Automatica 156 (2023) 111183

Contents lists available at ScienceDirect

Automatica

journal homepage: www.elsevier.com/locate/automatica

Brief paper

Training recurrent neural networks by sequential least squares and the alternating direction method of multipliers☆

Alberto Bemporad

IMT School for Advanced Studies Lucca, Piazza San Francesco 19, Lucca, Italy

ARTICLE INFO

Article history:
Received 31 December 2021
Received in revised form 26 January 2023
Accepted 29 May 2023
Available online xxxx

Keywords:
Recurrent neural networks
Nonlinear system identification
Nonlinear least-squares
Generalized Gauss–Newton methods
Levenberg–Marquardt algorithm
Alternating direction method of multipliers
Non-smooth loss functions

ABSTRACT

This paper proposes a novel algorithm for training recurrent neural network models of nonlinear dynamical systems from an input/output training dataset. Arbitrary convex and twice-differentiable loss functions and regularization terms are handled by sequential least squares and either a line-search (LS) or a trust-region method of Levenberg–Marquardt (LM) type for ensuring convergence. In addition, to handle non-smooth regularization terms such as ℓ_1, ℓ_0, and group-Lasso regularizers, as well as to impose possibly non-convex constraints such as integer and mixed-integer constraints, we combine sequential least squares with the alternating direction method of multipliers (ADMM). We call the resulting algorithm NAILS (nonconvex ADMM iterations and least squares) in the case line search (LS) is used, or NAILM if a trust-region method (LM) is employed instead. The training method, which is also applicable to feedforward neural networks as a special case, is tested in three nonlinear system identification problems.

1. Introduction

Neural networks (NNs) have recently gained attention in several fields, although their use for modeling dynamical systems was already popular in the 1990s (Narendra & Parthasarathy, 1990). While *feedforward* NNs can model the output function of a dynamical system in autoregressive form with exogenous inputs (NNARX), *recurrent* neural networks (RNNs) often better capture the dynamics of the system due to their state-space form.

The RNN training problem is usually solved by gradient descent methods in which backpropagation through time (or its approximate truncated version (Williams & Peng, 1990)) is used to evaluate derivatives. To enable minibatch stochastic gradient descent (SGD) methods, Beintema, Toth, and Schoukens (2021) proposed to split the experiment into multiple sections where the initial state of each section is written as a NN function of a finite set of past inputs and outputs. Gradient-descent methods, however, suffer from a slow convergence rate. To improve computation efficiency, quality of the trained model, and to be able to train RNN models in real-time as new input/output data become available, training methods based on extended Kalman filtering (EKF) (Puskorius & Feldkamp, 1994) or recursive least squares (RLS) (Xu, Krishnamurthy, McMillin, & Lu, 1994) were proposed based on minimizing the mean-squared error (MSE) loss. These results were recently extended in Bemporad (2023) to handle general convex and twice-differentiable loss terms when training RNNs whose state-update and output functions are multi-layer feedforward NNs.

This paper proposes an offline training method based on two main contributions. First, to handle arbitrary convex and twice-differentiable loss terms, we propose using sequential least squares constructed by linearizing the RNN dynamics successively and taking a quadratic approximation of the loss to minimize. An advantage of this approach, compared to more classical backpropagation (Werbos, 1990), is that the required Jacobian matrices, whose evaluation is usually the most expensive part of the training algorithm, can be computed in parallel to propagate the gradients. To force the objective function to decrease monotonically, we consider two alternative ways: a line-search (LS) method and a trust-region method, the latter in the classical Levenberg–Marquardt (LM) setting (Levenberg, 1944; Marquardt, 1963; Transtrum & Sethna, 2012).

A second contribution of this paper is to combine sequential least squares with nonconvex alternating direction method of multipliers (ADMM) iterations to handle non-smooth and possibly non-convex regularization terms. ADMM has been proposed for training NNs, mainly feedforward NNs (Taylor et al., 2016) but also recurrent ones (Tang et al., 2020). However, existing methods entirely rely on ADMM to solve the learning problem, such as to gain efficiency by parallelizing numerical operations.

☆ This paper was partially supported by the Italian Ministry of University and Research under the PRIN'17 project "Data-driven learning of constrained control systems", contract no. 2017J89ARP. The material in this paper was not presented at any conference. This paper was recommended for publication in revised form by Associate Editor Tianshi Chen under the direction of Editor Alessandro Chiuso.

E-mail address: alberto.bemporad@imtlucca.it.

https://doi.org/10.1016/j.automatica.2023.111183
0005-1098/

In this paper, we combine ADMM with the smooth nonlinear optimizer mentioned above based on sequential LS to handle the non-smooth terms, therefore allowing handling sparsification (via ℓ_1, ℓ_0, or group-Lasso penalties) and imposing (mixed-)integer constraints, such as for quantizing the network coefficients. In particular, we show that we can use ADMM and group Lasso to select the number of states to include in the nonlinear RNN dynamics, therefore allowing a tradeoff between model complexity and quality of fit, a fundamental aspect for identifying control-oriented black-box models. See also (Li et al., 2019) for using nonconvex ADMM to handle combinatorial constraints in combination with RNN training.

We call the overall algorithm NAILS (nonconvex ADMM iterations and least squares) when line search (LS) is used, or NAILM if a trust-region approach of Levenberg–Marquardt (LM) type is employed instead. We show that NAILS/NAILM is computationally more efficient and provides better-quality solutions than classical gradient-descent methods, and that is competitive with respect to EKF-based approaches (Bemporad, 2023) and general-purpose non-smooth nonlinear programming (NLP) algorithms (Burke, Curtis, Lewis, Overton, & Simões, 2020; Curtis, Mitchell, & Overton, 2017).

The paper is organized as follows. After formulating the training problem in Section 2, we present the sequential least-squares approach to solving the smooth version of the problem in Section 3 and extend the method to handle non-smooth/non-convex regularization terms by ADMM iterations in Section 4. The performance and versatility of the proposed approach are shown in three nonlinear identification problems in Section 5.

2. Problem formulation

Given a set of input/output training data (u_0, y_0), …, (u_{N-1}, y_{N-1}), $u_k \in \mathbb{R}^{n_u}$, $y_k \in \mathbb{R}^{n_y}$, we want to identify a nonlinear state-space model in recurrent neural network (RNN) form

$$x_{k+1} = f_x(x_k, u_k, \theta_x), \quad \hat{y}_k = f_y(x_k, u_k, \theta_y) \tag{1}$$

where k denotes the sample instant, $x_k \in \mathbb{R}^{n_x}$ is the vector of hidden states, $f_x : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$ and $f_y : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_y}$ are classical multi-layer feedforward neural networks parameterized by weight/bias terms $\theta_x \in \mathbb{R}^{n_{\theta x}}$ and $\theta_y \in \mathbb{R}^{n_{\theta y}}$, respectively. In particular, f_x is a feedforward NN with linear output function and $L_x - 1$ layers parameterized by weight/bias terms $\{A_i^x, b_i^x\}$ (whose components are collected in θ_x), $i = 1, \ldots, L_x$, and activation functions f_i^x, $i = 1, \ldots, L_x - 1$, and similarly f_y by weight/bias terms $\{A_i^y, b_i^y\}$, $i = 1, \ldots, L_y$ (forming θ_y) and activation functions f_i^y, $i = 1, \ldots, L_y - 1$, followed by a possibly nonlinear output function $f_{L_y}^y$ (Bemporad, 2022, 2023). Let us denote by $n_1^x, \ldots, n_{L_x-1}$ and $n_1^y, \ldots, n_{L_y-1}$ the number of neurons in the hidden layers of f_x and f_y, respectively. Note that the RNN (1) can be made strictly causal by simply omitting u_k in f_y, or, equivalently, forcing the last n_u columns of A_1^y to be zero.

