1} \leftarrow 4\lambda^n$ if $\rho_n > \frac{3}{4}$; $\lambda^{n+1} \leftarrow \frac{1}{2} \lambda^n$ if $\rho_n < \frac{1}{4}$; otherwise, $\lambda^{n+1} \leftarrow \lambda^n$.
6. $\Theta^{n+1} \leftarrow \Theta^n + \Delta \Theta^n$ if $\rho_n > 10^{-3}$; otherwise, $\Theta^{n+1} \leftarrow \Theta^n$.
7. $n = n + 1$.

### 3.2. Modified Backpropagation

Consider a multi-layer feedforward network, such as the three-layer network in Figure 1. This network can be seen as a function that relates the input $x \in \mathbb{R}^N$ to the output $z \in \mathbb{R}^N$. The parameter vector $\Theta$ contains all weights $w^{(n)}_{i,j}$ and bias $\gamma^{(n)}_i$ of the network. This subsection presents a modified version of backpropagation [32] for computing the neural network output $z$ and its Jacobian matrix for a given input $x$. The notation used is the one displayed in Figure 1.

#### 3.2.1. Forward Stage

For an $\mathcal{L}$ layer network the output nodes can be computed using the following recursive matrix relation:

$$\alpha^{(n)} = \begin{cases} x & n = 0; \\ h^{(n)}(W^{(n)}\alpha^{(n-1)} + \gamma^{(n)}) & n = 1, \ldots, \mathcal{L}, \end{cases}$$  \hspace{1cm} (10)

where, for the $n$-th layer, $W^{(n)}$ is a matrix containing the weights $w^{(n)}_{i,j}$, $\gamma^{(n)}$ is a vector containing the bias $\gamma^{(n)}_i$, and $h^{(n)}$ applies the nonlinear function $h^{(n)}$ element-wise. The output $z$ is given by:

$$z = \alpha^{(\mathcal{L})}.$$  \hspace{1cm} (11)
3.2.2. Backward Stage

The following recurrence relation can be used to compute \( \frac{\partial z}{\partial \beta(n)} \) for every \( n \):

\[
\frac{\partial z}{\partial \beta(n)} = \left\{ \begin{array}{ll}
\dot{H}(L) (\beta(L)) & n = L; \\
\dot{H}(n) (\beta(n)) \cdot W(n+1) \cdot \dot{H}(n+1) & n = L-1, \ldots, 1.
\end{array} \right.
\]

(12)

where \( \dot{H}(n) \) is given by the following diagonal matrix:

\[
\dot{H}(n) (\beta(n)) = \text{diag}(\dot{h}(n)(\beta(n)), \ldots, \dot{h}(n,N)_{sk})\).
\]

The recursive expression (12) follows from applying the chain rule \( \frac{\partial z}{\partial \beta(n)} = \frac{\partial z}{\partial \beta(n+1)} \frac{\partial \beta(n+1)}{\partial \alpha(n)} \frac{\partial \alpha(n)}{\partial \beta(n)} \), and considering \( \frac{\partial \beta(n+1)}{\partial \alpha(n)} = W(n+1) \) and \( \frac{\partial \alpha(n)}{\partial \beta(n)} = \dot{H}(n) \).

3.2.3. Computing Derivatives

The derivatives of \( z \) in relation to the weights \( w_{i,j}^{(n)} \) and the bias \( \gamma_i^{(n)} \) can be used to form the Jacobian matrix \( \frac{\partial z}{\partial \Theta} \) and can be computed using the following expressions:

\[
\frac{\partial z}{\partial w_{i,j}^{(n)}} = \frac{\partial z}{\partial \beta(n)} \frac{\partial \beta(n)}{\partial \alpha^{(n-1)}} \alpha^{(n-1)}; \\
\frac{\partial z}{\partial \gamma_i^{(n)}} = \frac{\partial z}{\partial \beta(n)} \frac{\partial \beta(n)}{\partial \gamma_i^{(n)}} \gamma_i^{(n)}. 
\]

(13)

(14)

Furthermore, the derivatives of \( z \) in relation to the inputs \( x \) are:

\[
\frac{\partial z}{\partial x} = \frac{\partial z}{\partial \beta(1)} \frac{\partial \beta(1)}{\partial \alpha(1)} \alpha(1) = \frac{\partial z}{\partial \beta(1)} W(1).
\]

(15)

3.3. Series-parallel Training

In the series-parallel configuration the parameters are estimated by minimizing \( \frac{1}{2} \| e_1 \|^2 \), what can be done using the algorithm described in Section 3.1. The required Jacobian matrix \( \frac{\partial z}{\partial \Theta} \) can be computed according to the following well known result.
Proposition 1. The Jacobian matrix of $e_1$ in relation to $\Theta$ is
\[
\frac{\partial e_1}{\partial \Theta} = \left[ \frac{\partial e_1[1]}{\partial \Theta}, \ldots, \frac{\partial e_1[N]}{\partial \Theta} \right]^T,
\]
where $\frac{\partial e_1[k]}{\partial \Theta} = \frac{\partial y[k]}{\partial \Theta} (\hat{y}[k], u[k], \Theta)$ that can be computed using the backpropagation described at Section 3.2.