The training problem we want to solve is:

$$\min_z \; r(x_0, \theta_x, \theta_y) + g(\theta_x, \theta_y) + \sum_{k=0}^{N-1} \ell(y_k, f_y(x_k, u_k, \theta_y)) \tag{2a}$$

$$\text{s.t. } x_{k+1} = f_x(x_k, u_k, \theta_x), \;\; k = 0, \ldots, N-2 \tag{2b}$$

where $z = [x_0' \; \ldots \; x_{N-1}' \; \theta_x' \; \theta_y']' \in \mathbb{R}^n$ is the overall optimization vector, $n = Nn_x + n_{\theta_x} + n_{\theta_y}$, $\ell : \mathbb{R}^{n_y} \times \mathbb{R}^{n_y} \to \mathbb{R}$ is a loss function that we assume strongly convex and twice differentiable with respect to its second argument, $r : \mathbb{R}^{n_x} \times \mathbb{R}^{n_{\theta_x}} \times \mathbb{R}^{n_{\theta_y}} \to \mathbb{R}$ is a regularization term that we also assume strongly convex and twice differentiable (for example, an ℓ_2-regularizer), and $g : \mathbb{R}^{n_{\theta_x}} \times \mathbb{R}^{n_{\theta_x}} \to \mathbb{R} \cup \{+\infty\}$ is a possibly non-smooth and non-convex regularization term. For simplicity, in the following we assume that r is separable with respect to x_0, θ_x, and θ_y, i.e., $r(x_0, \theta_x, \theta_y) = r_x(x_0) + r_\theta^x(\theta_x) + r_\theta^y(\theta_y)$, and that in case of multiple outputs $n_y > 1$, the loss function ℓ is also separable, i.e., $\ell(y, \hat{y}) = \sum_{i=1}^{n_y} \ell_i(y_{ki}, f_{yi}(x_k, u_k, \theta_y))$, where the subscript i denotes the ith component of the output signal y_k and the corresponding loss function.

Problem (2) can be rewritten as the following unconstrained NLP problem in *condensed form*

$$\min_{x_0, \theta_x, \theta_y} V(x_0, \theta_x, \theta_y) + g(\theta_x, \theta_y) \tag{3}$$

where, by replacing $x_{k+1} = f_x(x_k, u_k, \theta_x)$ iteratively, we have defined

$$V(x_0, \theta_x, \theta_y) = r(x_0, \theta_x, \theta_y) + \sum_{k=0}^{N-1} \ell(y_k, f_y(x_k, u_k, \theta_y)) \tag{4}$$

Note that Problem (3) can be highly nonconvex and present several local minima, see, e.g., recent studies reported in Ribeiro, Tiels, Umenberger, Schön, and Aguirre (2020).

2.1. Multiple training traces and initial-state encoder

For simplicity of notation, in this paper, we focus on training a RNN model (1) based on a single input/output trace. The extension to M multiple experiments is trivial and can be achieved by introducing an initial state $x_{0,j}$ per trace and minimizing the sum of the corresponding loss functions and the regularization terms with respect to $(x_{0,1}, \ldots, x_{0,M}, \theta_x, \theta_y)$. Alternatively, we can parameterize

$$x_{0,i} = f_{x0}(v_{0,i}, \theta_{x0}) \tag{5}$$

where $v_0 \in \mathbb{R}^{n_v}$ is a measured vector available at time 0, such as a collection of n_a past outputs and n_b past inputs, as also suggested in Beintema et al. (2021) and Masti and Bemporad (2021), and/or other measurable values that are known to affect the initial state of the system we want to model. The initial-state encoder function $f_{x0} : \mathbb{R}^{n_v} \times \mathbb{R}^{n_{\theta x0}} \to \mathbb{R}^{n_x}$ is a feedforward neural network defined similarly to f_x, f_y and parameterized by a new vector $\theta_{x0} \in \mathbb{R}^{n_{\theta x0}}$ to be learned jointly with θ_x, θ_y. The regularization term in this case is defined as $r(\theta_{x0}, \theta_x, \theta_y) = r_x(\theta_{x0}) + r_\theta^x(\theta_x) + r_\theta^y(\theta_y)$.

3. Sequential linear least squares

We first handle the smooth case $g(\theta_x, \theta_y) = 0$. Let $(x_0^0, \theta_x^0, \theta_y^0)$ be an initial guess. By simulating the RNN model (1) we get the corresponding sequence of hidden states x_k^0, $k = 1, \ldots, N-1$, and predicted outputs $\hat{y}_k^0$. At a generic iteration h of the algorithm proposed next, we assume that the nominal trajectory is feasible, i.e., $x_{k+1}^h = f_x(x_k^h, u_k, \theta_x^h)$, $k = 0, \ldots, N-2$. We want to find updates $x_k^{h+1}, \theta_x^{h+1}, \theta_y^{h+1}$ by solving a least-squares approximation of problem (2) as described in the next paragraphs.

3.1. Linearization of the dynamic constraints

To get a quadratic approximation of problem (3) we first linearize the RNN dynamic constraints (2b) around x_k^h by approximating $x_{k+1}^h + p_{x_{k+1}} = f_x(x_k^h + p_{x_k}, u_k, \theta_x^h + p_{\theta_x}) \approx f_x(x_k^h, u_k, \theta_x^h) + \nabla_x f_x(x_k^h, u_k, \theta_x^h) p_{x_k} + \nabla_{\theta_x} f_x(x_k^h, u_k, \theta_x^h) p_{\theta_x}$. Since $x_{k+1}^h = f_x(x_k^h, u_k, \theta_x^h)$, we can write the linearized state-update equations as

$$p_{x_{k+1}} = \nabla_x f_x(x_k^h, u_k, \theta_x^h) p_{x_k} + \nabla_{\theta_x} f_x(x_k^h, u_k, \theta_x^h) p_{\theta_x} \tag{6}$$

The dynamics (6) can be condensed to

$$p_{x_k} = M_{kx} p_{x_0} + M_{k\theta_x} p_{\theta_x} \tag{7}$$

where matrices $M_{kx} \in \mathbb{R}^{n_x \times n_x}$ and $M_{k\theta_x} \in \mathbb{R}^{n_x \times n_{\theta_x}}$, $k = 0, \dots, N-2$, are recursively defined as follows:

$$\begin{aligned} M_{0x} &= I, \quad M_{0\theta_x} = 0 \\ M_{(k+1)x} &= \nabla_x f_x(x_k^h, u_k, \theta_x^h) M_{kx} \\ M_{(k+1)\theta_x} &= \nabla_x f_x(x_k^h, u_k, \theta_x^h) M_{k\theta_x} + \nabla_{\theta_x} f_x(x_k^h, u_k, \theta_x^h) \end{aligned} \tag{8}$$

In case x_0 is parameterized as in (5), by approximating $x_0^h \approx f_{x0}(v_0, \theta_{x0}^h) + \nabla_{\theta_{x0}} f_{x0}(v_0, \theta_{x0}^h) p_{\theta_{x0}}$, where $p_{\theta_{x0}}$ is the increment of θ_{x0} to be computed, it is enough to replace p_{x_0} with $p_{\theta_{x0}}$ in (7) and $M_{0x} = \nabla_{\theta_{x0}} f_{x0}(v_0, \theta_{x0}^h)$ in (8).

3.2. Quadratic approximation of the cost function

Let us now find a quadratic approximation of the cost function (4). First, regarding the regularization term r, we take the 2nd-order Taylor expansions

$$\begin{aligned} r_x(x_0 + p_{x_0}) &\approx \frac{1}{2} p_{x_0}' \nabla^2 r_x(x_0^h) p_{x_0} + \nabla r_x(x_0^h) p_{x_0} + r(x_0^h) \\ r_\theta^x(\theta_x^h + p_{\theta_x}) &\approx \frac{1}{2} p_{\theta_x}' \nabla^2 r_\theta^x(\theta_x^h) p_{\theta_x} + \nabla r_\theta^x(\theta_x^h) p_{\theta_x} + r_\theta^x(\theta_x^h) \\ r_\theta^y(\theta_y^h + p_{\theta_y}) &\approx \frac{1}{2} p_{\theta_y}' \nabla^2 r_\theta^y(\theta_y^h) p_{\theta_y} + \nabla r_\theta^y(\theta_y^h) p_{\theta_y} + r_\theta^y(\theta_y^h) \end{aligned}$$

By neglecting the constant terms, minimizing the 2nd-order Taylor expansion of $r(x_0 + p_{x_0}, \theta_x + p_{\theta_x}, \theta_y + p_{\theta_y})$ is equivalent to minimizing the least-squares term

$$\frac{1}{2} \left\| \begin{bmatrix} L_{r0} p_{x0} + (L_{r0}')^{-1} \nabla_x r_x(x_0^h) \\ L_{rx} p_{\theta_x} + (L_{rx}')^{-1} \nabla r_\theta^x(\theta_x^h) \\ L_{ry} p_{\theta_y} + (L_{ry}')^{-1} \nabla r_\theta^y(\theta_y^h) \end{bmatrix} \right\|_2^2 \tag{9}$$

where $L_{r0}' L_{r0} = \nabla^2 r_x(x_0)$, $L_{rx}' L_{rx} = \nabla^2 r_\theta^x(\theta_x)$, and $L_{ry}' L_{ry} = \nabla^2 r_\theta^y(\theta_y)$ are Cholesky factorizations of the corresponding Hessian matrices. Note that in the case of standard ℓ_2-regularization $r_x(x_0) = \frac{\rho_x}{2}\|x_0\|_2^2$, $r_\theta^x(\theta_x) = \frac{\rho_\theta}{2}\|\theta_x\|_2^2$, $r_\theta^y(\theta_y) = \frac{\rho_\theta}{2}\|\theta_y\|_2^2$, we simply have $L_{r0} = \sqrt{\rho_x} I$, $L_{rx} = \sqrt{\rho_\theta} I$, $L_{ry} = \sqrt{\rho_\theta} I$, $\nabla_x r_x(x_0) = \sqrt{\rho_x} x_0$, $r_\theta^x(\theta_x) = \sqrt{\rho_\theta}\theta_x$, $r_\theta^y(\theta_y) = \sqrt{\rho_\theta}\theta_y$, and hence (9) becomes

$$\frac{1}{2} \left\| \begin{bmatrix} \sqrt{\rho_x} p_{x0} + \sqrt{\rho_x} x_0^h \\ \sqrt{\rho_\theta} p_{\theta_x} + \sqrt{\rho_\theta} \theta_x^h \\ \sqrt{\rho_\theta} p_{\theta_y} + \sqrt{\rho_\theta} \theta_y^h \end{bmatrix} \right\|_2^2 \tag{10}$$

Regarding the loss terms penalizing the ith output-prediction error at step k, we have that

$$\begin{aligned} \nabla_{x_k} \ell_i(y_{ki}, f_{yi}(x_k^h, u_k, \theta_y^h)) &= \nabla_{x_k} f_{yi}(x_k^h, u_k, \theta_y^h) \ell_{ik}' \\ \nabla_{\theta_y} \ell_i(y_{ki}, f_{yi}(x_k^h, u_k, \theta_y^h)) &= \nabla_{\theta_y} f_{yi}(x_k^h, u_k, \theta_y^h) \ell_{ik}' \end{aligned}$$

where $\ell_{ik}' \triangleq \frac{d\ell_i(y_k, f_{yi}(x_k^h, u_k, \theta_y^h))}{d\hat{y}_i}$, $\ell_{ik}' \in \mathbb{R}$, and hence, by differentiating again with respect to $\begin{bmatrix} x_k \\ \theta_y \end{bmatrix}$ and setting $\ell_{ik}'' \triangleq \frac{d^2\ell_i(y_k, f_{yi}(x_k^h, u_k, \theta_y^h))}{d\hat{y}_i^2}$, $\ell_{ik}'' \in \mathbb{R}$, $\ell_{ik}'' > 0$, we get $\nabla^2_{\begin{bmatrix} x_k \\ \theta_y \end{bmatrix}\begin{bmatrix} x_k \\ \theta_y \end{bmatrix}} \ell_i(y_{ki}, f_{yi}(x_k^h, u_k, \theta_y^h)) = (\nabla_{\begin{bmatrix} x_k \\ \theta_y \end{bmatrix}} f_{yi})(\nabla_{\begin{bmatrix} x_k \\ \theta_y \end{bmatrix}} f_{yi})' \cdot \ell_{ik}'' + \nabla^2_{\begin{bmatrix} x_k \\ \theta_y \end{bmatrix}\begin{bmatrix} x_k \\ \theta_y \end{bmatrix}} f_{yi}(x_k^h, u_k, \theta_y^h) \ell_{ik}'$ (we have omitted the dependence of ℓ', ℓ'' on the iteration h for simplicity). By neglecting the Hessian of the output function f_{yi} (as it is common in Gauss–Newton methods), the minimization of $\ell_i(y_{ki}, f_{yi}(x_k^h + p_{x_k}, u_k, \theta_x^h + p_{\theta_x}))$ can be approximated by the minimization of $\frac{1}{2} \begin{bmatrix} p_{x_k} \\ p_{\theta_x} \end{bmatrix}' \ell_{ik}'' (\nabla_{\begin{bmatrix} x_k \\ \theta_y \end{bmatrix}} f_{yi})(\nabla_{\begin{bmatrix} x_k \\ \theta_y \end{bmatrix}} f_{yi})' \begin{bmatrix} p_{x_k} \\ p_{\theta_y} \end{bmatrix} + \ell_{ik}' \begin{bmatrix} \nabla_{x_k} f_{yi} \\ \nabla_{\theta_y} f_{yi} \end{bmatrix}' \begin{bmatrix} p_{x_k} \\ p_{\theta_y} \end{bmatrix}$. By strong convexity of the loss function ℓ, this can be further rewritten as the minimization of the least-squares term

$$\frac{1}{2} \left\| (\ell_{ik}'')^{\frac{1}{2}} (\nabla_{\begin{bmatrix} x_k \\ \theta_y \end{bmatrix}} f_{yi})' \begin{bmatrix} p_{x_k} \\ p_{\theta_y} \end{bmatrix} + (\ell_{ik}'')^{-\frac{1}{2}} \ell_{ik}' \right\|_2^2 \tag{11}$$

that, by exploiting (7), is equivalent to

$$\frac{1}{2} \left\| (\ell_{ik}'')^{\frac{1}{2}} \begin{bmatrix} (\nabla_{x_k} f_{yi})' M_{kx} & (\nabla_{x_k} f_{yi})' M_{k\theta_x} & (\nabla_{\theta_y} f_{yi})' \end{bmatrix} \cdot \begin{bmatrix} p_{x_k} \\ p_{\theta_x} \\ p_{\theta_y} \end{bmatrix} + (\ell_{ik}'')^{-\frac{1}{2}} \ell_{ik}' \right\|_2^2 \tag{12}$$

Finally, the minimization of the quadratic approximation of (4) can be recast as the linear LS problem

$$p^h = \arg\min_p \frac{1}{2} \|A^h p - b^h\|_2^2 \tag{13}$$

where $p = \begin{bmatrix} p_{x_0} \\ p_{\theta_x} \\ p_{\theta_y} \end{bmatrix}$, A^h and b^h are constructed by collecting the least-squares terms from (10) and (12), $A^h \in \mathbb{R}^{n_A \times n_p}$, $b^h \in \mathbb{R}^{n_A}$, $n_A = Nn_y + n_x + n_{\theta_x} + n_{\theta_y}$, and $n_p = n_x + n_{\theta_x} + n_{\theta_y}$. The reader is referred to Bemporad (2022) for alternative ways of solving (13) and their numerical complexity.

We remark that most of the computation effort is usually spent in computing the Jacobian matrices $\nabla_{\begin{bmatrix} x_k \\ \theta_x \end{bmatrix}} f_x(x_k^h, u_k, \theta_x^h)$ and $\nabla_{\begin{bmatrix} x_k \\ \theta_y \end{bmatrix}} f_{yi}(x_k^h, u_k, \theta_y^h)$. Given the current nominal trajectory $\{x_k^h\}$, such computations can be completely *parallelized*. Hence, we are not forced to use *sequential* computations as in backpropagation (Werbos, 1990) to solve for p^h in (13), as typically done in classical training methods based on gradient descent.

Note also that for training *feedforward* neural networks ($n_x = 0$), we only optimize with respect to θ_y. In this case, the operations performed at each iteration h to predict $\hat{y}_k$ and compute V^h can also be parallelized.

For the sake of comparison, a gradient-descent method would set

$$\begin{aligned} p^h &= -\nabla V(x_0^h, \theta_x^h, \theta_y^h) \\ &= \begin{bmatrix} \nabla_{x_0} r_x(x_0^h) \\ \nabla_{\theta_x} r_\theta^x(\theta_x^h) \\ \nabla_{\theta_y} r_\theta^y(\theta_y^h) \end{bmatrix} + \sum_{k=0}^{N-1} \sum_{i=1}^{n_y} \ell_{ik}' \begin{bmatrix} \nabla_{x_k} f_{yi}(x_k^h, \theta_y^h)' M_{kx} \\ \nabla_{x_k} f_{yi}(x_k^h, \theta_y^h)' M_{k\theta_x} \\ \nabla_{\theta_y} f_{yi}(x_k^h, \theta_y^h) \end{bmatrix} \end{aligned} \tag{14}$$

since, from (8), $\nabla_{x_0} x_k = M_{kx}$ and $\nabla_{\theta_x} x_k = M_{k\theta_x}$.