Proof. Results from differentiating (3). \hfill \square

3.4. Parallel Training

In the parallel configuration the parameters are estimated by minimizing $\frac{1}{2} \| e_s \|^2$. There are two different ways to take into account the initial conditions $y_0$, they are: (i) to fix the initial conditions $y_0$ and estimate the model parameters $\Theta$; and, (ii) to define an extended parameter vector $\Phi = [\Theta^T, y_0^T]^T$ and estimate $y_0$ simultaneously with $\Theta$.

When using formulation (i), a suitable choice is to set the initial conditions equal to the measured outputs ($y_0[k] = y[k], k = 1, \ldots, n_y - 1$). When using formulation (ii) the measured outputs may be used as an initial guess to be refined by the optimization algorithm.

The optimal choice for the initial condition would be $y_0[k] = y^*[k]$ for $n = 1, \ldots, n_y - 1$. Formulation (i) uses the non-optimal choice $y_0[k] = y[k] \neq y^*[k]$. Formulation (ii) goes one step further and include the initial conditions $y_0[k]$ in the optimization problem, so it converges to $y^*[k]$ and hence improves the parameter estimation.

The Jacobian matrices $\frac{\partial e_s}{\partial \Theta}$ and $\frac{\partial e_s}{\partial y_0}$ can be computed according to the following proposition.

Proposition 2. The Jacobian matrices of $e_1$ in relation to $\Theta$ and $y_0$ are
\[
\frac{\partial e_s}{\partial \Theta} = \left[ \frac{\partial e_s[1]}{\partial \Theta}, \ldots, \frac{\partial e_s[N]}{\partial \Theta} \right]^T \text{ and } \frac{\partial e_s}{\partial y_0} = \left[ \frac{\partial e_s[1]}{\partial y_0}, \ldots, \frac{\partial e_s[N]}{\partial y_0} \right]^T
\]
and $\frac{\partial e_s[k]}{\partial y_0} = \frac{\partial y[k]}{\partial y_0}$ can be computed according to the following recursive formulas:

\[
\frac{\partial y[k]}{\partial \Theta} = \begin{cases} 
0, & 1 \leq k \leq n_y - 1; \\
\frac{\partial y}[{k}]}{\partial \Theta}(\hat{y}[k], u[k], \Theta) + \sum_{i=1}^{n_y} \frac{\partial y}[{k}]}{\partial y_{k-1}}(\hat{y}[k], u[k], \Theta) \frac{\partial y_{k-1]}{\partial \Theta}, & n \geq n_y,
\end{cases}
\]

(16)

\[
\frac{\partial y[k]}{\partial y_0} = \begin{cases} 
D^{(n)}, & 1 \leq k \leq n_y - 1; \\
\sum_{i=1}^{n_y} \frac{\partial y[k]}{\partial y_{k-1}}(\hat{y}[k], u[k], \Theta) \frac{\partial y_{k-1]}{\partial y_0}, & n \geq n_y,
\end{cases}
\]

(17)

where $D^{(n)} \in \mathbb{R}^{N_y \times n_y N_y}$ is defined as:

\[
\{ D^{(n)} \}_{i,j} = \begin{cases} 
1, & \text{if } j = (n-1) \cdot N_y + i, \\
0, & \text{otherwise.}
\end{cases}
\]

(18)

Proof. Results from differentiating (4) and applying the chain rule. \hfill \square
4. Complexity Analysis

In this section we present a novel complexity analysis comparing series-parallel training (SP), parallel training with fixed initial conditions (PΘ) and parallel training with extended parameter vector (PΦ). We show that the training methods have similar computational cost for the nonlinear least-squares formulation. The number of floating point operations (flops) is estimated based on [36, Table 1.1.2]. Low order terms, as usual, are neglected in the analysis.

4.1. Neural Network Output and its Partial Derivatives

The backpropagation algorithm described in Section 3.2 can be used for training both fully or partially connected networks. What would have to change is the internal representation of the weight matrices W(n): for a partially connected network the matrices would be stored using a sparse representation, e.g. compressed sparse column (CSC) representation.