3.3. Line search

To enforce the decrease of the cost function (2a) across iterations $h = 0, 1, \dots, E$, where E is the maximum number of epochs we allow processing, a possibility is to update the solution as

$$\begin{bmatrix} x_0^{h+1} \\ \theta_x^{h+1} \\ \theta_y^{h+1} \end{bmatrix} = \begin{bmatrix} x_0^h \\ \theta_x^h \\ \theta_y^h \end{bmatrix} + \alpha_h \begin{bmatrix} p_{x_0}^h \\ p_{\theta_x}^h \\ p_{\theta_y}^h \end{bmatrix} \tag{15}$$

and set $x_{k+1}^{h+1} = f_x(x_k^{h+1}, u_k, \theta_x^{h+1})$, $k = 1, \dots, N-1$. We choose the step-size α_h by finding the largest α satisfying the classical Armijo condition

$$V^{h+1} \le V^h + c_1 \alpha (\nabla V^h)' p^h \tag{16}$$

where V^{h+1}, V^h are the costs defined as in (2) associated with the corresponding combinations of $(x_0, \theta_x, \theta_y)$, the gradient is taken with respect to $\begin{bmatrix} x_0 \\ \theta_x \\ \theta_y \end{bmatrix}$, and c_1 is a constant (e.g., $c_1 = 10^{-4}$) (Nocedal & Wright, 2006, Chapter 3.1). The algorithm is stopped when $V^h - V^{h-1} \le \epsilon_V$, where ϵ_V is a prescribed optimality tolerance, or $h \ge E$.

In this paper we use the common approach of starting with $\alpha = 1$ and then changing $\alpha \leftarrow \sigma\alpha$, $\sigma \in (0, 1)$ until (16) is satisfied. We set a bound N_σ on the number of line-search steps that can be performed, setting $\alpha = 0$ in the case of line-search failure , which makes $V^h = V^{h-1}$ and hence stops the iterations. Since $V(x_0 + \alpha p_{x_0}^h, \theta_x + \alpha p_{\theta_x}^h, \theta_y + \alpha p_{\theta_y}^h) \approx V(x_0, \theta_x^h, \theta_y^h) + \frac{1}{2}\|A^h \alpha p^h -$

$b^h\|_2^2 - \frac{1}{2}\|b^h\|_2^2$ for small values of α, the directional derivative in (16) can be computed from A^h, b^h as

$$\begin{aligned}(\nabla V^h)'p^h &= \lim_{\alpha\to 0}\frac{V(x_0+\alpha p^h_{x_0},\theta_x+\alpha p^h_{\theta_x},\theta_y+\alpha p^h_{\theta_y})-V(x_0,\theta^h_x,\theta^h_y)}{\alpha}\\ &= \lim_{\alpha\to 0}\frac{1}{2}\frac{\|A^h\alpha p^h-b^h\|_2^2-\|b^h\|_2^2}{\alpha} = -(b^h)'A^hp^h\end{aligned}$$

Note that when the loss function $\ell(y,\hat{y}) = \frac{1}{2}\|y-\hat{y}\|_2^2$ and the regularization terms r_x, r_{θ_y}, r_{θ_x} are quadratic, the loss $V(x_0,\theta_x^h,\theta_y^h) = \frac{1}{2}(b^h)'b^h$ by construction. See Bemporad (2022) for further details.

The proposed method belongs to the class of Generalized Gauss–Newton (GGN) methods (Messerer, Baumgärtner, & Diehl, 2021) applied to solve problem (4), with the addition of line search. In fact, following the notation in Messerer et al. (2021), we can write $V(x_0,\theta_x,\theta_y) = \Phi_0(F(w))$ where $w \triangleq [x_0'\ \theta_x'\ \theta_y]'$ is the optimization vector, function $\Phi_0 : \mathbb{R}^{N_\phi} \to \mathbb{R}$, $N_\phi \triangleq n_x + n_{\theta_x} + n_{\theta_y} + Nn_y$, is given by the composition $\Phi_0 = \Phi_+ \circ \Phi_\ell$ of the summation function $\Phi_+ : \mathbb{R}^{N_\phi} \to \mathbb{R}$, $\Phi_+(x) = \sum_{i=1}^{N_\phi} x_i$, with the vectorized loss $\Phi_\ell : \mathbb{R}^{N_\phi} \to \mathbb{R}^{N_\phi}$, $\Phi_\ell([x_0'\ \theta_x'\ \theta_y'\ \hat{y}_0'\ \dots\ \hat{y}_{N-1}']') = [r_x(x_0)\ r_\theta^x(\theta_x)\ r_\theta^y(\theta_y)\ \ell(y_0,\hat{y}_0)'\ \dots\ \ell(y_{N-1},\hat{y}_{N-1})']'$. By assumption, all the components of Φ_ℓ are strongly convex functions, and since Φ_+ is also strongly convex, Φ_0 is strongly convex with respect to its argument $F(w)$. As in GGN methods, we take a second-order approximation of Φ_0 and a linearization of $F(w)$ during the iterations. In Messerer et al. (2021), local convergence under the full step $\alpha_h \equiv 1$, $\forall h \geq 0$, is shown under additional assumptions on Φ_0 and the neglected second-order derivatives of F, while here we force the monotonic decrease of the loss function V at each iteration h by line search, hence obtaining a "damped" GGN algorithm.

3.4. A Levenberg–Marquardt variant

In alternative to the line-search method based on (16), we can force monotonicity of the objective function by adopting a trust-region method, in particular, the well-known Levenberg–Marquardt (LM) algorithm (Levenberg, 1944; Marquardt, 1963). In LM methods, (13) is changed to

$$p^h = \arg\min_p \frac{1}{2}\|A^hp - b^h\|_2^2 + \frac{1}{2}\lambda_h\|p\|_2^2 \tag{17}$$

where the regularization term λ_h is tuned at each iteration h to meet the condition $V^h \leq V^{h-1}$. It is easy to show that tuning λ_h is equivalent to tuning the trust-region radius, see e.g. Nocedal and Wright (2006, Chapter 10). Clearly, setting $\lambda_h = 0$ corresponds to the full GGN step, as in line-search when $\alpha_h = 1$, and $\lambda_h \to \infty$ corresponds to $p^h \to 0$ ($\alpha_h \to 0$). Several methods exist to update λ_h (Transtrum & Sethna, 2012). In this paper, we take the simplest approach suggested in Transtrum and Sethna (2012, Section 2.1) and, starting from $\lambda_h = \lambda_{h-1}$, keep multiplying $\lambda_h \leftarrow c_2\lambda_h$ until the condition $V^h \leq V^{h-1}$ is met, then reduce $\lambda_h \leftarrow \lambda_h/c_3$ before proceeding to step $h+1$.

Similarly to the line-search method, we set a bound N_λ on the number of least-squares problems that can be solved at each epoch h. Again, the LM algorithm is stopped when either $V^h - V^{h-1} \leq \epsilon_V$, or $h \geq E$.

The main difference between performing line search and computing LM iterations instead is that at each epoch h, the former only solves the linear least-squares problems (13), the latter requires solving up to N_λ problems (17). Note that, however, both only construct A^h and b^h once at each iteration h, which, as remarked earlier, is usually the dominant computation effort due to the evaluation of the required Jacobian matrices $\nabla_x f_x$, $\nabla_{\theta_x} f_x$, $\nabla_x f_y$, $\nabla_{\theta_y} f_y$.

3.5. Initialization

In this paper, we initialize $x_0^0 = 0$ (unless an additional state-encoder network f_{x0} is used as described in (5)), set zero initial bias terms, and draw the remaining components of θ_x^0 and θ_y^0 from the normal distribution with zero mean and standard deviation defined as in Glorot and Bengio (2010), further multiplied by the quantity $\sigma_0 \leq 1$. The rationale to select small initial values of θ_x^0, θ_y^0 is to avoid that the initial dynamics f_x used to compute x_k^0 for $k = 1,\dots,N-1$ are unstable. Then, since the overall loss V^h is decreasing, the divergence of state trajectories is unlikely to occur at subsequent iterations. Note that this makes the elimination of the state variables $x_1,\dots,x_{N-1}$ in the condensing approach (3) suitable for the current RNN training setting, contrarily to solving nonlinear model predictive problems in which the initial state x_0 and the model coefficients θ_x, θ_y are fixed and may therefore lead to excite unstable linearized model responses, with consequent numerical issues (Bemporad & Cimini, 2023). Note also that ideas to enforce open-loop stability could be introduced in our setting by adopting ideas as in Kolter and Manek (2019), where the authors propose to also learn a Lyapunov function while learning the model.

4. Non-smooth regularization

We now want to solve the full problem (2) by considering also the additional non-smooth regularization term $g(\theta_x,\theta_y)$. Consider again the condensed loss function (4) and rewrite problem (2) according to the following split

$$\begin{aligned}\min_{x_0,\theta_x,\theta_y,\nu_x,\nu_y}\ & V(x_0,\theta_x,\theta_y) + g(\nu_x,\nu_y)\\ \text{s.t. }& \theta_x = \nu_x,\quad \theta_y = \nu_y\end{aligned} \tag{18}$$

We solve problem (18) by executing the following scaled ADMM iterations

$$\begin{bmatrix}x_0(t+1)\\ \theta_x(t+1)\\ \theta_y(t+1)\end{bmatrix} = \operatorname*{argmin}_{x_0,\theta_x,\theta_y} V(x_0,\theta_x,\theta_y) + \frac{\rho}{2}\left\|\begin{bmatrix}\theta_x-\nu_x(t)+w_x(t)\\ \theta_y-\nu_y(t)+w_y(t)\end{bmatrix}\right\|_2^2 \tag{19a}$$

$$\begin{bmatrix}\nu_x(t+1)\\ \nu_y(t+1)\end{bmatrix} = \operatorname*{argmin}_{\nu_x,\nu_y}\frac{1}{\rho}g(\nu_x,\nu_y) + \frac{1}{2}\left\|\begin{bmatrix}\nu_x-\theta_x(t+1)-w_x(t)\\ \nu_y-\theta_y(t+1)-w_y(t)\end{bmatrix}\right\|_2^2$$

$$= \operatorname{prox}_{\frac{1}{\rho}g}(\theta_x(t+1)+w_x(t),\theta_y(t+1)+w_y(t)) \tag{19b}$$

$$\begin{bmatrix}w_x(t+1)\\ w_y(t+1)\end{bmatrix} = \begin{bmatrix}w_x(t)+\theta_x(t+1)-\nu_x(t+1)\\ w_y(t)+\theta_y(t+1)-\nu_y(t+1)\end{bmatrix} \tag{19c}$$

for $t = 0,\dots,N_{\text{ADMM}}-1$, where "prox" denotes the proximal operator. Problem (19a) is solved by running sequential least-squares with line search or its LM variant, with initial condition $x_0^0 = x_0(t)$, $\theta_x^0 = \theta_x(t)$, $\theta_y^0 = \theta_y(t)$. Note that when solving the least-squares problem (13) or (17), A^h, b^h are augmented to include the additional ℓ_2-regularization term $\frac{1}{2}\left\|\sqrt{\rho}\begin{bmatrix}p_{\theta_x}\\ p_{\theta_y}\end{bmatrix} - \sqrt{\rho}\begin{bmatrix}\nu_x(t)-\theta_x^h-w_x(t)\\ \nu_y(t)-\theta_y^h-w_y(t)\end{bmatrix}\right\|_2^2$ introduced in the ADMM iteration (19a).

Our numerical experiments show that setting $E = 1$ is a good choice, which corresponds to just computing one sequential LS iteration to solve (19a) approximately. The overall algorithm that we call NAILS (nonconvex ADMM iterations) when line-search is used, or NAILM when the Levenberg–Marquardt method is instead used, is summarized in Algorithm 1. Note that NAILS/NAILM processes the training dataset for a maximum of $E\cdot N_{\text{ADMM}}$ epochs.

The final model parameters are given by $\theta_x = \nu_x(t)$, $\theta_y = \nu_y(t)$, $x_0(t)$, where $t = N_{\text{ADMM}}$ is the last ADMM iteration executed. In case x_0 also needs to be regularized by a non-smooth penalty, we can extend the approach to include a further split $\nu_{x_0} = x_0$.

Algorithm 1 can be modified to return the initial state and model obtained after each ADMM step that provide the best loss

Algorithm 1. Nonconvex ADMM iterations and least squares, either with a line-search (NAILS) or Levenberg–Marquardt (NAILM) approach.

Input: Training dataset $\{u_k, y_k\}_{k=0}^{N-1}$, initial guess x_0^0, θ_x^0, θ_y^0, number N_{ADMM} of ADMM iterations, ADMM parameter $\rho > 0$, maximum number E of epochs, tolerance $\epsilon_V > 0$; line-search parameters $c_1 > 0$, $\sigma \in (0,1)$, and $N_\sigma > 1$ (NAILS), or LM parameters $\lambda_0 > 0$, $c_2, c_3 > 1$, and $N_\lambda > 0$ (NAILM).

1. $\begin{bmatrix} v_x^0 \\ v_y^0 \end{bmatrix} \leftarrow \begin{bmatrix} \theta_x^0 \\ \theta_y^0 \end{bmatrix}$, $\begin{bmatrix} w_x^0 \\ w_y^0 \end{bmatrix} \leftarrow 0$;
2. **for** $t = 0, 1, \ldots, N_{\text{ADMM}} - 1$ **do**:
   - 2.1. **update** $x_0(t+1)$, $\theta_x(t+1)$, and $\theta_y(t+1)$ as in Eq. (19a), using one of the methods described in Section 3;
   - 2.2. **update** $\begin{bmatrix} v_x(t+1) \\ v_y(t+1) \end{bmatrix}$ as in Eq. (19b) and $\begin{bmatrix} w_x(t+1) \\ w_y(t+1) \end{bmatrix}$ as in Eq. (19c);
   - 2.3. $t \leftarrow t+1$;
3. **end**.

Output: RNN parameters $\theta_x = v_x(t), \theta_y = v_y(t)$ and initial hidden state $x_0(t)$.

$\sum \ell(y_k, \hat{y}_k)$ observed during the execution of the algorithm on training data or, if available, on a separate validation dataset.

Note that the iterations (19) are not guaranteed to converge to a global optimum of the problem. Moreover, it is well known that in some cases, nonconvex ADMM iterations may even diverge (Houska, Frasch, & Diehl, 2016). The reader is referred to Hong, Luo, and Razaviyayn (2016), Houska et al. (2016), Themelis and Patrinos (2020) and Wang, Yin, and Zeng (2019) for convergence conditions and/or alternative ADMM formulations.

We finally remark that, in the absence of the non-smooth regularization term g, Algorithm 1 simply reduces to a smooth GGN method running with tolerance ϵ_V for a maximum number E of epochs.

4.1. ℓ_1-regularization

A typical instance of regularization is the ℓ_1-penalty

$$g(\theta_x, \theta_y) = \tau_x \|\theta_x\|_1 + \tau_y \|\theta_y\|_1 \tag{20}$$

to attempt pruning an overly-parameterized network structure, with $\tau_x, \tau_y \geq 0$. In this case, the update in (19b) simply becomes $v_x(t+1) = S_{\frac{\tau_x}{\rho}}(\theta_x(t+1)+w_x^h)$, $v_y(t+1) = S_{\frac{\tau_y}{\rho}}(\theta_y(t+1)+w_y^h)$, where S_β is the soft-threshold operator $[S_\beta(w)]_i = \max\{w_i - \beta, 0\} + \min\{w_i + \beta, 0\}$.

4.2. Group-lasso regularization and model reduction

The complexity of the structure of the networks f_x, f_y can be reduced by using group-Lasso penalties (Yuan & Lin, 2006) that attempt zeroing entire subsets of parameters. In particular, the order n_x of the RNN can be penalized by creating n_x groups θ_i^g of variables, $\theta_i^g \in \mathbb{R}^{n_g}$, $n_g = n_1^x + n_1^y + n_{L_x-1}^x + 1$, $i = 1, \ldots, n_x$, each one stacking the ith columns of A_1^x and A_1^y, the ith row of A_{L_x}, and the ith entry of b_{L_x}. In this case, in (18) we only introduce the splitting $\theta_i^g = v_i^g$ on those variables and penalize $g(v_i^g) = \tau_g \sum_{i=1}^{n_x} \|v_i^g\|_2$, $\tau_g \geq 0$. Then, we update $v_i^g(t+1) = S^g_{\frac{\tau_g}{\rho}}(\theta_i^g(t+1) + w_i^g(t))$, where $S_\beta^g(w) = (1 - \frac{\beta}{\|w\|_2})w$ if $\|w\|_2 > \beta$, or 0 otherwise, is the block soft thresholding operator.

The larger the penalty τ_g, the larger is expected the number of state indices i such that the ith column of A_1^x and A_1^y, the ith row of A_{L_x}, and the ith entry of b_{L_x} are all zero. Consequently, the corresponding ith hidden state does not influence the model.

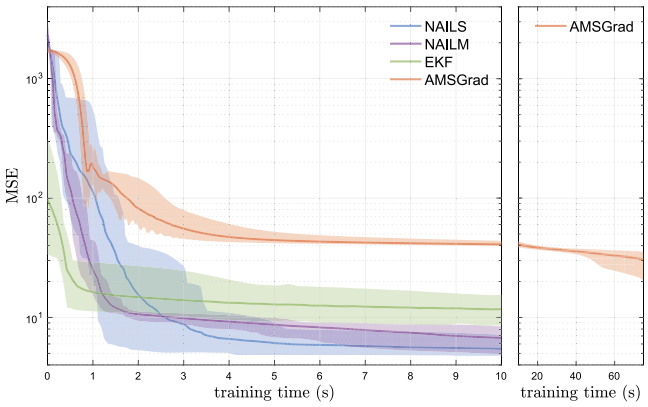

Fig. 1. MSE loss (4) evaluated on training data as a function of training time (mean value and range computed over 20 runs from different random initial weights).