The total number of flops required to evaluate the output and to compute its partial derivatives for the feedforward network is summarized in Table 1 considering a fully connected network. The total number of network weights is denoted as Nw and the total number of bias terms as Nγ, such that Nw + Nγ = NΦ. For this fully connected network:

\[
N_w = N_x \cdot N_{s1} + N_{s1} \cdot N_{s2} + \cdots + N_{s(L-1)} \cdot N_sL,
\]

\[
N_\gamma = N_{s1} + N_{s2} + \cdots + N_sL.
\]

Table 1: Modified backpropagation number of flops for a fully connected network.

| Computing Neural Network Output | Computing Partial Derivatives |
|--------------------------------|--------------------------------|
| i) Compute \( F(x; \Theta) \) — Eq. (10)-(11) | \( 2N_w \) |
| ii) Backward Stage — Eq. (12) | \( (2N_z + 1)(N_w - N_x N_{s1}) \) |
| iii) Compute \( \frac{\partial F}{\partial \Theta} \) — Eq. (13)-(14) | \( N_w \cdot N_z \) |
| iv) Compute \( \frac{\partial F}{\partial x} \) — Eq. (15) | \( 2N_z \cdot N_{s1} \cdot N_z \) |

Since the more relevant terms of the complexity analysis in Table 1 are being expressed in terms of \( N_w \) and \( N_z \) the results for a fully connected network are similar to the ones that would be obtained for a partially connected network using a sparse representation.

4.2. Number of Flops for Series-Parallel and Parallel Training

The number of flops of each iteration of the Levenberg-Marquardt algorithm is summarized in Table 2. Entries (i) to (iv) in Table 2 follow directly from Table 1, considering \( N_z = N_y \) and multiplying the costs by \( N \) because of
Table 2: Levenberg-Marquardt number of flops per iteration for series-parallel training (SP), parallel training with fixed initial conditions (PΘ) and parallel training with extended parameter vector (PΦ). A mark * signals which calculation is required in each method.

|                                      | SP | PΘ | PΦ |
|--------------------------------------|----|----|----|
| **Computing Error**                  |    |    |    |
| i) Compute $F(x; \Theta)$            | $2N \cdot N_w$ | * |    |
| **Computing Partial Derivatives**    |    |    |    |
| ii) Backward Stage                   | $N(2N_y + 1)(N_w - N_x N_{s1})$ | * | * | * |
| iii) Compute $\frac{\partial F}{\partial \Theta}$ | $N \cdot N_w \cdot N_y$ | * | * | * |
| iv) Compute $\frac{\partial F}{\partial x}$ | $2N \cdot N_x \cdot N_{s1} \cdot N_y$ | * | * | * |
| v) Equation (16)                     | $2N \cdot N_{\Theta} \cdot (N_y^2 + N_y)$ | * | * | * |
| vi) Equation (17)                    | $2N \cdot n_y \cdot N_y \cdot (N_y^2 + N_y)$ | * | * | * |
| **Solving Equation (8)**             |    |    |    |
| vii) Solve (8) — Θ                   | $2N \cdot N_{\Theta}^2 + \frac{1}{3} N_{\Theta}^3$ | * | * | * |
| viii) Solve (8) — Φ                  | $2N \cdot N_{\Phi}^2 + \frac{1}{3} N_{\Phi}^3$ | | * | |

the number of different inputs being evaluated. Furthermore, in entries (v) and (vi), it is considered that the evaluation of (16) and (17) is done, not by recomputing the entire summation each time, but by storing each computed values and computing only one new matrix-matrix product per evaluation.

The cost of solving (8) is about $2 \cdot N \cdot N_{\Theta}^2 + \frac{1}{3} \cdot N_{\Theta}^3$ where the cost $2N \cdot N_{\Theta}^2$ is due to the multiplication of the Jacobian matrix by its transpose and $\frac{1}{3} N_{\Theta}^3$ is due to the needed Cholesky factorization. For an extended parameter vector $N_{\Theta}$ is replaced by $N_{\Phi}$ in the analysis.

4.3. Comparing Methods

Assuming the number of nodes in the last hidden layer is greater than the number of outputs ($N_y < N_{s(L-1)}$), the inequalities follow directly from this paper definitions:

$$n_y \cdot N_y \leq N_x \leq N_x \cdot N_{s1} \leq N_w < N_{\Theta};$$

$$N_y < N_y^2 < N_y \cdot N_{s(L-1)} \leq N_w < N_{\Theta}.$$  \hspace{1cm} (20)

From Table 2 and from the above inequalities it follows that the cost of each Levenberg-Marquardt iteration is dominated by the cost of solving Equation (8). Furthermore, $N_{\Phi} = N_{\Theta} + n_y \cdot N_y < 2N_{\Theta}$, therefore, SΦ training method has the same asymptotic computational cost of SP and SΘ methods: $O(N \cdot N_{\Theta}^2 + N_{\Theta}^3)$.

From Table 2 it is also possible to analyze the cost of each of the major steps needed in each full iteration of the algorithm:

- **Computing Error**: The cost of computing the error is the same for all of the training methods.
• **Computing Partial Derivative:** The computation of partial derivatives has a cost of $O(N \cdot N_w \cdot N_y)$ for the SP training method and a cost $O(N \cdot N_\Theta \cdot N_y^2)$ for both $P\Phi$ and $P\Theta$. For many cases of interest in system identification, the number of model outputs $N_y$ is small. Furthermore, $N_\Theta = N_w + N_\gamma \approx N_w$ (see Eq. (19)). That is why the cost of computing the partial derivatives for parallel training is comparable to the cost for series-parallel training.