4.3. ℓ_0-regularization

The ℓ_0-regularization term $g(\theta_x, \theta_y) = \tau_x^0 \|\theta_x\|_0 + \tau_y^0 \|\theta_y\|_0$, $\tau_x^0, \tau_y^0 \geq 0$, also admits the explicit proximal "hard-thresholding" operator $[\text{prox}_{\beta\|x\|_0}]_i = x_i$ if $x_i^2 \geq 2\beta$, or 0 otherwise, and can be used in alternative to (20).

4.4. Mixed-integer constraints

Constraints on θ_x, θ_y that can be expressed as mixed-integer linear inequalities

$$A_{\text{MI}} \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} \leq b_{\text{MI}}, \quad \theta_{xi}, \theta_{y_j} \in \{0, 1\}, \ \forall i \in I_x, \ j \in I_y \tag{21}$$

where $I_x \subseteq \{1, \ldots, n_{\theta_x}\}$, $I_y \subseteq \{1, \ldots, n_{\theta_y}\}$, $A_{\text{MI}} \in \mathbb{R}^{n_{\text{MI}} \times (n_{\theta_x} + n_{\theta_y})}$, $b_{\text{MI}} \in \mathbb{R}^{n_{\text{MI}}}$, $n_{\text{MI}} \geq 0$, can be handled by setting g as the indicator function

$$g(\theta_x, \theta_y) = \begin{cases} 0 & \text{if (21) is satisfied} \\ +\infty & \text{otherwise} \end{cases}$$

In this case, v_x^{h+1}, v_y^{h+1} can be computed from (19b) by solving the mixed-integer quadratic programming (MIQP) problem

$$\begin{aligned} \min_{v_x, v_y} \quad & \tfrac{1}{2} \left\| \begin{bmatrix} v_x - \theta_x(t+1) - w_x(t) \\ v_y - \theta_y(t+1) - w_y(t) \end{bmatrix} \right\|_2^2 \\ \text{s. t.} \quad & A_{\text{MI}} \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} \leq b_{\text{MI}} \\ & \theta_{xi}, \theta_{y_j} \in \{0, 1\} \ \forall i \in I_x, \ j \in I_y \end{aligned} \tag{22}$$

Note that in the special case $n_{\text{MI}} = 0$, (22) has the following explicit solution (Takapoui, Moehle, Boyd, & Bemporad, 2020)

$$\begin{aligned} v_{xi}(t+1) &= \text{round}(\theta_{xi}(t+1) + w_{xi}(t)), \ i \in I_x \\ v_{yj}(t+1) &= \text{round}(\theta_{xj}(t+1) + w_{xj}(t)), \ j \in I_y \end{aligned} \tag{23}$$

where $\text{round}(\beta) = 0$ if $\beta < 0.5$, or 1 otherwise.

Binary constraints can be extended to more general quantization constraints $\theta_{xi}, \theta_{y_j} \in \mathcal{Q} \triangleq \{q_1, \ldots, q_{n_Q}\} \subset \mathbb{R}$, $\forall i \in I_x$, $\forall j \in I_y$. Such constraints can be handled by setting $v_{xi}(t+1)$ equal to the value in $\mathcal{Q}$ that is closest to $\theta_{xi}(t+1) + w_{xi}^h$, and similarly for $v_{yj}(t+1)$.

5. Numerical experiments

We test NAILS and NAILM on three nonlinear system identification problems. The hyper-parameters of Algorithm 1 are

Table 1
Fluid damper benchmark: mean (standard deviation) of BFR on training and test data.

BFR	Training	Test
NAILS	94.41 (0.27)	89.35 (2.63)
NAILM	94.07 (0.38)	89.64 (2.30)
EKF	91.41 (0.70)	87.17 (3.06)
AMSGrad	84.69 (0.15)	80.56 (0.18)

summarized in Bemporad (2022, Table 1). All computations have been carried out in MATLABR2022b on an Apple M1 Max CPU, using the library CasADi (Andersson, Gillis, Horn, Rawlings, & Diehl, 2019) to compute the Jacobian matrices via automatic differentiation. The scaling factor $\sigma_0 = 0.15$ is used to initialize the weights of the neural networks in all the experiments. Unless the state encoder (5) is used, after training a model the best initial condition x_0 is computed by solving the small-scale non-convex simulation-error minimization problem $\min_{x_0} \rho_x\|x_0\|_2^2 + \sum_{k=0}^{N_0} \ell(y_k, \hat{y}_k)$ by using the particle swarm optimizer (PSO) PSwarm (Vaz & Vicente, 2009) with initial population of $2n_x$ samples, each component of x_0 constrained in $[-3, 3]$, and with $N_0 = 100$. When solving the training problems by gradient descent methods, the learning rate of the latter are selected by trial and error to achieve a good tradeoff between increasing the convergence rate and reducing oscillations.

5.1. RNN training with smooth quadratic loss

We first test NAILS/NAILM against the EKF approach (Bemporad, 2023) and gradient-descent based on the AMSGrad algorithm (Reddi, Kale, & Kumar, 2019), for which we compute the gradient as in (14), on real-world data from the nonlinear magneto-rheological fluid damper benchmark proposed in Wang, Sano, Chen, and Huang (2009). We use $N = 2000$ data samples for training and 1499 samples for testing the model, obtained from the System Identification (SYS-ID) Toolbox for MATLAB (Ljung, 2001), where a nonlinear autoregressive (NLARX) model identification algorithm is employed for black-box nonlinear modeling.

We consider a RNN model (1) with $n_x = 4$ hidden states and shallow state-update and output network functions ($L_x = L_y = 2$) with $n_1^x = n_1^y = 4$ neurons, hyperbolic-tangent activation functions f_1^x, f_1^y, and linear output function f_2^y, parameterized by $\theta_x \in \mathbb{R}^{44}$, $\theta_y \in \mathbb{R}^{29}$. The CPU time to retrieve x_0 by PSO is approximately 30 ms in all tests.

In (2) we use the quadratic loss $\ell(y_k, \hat{y}_k) = \frac{1}{2\sigma_y^2}\|y_k - \hat{y}_k\|_2^2$, where σ_y is the standard deviation of the output training data, and quadratic regularization $r(x_0, \theta_x, \theta_y) = \frac{1}{2}(\rho_x\|x_0\|_2^2 + \rho_\theta\|\theta_x\|_2^2 + \rho_\theta\|\theta_x\|_2^2)$ with $\rho_x = 1$, $\rho_\theta = 0.1$. As $g(\theta_x, \theta_y) = 0$, NAILS and NAILM only solve a nonlinear least-squares problem once with parameters $\epsilon_V = 10^{-6}$ and, respectively, $c_1 = 10^{-4}$, $\sigma = 0.5$, $N_\sigma = 20$ (NAILS), and $\lambda_0 = 100$, $c_2 = 1.5$, $c_3 = 5$, $N_\lambda = 20$ (NAILM). The initial condition is $x_0^0 = 0$ and the components of θ_x^0, θ_y^0 are randomly generated as described in Section 3.5.

Table 1 shows the mean and standard deviation (computed over 20 experiments, each one starting from a different random set of weights and zero bias terms) of the final best fit rate BFR $= 100(1 - \|Y - \hat{Y}\|_2/\|Y - \bar{y}\|_2)$, where Y is the vector of measured output samples, $\hat{Y}$ the vector of output samples simulated by the identified model fed in open-loop with the input data, and $\bar{y}$ is the mean of Y, achieved on training and test data. For illustration, the evolution of the MSE loss $\frac{1}{2N}\sum_{k=0}^{N-1}(y_k - \hat{y}_k)^2$ as a function of training time is shown in Fig. 1 for each run. The average CPU time per epoch spent by the training algorithm is 54.3 ms (NAILS), 72.2 ms (NAILM), 253.3 ms (EKF), and 34.6 ms (AMSGrad).

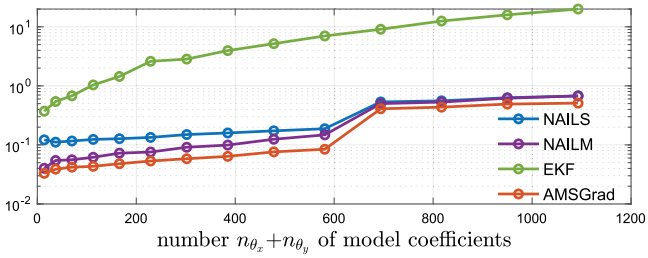

Fig. 2. Fluid damper benchmark: average CPU time (s) per epoch against number $n_{\theta_x} + n_{\theta_y}$ of model coefficients.