• **Solving Equation (8):** It already has been established that the cost of this step $O(N \cdot N_\Theta^2 + N_y^3)$ dominates the computational cost for all the training methods. Furthermore $n_y \cdot N_y$ is usually much smaller than $N_\Theta$ such that $N_\Phi \approx N_\Theta$ and the number of flops of this stage is basically the same for all the training methods.

### 4.4. Memory Complexity

Considering that $N$ is significantly larger than $n_y$, it follows that, for the three training methods, the storage capacity is dominated by the storage of the Jacobian matrix $\frac{\partial e}{\partial \Theta}$ or of the matrix resulting from the product $[\frac{\partial e}{\partial \Theta}]^T [\frac{\partial e}{\partial \Theta}]$. Therefore the size of the memory used by the algorithm is about $O(\max(N \cdot N_y \cdot N_\Theta, N_\Theta^2))$.

For very large datasets and a large number of parameters, this storage requirements may be prohibitive and others methods should be used (e.g. stochastic gradient descent or variations). Nevertheless, for datasets of moderate size and networks with few hundred parameters, as it is usually the case for system identification, the use of nonlinear least-squares is a viable option.

### 5. Practical Aspects

#### 5.1. Convergence towards a Local Minima

The optimization problem that results from both series-parallel and parallel training of neural networks are non-convex and may have multiple local minima. The solution of the Levenberg-Marquardt algorithm discussed in this paper, as well as most algorithms of interest for training neural networks (e.g. stochastic gradient descent, L-BFGS, conjugate gradient), converges towards a local minima\(^4\). However, there is no guarantee for neither series-parallel nor parallel training that solution found is the global optimum. The convergence to “bad” local solutions may happen for both training methods, however, as illustrated in the numerical examples, it seems to happen more often for parallel training.

\(^4\) It is proved in [37] that the Levenberg-Marquardt (not exactly the one discussed here) converges towards a local minima or a stationary point under simple assumptions.
5.2. Signal Unboundedness During Training

Signals obtained in intermediary steps of parallel identification may become unbounded. The one-step-ahead predictor, used in series-parallel training, is always bounded since the prediction depends only on measured values – it has a FIR (Finite Impulse Response) structure – while, for the parallel training, the free-run simulation could be unbounded for some choice of parameters because of its dependence on its own past simulation values.

This is a potential problem because during an intermediary stage of parallel training a choice of $\Theta^k$ that results in unbounded values of $\hat{y}_s[k]$ may need to be evaluated, resulting in overflows. Hence, optimization algorithms that may work well minimizing one-step-ahead prediction errors $e_1$, may fail when minimizing simulation errors $e_s$.

For instance, steepest descent algorithms with a fixed step size may, for a poor choice of step size, fall into a region in the parameter space for which the signals are unbounded and the computation of the gradient will suffer from overflow and prevent the algorithm from continuing (since it does not have a direction to follow). This may also happen in more practical line search algorithms (e.g. the one described in [31, Sec. 3.5]).

The Levenberg-Marquardt algorithm, on the other hand, is robust against this kind of problem because every step $\Theta^n + \Delta \Theta^n$ that causes overflow in the objective function computation yields a negative $\rho_n^5$, hence the step is rejected by the algorithm and $\lambda_n$ is increased. The increase in $\lambda_n$ causes the length of $\Delta \Theta^n$ to decrease$^6$. Therefore, the step length is decreased until a point is find close enough to the current iteration such that overflow does not occur. Hence, the Levenberg-Marquardt algorithm does not fail or stall due to overflows. Similar reasoning could be used for any trust-region method or for backtracking line-search.

Regardless of the optimization algorithm, signal unboundedness is not a problem for feedforward networks with bounded activation functions (e.g. Logistic or Hyperbolic Tangent) in the hidden layers, because its output is always bounded. Hence parallel training of these particular neural networks is not affected by the previously mentioned difficulties.

6. Implementation and Test Results

The implementation is in Julia and runs on a computer with a processor Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz. For all examples in this paper, the activation function used in hidden layers is the hyperbolic tangent, the weights $w_{ij}^{(n)}$ initial values are drawn from a normal distribution with standard deviation $\sigma = (N_{s(n)})^{-0.5}$ and the bias terms $\gamma_i^{(n)}$ are initialized with zeros [38].

---

5 Programming languages as Matlab, C, C++ and Julia returns the floating point value encoded for infinity when an overflow occur. In this case formula (9) yields a negative $\rho_n$.

6 This inverse relation between $\lambda_n$ and $\|\Delta \Theta^n\|$ is explained in [31].
Furthermore, in all parallel training examples we include the initial conditions as parameters of the optimization process (P→ training).

The free run simulation mean square error (MSE = \( \sum_{n} (y[k] - \hat{y}[k])^2 \)) is used as a goodness of fit measure to compare the models in the validation window.

The first example compares the training method using data from a real process and the second one investigates different noise configurations on computer generated data. The code and data used in the numerical examples are available in the GitHub repository: https://github.com/antonior92/ParallelTrainingNN.jl.

6.1. Example 1: Data from a Pilot Plant

In this example, input-output signals were collected from LabVolt Level Process Station (model 3503-MO [39]). This plant consists of a tank that is filled with water from a reservoir. The water is pumped at fixed speed, while the flow is regulated using a pneumatically operated control valve driven by a voltage \( u \). The water column height \( y \) is indirectly measured using a pressure sensor at the bottom of the tank. Figure 2 shows the free-run simulation in the validation window of models obtained for this process using parallel and series-parallel training.

![Figure 2: Displays free-run simulation in the validation window for models obtained using series-parallel (SP) and parallel (P) training. The mean square errors are MSE\(_{SP} = 1144.6\); MSE\(_{P} = 296.2\). The models have \( n_y = n_u = 1 \) and 10 nodes in the hidden layer and were trained on a two hour long dataset sampled at \( T_s = 10s \). The same initial parameter guess was used for both training methods. The training was 100 epochs long, which took 3.3 and 3.9 seconds, respectively, for series-parallel and parallel training.](image)

Since the parameters initial guess are randomly initialized, different realizations will yield different results. Figure 3 shows the validation errors for both training methods for randomly drawn initial guesses. While parallel training consistently provides models with better validation results than series-parallel training, it also has some outliers with very bad validation results. We interpret these outliers in the boxplot as consequence of the parallel training getting trapped in “bad” local minima, as mentioned in Section 5.
Figure 3: Boxplots show the distribution of the free-run simulation MSE over the validation window for models trained using series-parallel (SP) and parallel (P) methods under the circumstances specified in Figure 2. There are 100 realizations of the training in each boxplot, in each realization the weights $w_{i,j}^{(n)}$ are draw from a normal distribution with standard deviation $\sigma = (N_s(n-1))^{-0.5}$ and the bias terms $\gamma_i^{(n)}$ are initialized with zeros [38].

The training of the neural network was performed using normalized data. However, if unscaled data were used instead, parallel training would yield models with MSE $> 10000$ over the validation window while series-parallel training can still yield solutions with a reasonable fit to the validation data. We understand this as another indicator of parallel training greater sensitivity to the initial parameter guess: for unscaled data, the initial guess is far away from meaningful solutions of the problem, and, while series-parallel training converges to acceptable results, parallel training gets trapped in “bad” local solutions.

6.2. Example 2: Investigating the Noise Effect

The non-linear system: [40]

$$y^*[k] = (0.8 - 0.5e^{-y^*[k-1]^2})y^*[k-1] -$$
$$ (0.3 + 0.9e^{-y^*[k-1]^2})y^*[k-2] + u[k-1] +$$
$$ 0.2u[k-2] + 0.1u[k-1]u[k-2] + v[k]$$

$$y[k] = y^*[k] + w[k],$$ (21)

was simulated and the generated dataset was used to build neural network models. Figure 4 shows the validation results for models obtained for a training set generated with white Gaussian equation and output errors $v$ and $w$. In this section, we repeat this same experiment for diverse random processes applied to $v$ and $w$ in order to investigate how different noise configurations affect parallel and series-parallel training.

6.2.1. White Noise

Let $v$ be white Gaussian noise with standard deviation $\sigma_v$ and let $w$ be zero. Figure 5 (a) shows the free-run simulation error on the validation window using
Figure 4: Displays the first 100 samples of the free-run simulation in the validation window for models trained using series-parallel (SP) and parallel (P) methods. The mean square errors are $\text{MSE}_{\text{SP}} = 0.39$; $\text{MSE}_{\text{P}} = 0.06$. The models have $n_y = n_u = 2$ and a single hidden layer with 10 nodes. The training set has $N = 1000$ samples and was generated with (21) for $v$ and $w$ white Gaussian noise with standard deviations $\sigma_v = 0.1$ and $\sigma_w = 0.5$. The validation window is generated without the noise effect. For both, the input $u$ is randomly generated with standard Gaussian distribution, each randomly generated value held for 5 samples. The training was 100 epochs long, which took 5.0 and 6.1 seconds for, respectively, series-parallel and parallel training.

parallel or series-parallel training for increasing values of $\sigma_v$. Figure 5 (b) shows the complementary experiment, for which $v$ is zero and $w$ is white Gaussian noise with increasing larger values of $\sigma_w$ being tried out.