It is apparent that NAILS and NAILM obtain similar quality of fit and are computationally comparable. They get better fit results than EKF on average and, not surprisingly, outperform gradient descent. On the other hand, EKF converges more quickly to a good quality of fit, as the model parameters θ_x, θ_y are updated *within* each epoch, rather than *after* each epoch is processed.

Fig. 2 shows how the average CPU time *per epoch* spent by the tested learning algorithms scales as a function of the number $n_{\theta_x} + n_{\theta_y}$ of trained model parameters using the same training dataset. Each model contains n_x states, $n_x \in \{1, \ldots, 14\}$, and 2 hidden layers ($L_x = L_y = 3$) containing n_x neurons each. Note that the considered algorithms require a substantially different *total* number E of epochs to reach a comparable model quality.

5.1.1. Silverbox benchmark problem

We test the proposed training algorithm on the Silverbox dataset, a popular benchmark for nonlinear system identification, for which we refer the reader to Wigren and Schoukens (2013) for a detailed description. The first 40 000 samples are used as the test set, the remaining samples, which correspond to a sequence of 10 different random odd multi-sine excitations, are manually split in $M = 10$ different training traces of about 8600 samples each (the intervals between each different excitation are removed from the training set, as they bring no information).

We consider a RNN model (1) with $n_x = 8$ hidden states, no I/O feedthrough ($A_1^y \in \mathbb{R}^{n_1^y \times n_x}$), and state-update and output network functions with three hidden layers ($L_x = L_y = 4$) with 8 neurons each, a neural network model f_{x0} (5) with two hidden layers of 4 neurons each mapping the vector v_0 of past $n_a = 8$ outputs and $n_b = 8$ inputs to the initial state x_0, hyperbolic-tangent activation functions, and linear output functions. The total number of parameters is $n_{\theta_x} + n_{\theta_y} + n_{\theta_{x0}} = 296 + 225 + 128 = 649$, that we train using NAILM[1] on $E = 150$ epochs with $\rho_\theta = 0.01$ and regularization $\frac{0.1}{2}\|\theta_{x0}\|_2^2$ on the parameters defining f_{x0}. For comparison, we consider the autoregressive models considered in Ljung, Zhang, Lindskog, and Juditski (2004), in particular the ARX model `ml` and NLARX models `ms`, `mlc`, `ms`, and `ms8c50`, that we trained on the same dataset by using the SYS-ID Toolbox (Ljung, 2001) using the same commands reported in Ljung et al. (2004). Note that the latter two models explicitly contain the nonlinear term y_{k-1}^3, in accordance with the physics-based model of the Silverbox system, which is an electronic implementation of the Duffing oscillator. The results of the comparison on test data are shown in Table 2, which also includes the results reported in the recent paper (Beintema et al., 2021) and those obtained by the LSTM model proposed in Ljung, Andersson, Tiels, and Schön (2020). The results reproduced here differ from those reported originally in the papers that we show included in brackets (the results originally reported in Ljung et al. (2020) have been omitted, as they were obtained on a reduced subset of test and training data).

[1] The resulting model can be retrieved at http://cse.lab.imtlucca.it/~bemporad/shared/silverbox/rnn888.zip.

Table 2
Silverbox benchmark: RMSE (mV) and BFR on test data obtained by different methods. The numbers in brackets refer to the results reported in the corresponding original papers.

Identification method	RMSE	BFR
ARX (ml) (Ljung et al., 2004)	16.29 [4.40]	69.22 [73.79]
NLARX (ms) (Ljung et al., 2004)	8.42 [4.20]	83.67 [92.06]
NLARX (mlc) (Ljung et al., 2004)	1.75 [1.70]	96.67 [96.79]
NLARX (ms8c50) (Ljung et al., 2004)	1.05 [0.30]	98.01 [99.43]
Recurrent LSTM model (Ljung et al., 2020)	2.20	95.83
SS encoder (Beintema et al., 2021) ($n_x = 4$)	[1.40]	[97.35]
NAILM (this paper)	0.35	99.33

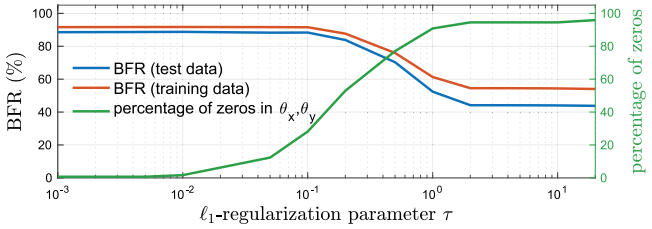

Fig. 3. Fluid damper benchmark: BRF and sparsity of θ_x, θ_y for different ℓ_1-regularization coefficients τ (mean results over 20 runs from different random initial weights).

5.2. RNN training with ℓ_1-regularization

We consider the fluid damper benchmark dataset again and add an ℓ_1-regularization term $g(\theta_x, \theta_y)$ as in (20) and use NAILS to train the same RNN model structure (1), for different values of $\tau_x = \tau_y \triangleq \tau$. Fig. 3 shows the resulting BRF. Fig. 3 also shows the percentage of zero coefficients in the resulting parameter vector $\begin{bmatrix}\theta_x\\ \theta_y\end{bmatrix} \in \mathbb{R}^{73}$, i.e., its sparsity.

The benefit of using ℓ_1-regularization to reduce the complexity of the model without sacrificing fit quality is clear. In fact, the BRF value on test data remains roughly constant until $\tau \approx 0.1$, which corresponds to $\approx 30\%$ zero coefficients in the model.

In order to compare NAILS/NAILM with other state-of-the-art training algorithms that can deal with ℓ_1-regularization terms, we report in Table 3 the results obtained by solving the training problem for $\tau = 0.2$ using the EKF approach (Bemporad, 2023), some of the SGD algorithms mostly used in machine-learning packages (Adam (Kingma & Ba, 2014), AMSGrad (Reddi et al., 2019), and diffGrad (Dubey et al., 2019)), the state-of-the-art non-smooth NLP solvers GRANSO (Curtis et al., 2017) and HANSO (Burke et al., 2020) with default options, and, although not explicitly conceived to handle non-smooth terms, MATLAB's interior-point NLP solver `fmincon`. NAILS/NAILM is run for $E = 250$, SGD for $E = 2000$, EKF for $E = 50$, GRANSO/HANSO/`fmincon` for $E = 1000$ epochs, where the number of epochs has been selected to ensure a good convergence of the solvers. In fact, during our tests, we observed that SGD and NLP solvers require many more iterations than NAILS/NAILM and EKF to achieve solutions with a good percentage of zeros. All the methods but EKF compute the gradient ∇V of the smooth part of the loss function as in (14). SGD and NLP solvers take $\nabla\|\theta\|_1 \triangleq \text{sign}(\theta)$. While all methods get a similar BFR, NAILS and NAILM are more effective not only in execution time but especially in sparsifying θ_x, θ_y, which is the ultimate reason for introducing ℓ_1-penalties.

For the Silverbox benchmark problem, following the analysis performed in Marconato, Schoukens, Rolain, and Schoukens (2013), Fig. 4 shows the obtained RMSE on test data as a function of the number of nonzero parameters in the model that are obtained by running NAILM to train the same RNN model structure defined in Section 5.1.1 under an additional ℓ_1-regularization term with weight τ between 10^{-4} and 20. The figure shows the

Table 3
Fluid damper benchmark: mean BRF, sparsity of $[\theta_x' \ \theta_y']'$ (standard deviation in parentheses), and average total CPU time to process all epochs of different solution methods when solving the ℓ_1-regularized training problem with $\tau = 0.2$ (results obtained over 20 runs from different random initial weights). When using NAILS/NAILM about $\frac{3}{4}$ of the entries in θ_x and half the entries in θ_y are zeroed.