In Section 2, series-parallel training was derived considering only the presence of white equation error and, in this situation, the numerical results illustrate the model using this training method presents the best validation results (Figure 5 (a)). On the other hand, parallel training was derived considering only the presence of white output error and is significantly better in this alternative situation (Figure 5 (b)).
6.2.2. Colored Noise

Consider \( w = 0 \) and \( v \) a white Gaussian noise filtered by a low pass filter with cutoff frequency \( \omega_c \). Figure 6 shows the free-run simulation error in the validation window for both parallel and series-parallel training for a sequence of values of \( \omega_c \) and different noise intensities. The result indicates parallel training provides the best results unless the equation error has a very large bandwidth.

More extensive tests are summarized in Table 3, which shows the validation errors for a training set with colored Gaussian errors in different frequency bands. Again, except for white or large bandwidth equation error, parallel training seems to provide the smallest validation errors.

Equation error can be interpreted as the effect of unaccounted inputs and unmodeled dynamics, hence situations where this error is not auto-correlated are very unlikely. Therefore, the only situations we found series-parallel training to perform better (when the equation error power spectral density occupy almost the whole frequency spectrum) seem unlikely to appear in practice. This may justify parallel training to produce better models for real application problems as the pilot plant in Example 1, the battery modeling described in [23], or the boiler unit in [17].

6.3. Timings

In Section 4 we find out the computational complexity of \( O(N \cdot N_\theta^2 + N_\theta^3) \). The first term \( O(N \cdot N_\theta^2) \) seems to dominate and in Figure 7 we show that the running time grows linearly with the number of training samples \( N \) and quadratically with the number of parameters \( N_\theta \).

The running time growing with the same rate for both training methods implies that the ratio between series-parallel and parallel training running time is bounded by a constant. For the examples we presented in this paper the
parallel training takes about 20% more than series-parallel training. Hence the difference of running times for sequential execution does not justify the use of one method over the other. Parallel training is, however, much less amenable to parallelization because of the dependencies introduced by the recursive relations used for computing its error and Jacobian matrix. We propose a possible solution to this problem in the final remarks.

7. Final Remarks

In this paper we have studied different aspects of parallel training under a nonlinear least squares formulation. Due to the results presented in the numerical examples we have reasons to believe parallel training can provide models with smaller generalization error than series-parallel training under more realistic noise assumptions. Furthermore, for sequential execution both the com-
In both situations, the rows where the frequency ranges from 0.0 to 1.0 (the whole spectrum) corresponds to white noise, in the remaining rows we apply a 4th-order lowpass (or highpass) Butterworth filter to white Gaussian noise (in both the forward and reverse directions) in order to obtain the signal in the desired frequency band. The cell of the training method with the best validation results between the two models is colored. Its colored red when the difference in the MSE is larger than the sum of standard deviations and yellow when it is not.

| ω range | (a) $v \neq 0, w = 0$ | (b) $w \neq 0, v = 0$ |
|---------|-----------------------|-----------------------|
|         | SP                    | P                     | SP                    | P                     |
| 0.0 → 1.0 | 0.36 ± 0.09           | 0.78 ± 0.18           | 0.58 ± 0.03           | 0.13 ± 0.03           |
| 0.0 → 0.8 | 0.53 ± 0.13           | 0.58 ± 0.10           | 0.44 ± 0.02           | 0.12 ± 0.03           |
| 0.0 → 0.6 | 0.94 ± 0.13           | 0.75 ± 0.23           | 0.30 ± 0.05           | 0.15 ± 0.05           |
| 0.0 → 0.4 | 1.86 ± 0.20           | 1.07 ± 0.38           | 0.46 ± 0.05           | 0.15 ± 0.02           |
| 0.0 → 0.2 | 2.60 ± 0.26           | 1.16 ± 0.44           | 0.71 ± 0.05           | 0.18 ± 0.04           |
| 0.0 → 1.0 | 0.36 ± 0.09           | 0.78 ± 0.18           | 0.58 ± 0.03           | 0.13 ± 0.03           |
| 0.2 → 1.0 | 0.52 ± 0.07           | 0.57 ± 0.12           | 0.59 ± 0.03           | 0.09 ± 0.02           |
| 0.4 → 1.0 | 0.57 ± 0.08           | 0.22 ± 0.05           | 0.63 ± 0.05           | 0.05 ± 0.01           |
| 0.6 → 1.0 | 0.54 ± 0.07           | 0.21 ± 0.01           | 0.68 ± 0.03           | 0.03 ± 0.02           |
| 0.8 → 1.0 | 0.58 ± 0.05           | 0.17 ± 0.04           | 0.78 ± 0.09           | 0.03 ± 0.01           |

Several published works take for granted that series-parallel training always provide better results with lower computational cost [1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]. The results presented in this paper show this is not always the case and that parallel training does provide advantages that justify its use for training neural networks in several situations (see the numerical examples).

Series-parallel training, however, has two advantages over parallel training: i) it seems less likely to be trapped in “bad” local solutions; ii) it is more amenable to parallelization. For the examples presented in this paper the possibility of being trapped in “bad” local solutions is only a small inconvenience, requiring the data to be carefully normalized and, in some rare situations, the model to be retrained. We believe this to be the case in many situations. An exception are chaotic systems for which small variations in the parameters may cause great variations in the free-run simulation trajectory, causing the parallel training objective function to be very intricate and full of undesirable local minima. In [41] a technique called multiple shooting is introduced in the prediction error methods framework as a way of reducing the possibility of parallel training.

Complexity analysis and the numerical examples suggest the computational cost is not significantly different for both methods. Some other reasons mentioned in the literature to avoid parallel training, as the possibility of signal unboundedness [1, 2, 3], are also easy to circumvent (see Section 5).

In [41] a technique called multiple shooting is introduced in the prediction error methods framework as a way of reducing the possibility of parallel training...
Figure 7: Running time (in seconds) of training (100 epochs). The average running time of 5 realizations is displayed for series-parallel training (●) and for parallel training (○). The neural network has a single hidden layer and a total of $N_{\Theta}$ parameters to be estimated. The training set has $N$ samples and was generated as described in Figure 4. In (a), we fix $N_{\Theta} = 61$ and plot the timings as a function of the number of training samples $N$, a line is adjusted to illustrate the running time grows linearly with the training size for both training methods. In (b), we fix $N = 10000$ and plot the timings as a function of the number of parameters $N_{\Theta}$, a second order polynomial is adjusted to illustrate the running time quadratic growth.

getting trapped in “bad” local minima. Multiple shooting can also make the algorithm much more amenable to parallelization and seems to be a promising way to solve the shortcomings of parallel training.

Acknowledgments

This work has been supported by the Brazilian agencies CAPES, CNPq and FAPEMIG.

References

[1] K. S. Narendra, K. Parthasarathy, Identification and control of dynamical systems using neural networks, IEEE Transactions on Neural Networks 1 (1) (1990) 4–27.

[2] D.-y. Zhang, L.-p. Sun, J. Cao, Modeling of temperature-humidity for wood drying based on time-delay neural network, Journal of Forestry Research 17 (2) (2006) 141–144.

[3] M. Singh, I. Singh, A. Verma, Identification on non linear series-parallel model using neural network, MIT Int. J. Electr. Instrumen. Eng 3 (1) (2013) 21–23.
[4] M. Saad, P. Bigras, L.-A. Dessaint, K. Al-Haddad, Adaptive robot control using neural networks, IEEE Transactions on Industrial Electronics 41 (2) (1994) 173–181.

[5] M. H. Beale, M. T. Hagan, H. B. Demuth, Neural network toolbox for use with MATLAB, Tech. Rep., Mathworks, 2017.

[6] M. Saggar, T. Meričli, S. Andoni, R. Mliikkulainen, System identification for the Hodgkin-Huxley model using artificial neural networks, in: Neural Networks, 2007. IJCNN 2007. International Joint Conference on, IEEE, 2239–2244, 2007.

[7] E. Petrović, Ž. Ćojbašić, D. Ristić-Durrant, V. Nikolić, I. Ćirić, S. Matić, Kalman filter and NARX neural network for robot vision based human tracking, Facta Universitatis, Series: Automatic Control And Robotics 12 (1) (2013) 43–51.

[8] I. B. Tijani, R. Akmeliawati, A. Legowo, A. Budiwono, Nonlinear identification of a small scale unmanned helicopter using optimized NARX network with multiobjective differential evolution, Engineering Applications of Artificial Intelligence 33 (2014) 99–115.

[9] E. A. Khan, M. A. Elgamal, S. M. Shaarawy, Forecasting the Number of Muslim Pilgrims Using NARX Neural Networks with a Comparison Study with Other Modern Methods, British Journal of Mathematics & Computer Science 6 (5) (2015) 394.

[10] E. Diaconescu, The use of NARX neural networks to predict chaotic time series, WSEAS Transactions on Computer Research 3 (3) (2008) 182–191.

[11] K. Warwick, R. Craddock, An introduction to radial basis functions for system identification. A comparison with other neural network methods, in: Decision and Control, 1996., Proceedings of the 35th IEEE Conference on, vol. 1, IEEE, 464–469, 1996.

[12] W. Kamiński, P. Strumitto, E. Tomczak, Genetic algorithms and artificial neural networks for description of thermal deterioration processes, Drying Technology 14 (9) (1996) 2117–2133.

[13] M. F. Rahman, R. Devanathan, Z. Kuanyi, Neural network approach for linearizing control of nonlinear process plants, IEEE Transactions on Industrial Electronics 47 (2) (2000) 470–477.

[14] H. T. Su, T. J. McAvoy, P. Werbos, Long-term predictions of chemical processes using recurrent neural networks: A parallel training approach, Industrial & Engineering Chemistry Research 31 (5) (1992) 1338–1352.

[15] H.-T. Su, T. J. McAvoy, Neural model predictive control of nonlinear chemical processes, in: Intelligent Control, 1993., Proceedings of the 1993 IEEE International Symposium on, IEEE, 358–363, 1993.
[16] L. A. Aguirre, B. H. Barbosa, A. P. Braga, Prediction and simulation errors in parameter estimation for nonlinear systems, Mechanical Systems and Signal Processing 24 (8) (2010) 2855–2867.

[17] K. Patan, J. Korbicz, Nonlinear model predictive control of a boiler unit: A fault tolerant control study, International Journal of Applied Mathematics and Computer Science 22 (1) (2012) 225–237.

[18] L. Piroddi, W. Spinelli, An identification algorithm for polynomial NARX models based on simulation error minimization, International Journal of Control 76 (17) (2003) 1767–1781.

[19] M. Farina, L. Piroddi, Some convergence properties of multi-step prediction error identification criteria, in: Decision and Control, 2008. CDC 2008. 47th IEEE Conference on, IEEE, 756–761, 2008.

[20] M. Farina, L. Piroddi, An iterative algorithm for simulation error based identification of polynomial input–output models using multi-step prediction, International Journal of Control 83 (7) (2010) 1442–1456.

[21] M. Farina, L. Piroddi, Simulation error minimization identification based on multi-stage prediction, International Journal of Adaptive Control and Signal Processing 25 (5) (2011) 389–406.

[22] M. Farina, L. Piroddi, Identification of polynomial input/output recursive models with simulation error minimisation methods, International Journal of Systems Science 43 (2) (2012) 319–333.

[23] C. Zhang, K. Li, Z. Yang, L. Pei, C. Zhu, A new battery modelling method based on simulation error minimization, in: 2014 IEEE PES General Meeting— Conference & Exposition, 2014.

[24] L. Ljung, System Identification, Springer, 1998.

[25] J. Sjöberg, Q. Zhang, L. Ljung, A. Benveniste, B. Delyon, P.-Y. Glorennec, H. Hjalmarsson, A. Juditsky, Nonlinear black-box modeling in system identification: a unified overview, Automatica 31 (12) (1995) 1691–1724.

[26] P. M. Nørgård, O. Ravn, N. K. Poulsen, L. K. Hansen, Neural Networks for Modelling and Control of Dynamic Systems-A Practitioner’s Handbook, Springer-London, 2000.

[27] T. Hastie, R. Tibshirani, J. Friedman, The elements of statistical learning, vol. 1, Springer Series in Statistics, second edn., 2008.

[28] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural networks 2 (5) (1989) 359–366.

[29] O. Nelles, Nonlinear System Identification: from classical approaches to neural networks and fuzzy models, Springer Science & Business Media, 2001.
[30] A. H. Ribeiro, L. A. Aguirre, Selecting Transients Automatically for the Identification of Models for an Oil Well, IFAC-PapersOnLine 48 (6) (2015) 154–158.

[31] J. Nocedal, S. Wright, Numerical Optimization, Springer Science & Business Media, second edn., 2006.

[32] M. T. Hagan, M. B. Menhaj, Training feedforward networks with the Marquardt algorithm, IEEE Transactions on Neural Networks 5 (6) (1994) 989–993.

[33] R. J. Williams, D. Zipser, Experimental Analysis of the Real-Time Recurrent Learning Algorithm, Connection Science 1 (1) (1989) 87–111.

[34] D. W. Marquardt, An algorithm for least-squares estimation of nonlinear parameters, Journal of the Society for Industrial and Applied Mathematics 11 (2) (1963) 431–441.

[35] R. Fletcher, A modified Marquardt subroutine for non-linear least squares, 1971.

[36] G. H. Golub, C. F. Van Loan, Matrix Computations, vol. 3, JHU Press, 2012.

[37] J. J. Moré, The Levenberg-Marquardt algorithm: implementation and theory, in: Numerical analysis, Springer, 105–116, 1978.

[38] Y. A. LeCun, L. Bottou, G. B. Orr, K.-R. Müller, Efficient backprop, in: Neural Networks: Tricks of the trade, Springer, 9–48, 2012.

[39] LabVolt, Mobile Instrumentation and Process Control Training Systems, Tech. Rep., Festo, 2015.

[40] S. Chen, S. A. Billings, P. M. Grant, Non-linear system identification using neural networks, International Journal of Control 51 (6) (1990) 1191–1214.

[41] A. H. Ribeiro, L. A. Aguirre, Shooting Methods for Parameter Estimation of Output Error Models, IFAC-PapersOnLine 50 (1) (2017) 13998 – 14003, ISSN 2405-8963.