Training algorithm	BFR training	BFR test	Sparsity %	CPU time
NAILS	91.00 (1.66)	87.71 (2.67)	65.1 (6.5)	11.4 s
NAILM	91.32 (1.19)	87.80 (1.86)	64.1 (7.4)	11.7 s
EKF	89.27 (1.48)	86.67 (2.71)	47.9 (9.1)	13.2 s
AMSGrad	91.04 (0.47)	88.32 (0.80)	16.8 (7.1)	64.0 s
Adam	90.47 (0.34)	87.79 (0.44)	8.3 (3.5)	63.9 s
diffGrad	90.05 (0.64)	87.34 (1.14)	7.4 (4.5)	63.9 s
GRANSO	91.58 (0.91)	88.15 (1.40)	44.1 (11.2)	41.9 s
HANSO	91.66 (0.65)	88.41 (1.17)	44.1 (11.2)	40.5 s
fmincon	92.07 (0.63)	88.95 (1.10)	23.2 (13.7)	32.4 s

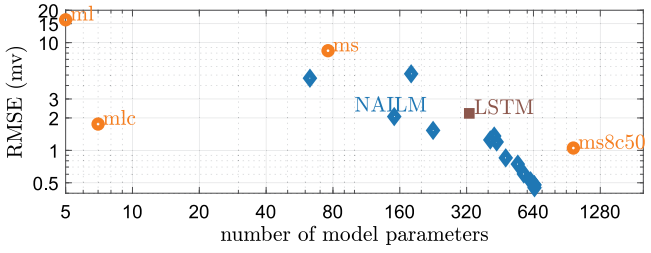

Fig. 4. Silverbox benchmark: RMSE on test data vs. number of parameters. Models ml, ms, mlc, and ms8c50 (Ljung et al., 2004), LSTM model (Ljung et al., 2020), and RNN models generated by NAILM for values of the ℓ_1-regularization parameter τ between 0 and 20.

effectiveness of NAILM in balancing the tradeoff between model simplicity and quality of fit.

5.3. RNN training with quantization

Consider again the fluid damper benchmark dataset and let θ_x, θ_y be only allowed to take values in the finite set $\mathcal{Q}$ of the multiples of 0.1 between -0.5 and 0.5. We also change the activation function of the neurons to the leaky-ReLU function $f(x) = \max\{x, 0\} + 0.1\min\{x, 0\}$, so that the evaluation of the resulting model amounts to extremely simple arithmetic operations, and increase the number of hidden neurons to $n_1^x = n_1^y = 6$. We train the model using NAILS to handle the non-smooth and nonconvex regularization term $g(\theta_x, \theta_y) = 0$ if $\theta_x \in \mathcal{Q}^{n_{\theta_x}}$, $\theta_y \in \mathcal{Q}^{n_{\theta_y}}$, $g(\theta_x, \theta_y) = +\infty$ otherwise, with $E = 1$, $N_{\text{ADMM}} = 200$, $\rho = 10$, without changing the remaining parameters. For comparison, we solve the same problem without non-smooth regularization term g by running sequential LS with line-search for $E = N_{\text{ADMM}}$ epochs and then quantize the components of the resulting vectors θ_x, θ_y to their closest values in $\mathcal{Q}$.

Table 4
Fluid damper benchmark with quantized model coefficients: mean BRF and CPU time (standard deviation) over 20 runs.

BFR	NAILS	Post-quantization
Training	84.36 (3.00)	17.64 (6.41)
Test	78.43 (4.38)	12.79 (7.38)
CPU time	12.04 (0.54)	7.19 (2.82)

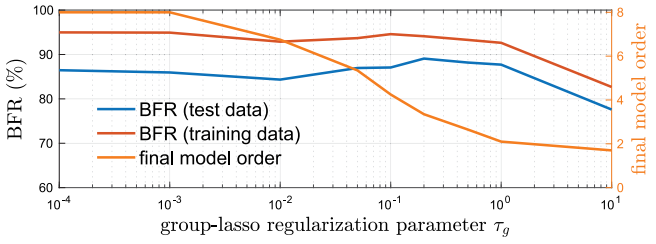

Fig. 5. BRF and resulting model order n_x for different group-Lasso regularization coefficients τ_g (mean results over 20 runs from different random initial weights).

The mean and standard deviation of the BRF and corresponding CPU time obtained over 20 runs from different random initial weights are reported in Table 4. Quantizing a posteriori the parameters of a trained RNN leads to much poorer results.

5.4. Group-lasso penalty and model-order reduction

We use the group-Lasso penalty τ_g described in Section 4.2 to control the model order n_x when training a RNN model (1) on the fluid damper benchmark dataset. We consider the same settings as in Section 5.1, except for the model-order $n_x = 8$, $n_1^x = n_1^y = 6$ neurons in the hidden layers of the state-update and output functions, and set the ADMM parameter $\rho = 10\tau_g$. The mean BRF results over 20 different runs of the NAILS algorithm obtained for different values of τ_g are reported in Fig. 5, along with the resulting mean model order obtained. From the figure, one can see that $n_x = 3$ is a good candidate model order to provide the best BRF on test data.

5.5. RNN training with smooth non-quadratic loss

To test the effectiveness of the proposed approach in handling non-quadratic strongly convex and smooth loss terms, we consider 2000 input/output pairs generated by the following nonlinear system with binary outputs

$$\begin{aligned} x(k+1) &= \begin{bmatrix} .8 & .2 & -.1 \\ 0 & .9 & .1 \\ .1 & -.1 & .7 \end{bmatrix} \mathrm{diag}(x(k))(0.9 + 0.1\sin(x(k))) \\ &\quad + \begin{bmatrix} -.1 \\ .5 \\ 1 \end{bmatrix} u(k)(1-u^3(k)) + \xi(k) \\ y(k) &= \begin{cases} 1 & \text{if } \begin{bmatrix} -2 \\ 1.5 \\ 0.5 \end{bmatrix}' (x(k) + \frac{1}{3}x^3(k)) - 4 + \zeta(k) \geq 0 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

from $x(0) = 0$, with the values of the input $u(k)$ changed with 90% probability from step k to $k+1$ with a new value drawn from the uniform distribution on $[0, 1]$. We consider independent noise signals $\xi_i(k), \zeta(k) \sim \mathcal{N}(0, \sigma_n^2)$, $i = 1, 2, 3$, for different values of the standard deviation σ_n. The first $N = 1000$ samples are used for training, and the rest for testing the model. NAILS and NAILM are used to train a RNN model (1) with no feedthrough, one hidden layer ($L_x = L_y = 2$) with 5 neurons and hyperbolic-tangent activation function, and sigmoid output function $f_2^y(y) = 1/(1 + e^{-A_2^y[x'(k)\ u(k)]' - b_2^y})$. The resulting model-parameter vectors $\theta_x \in \mathbb{R}^{43}$ and $\theta_y \in \mathbb{R}^{26}$ are trained using the modified cross-entropy loss $\ell(y(k), \hat{y}) = \sum_{i=1}^{n_y} -y_i(k)\log(\epsilon + \hat{y}_i) - (1-y_i(k))\log(1+\epsilon - \hat{y}_i)$ (Bemporad, 2023), where $\hat{y} = f_y(x_k, \theta_y)$ and $\epsilon = 10^{-4}$, quadratic regularization with $\rho_x = 0.1$, $\rho_\theta = 0.01$, optimality tolerance $\epsilon_V = 10^{-6}$, parameters $c_1 = 10^{-4}$, $\sigma = 0.5$, $N_\sigma = 10$ (NAILS), and $\lambda_0 = 100$, $c_2 = 1.5$, $c_3 = 5$, $N_\lambda = 30$ (NAILM), for maximum $N_{\text{ADMM}} = 150$ epochs.

Table 5
Nonlinear system with binary outputs: mean final accuracy (%) (standard deviation) over 20 runs on training and test data.

σ_n		Accuracy (%) Training data	Accuracy (%) Test data
0.00	NAILS	99.16 (0.6)	96.87 (1.0)
	NAILM	99.30 (1.0)	96.66 (1.3)
0.05	NAILS	98.45 (0.7)	94.53 (1.2)
	NAILM	98.13 (0.3)	94.73 (1.9)
0.20	NAILS	86.03 (4.1)	83.32 (5.5)
	NAILM	86.33 (1.5)	85.52 (1.7)

Table 5 shows the final accuracy (%) achieved on training and test data for different levels σ_n of noise, averaged over 20 runs from different initial values of the model parameters. The average CPU time to converge is 5.2 s (NAILS) and 6.2 s (NAILM).

6. Conclusions

We have proposed a training algorithm for RNNs that is computationally efficient, provides very good quality solutions, and can handle rather general loss and regularization terms. Although we assumed that (1) represents a recurrent neural network model in state-space form, the algorithm is applicable to learn parametric models with rather arbitrary structures, including other black-box state-space models (such as LSTM models), gray-box (physics-informed) and white-box models, and static models.

Future research will be devoted to further analyzing the convergence properties of the proposed algorithms within the Generalized Gauss–Newton framework, and to establish conditions on the ADMM parameter ρ, number of epochs E used when solving (19a), the model structure to learn, and the regularization function g that guarantee convergence of the nonconvex ADMM iterations.

Acknowledgments

The author thanks M. Schoukens for exchanging ideas on the setup of the Silverbox benchmark problem.

References

Andersson, J., Gillis, J., Horn, G., Rawlings, J., & Diehl, M. (2019). CasADi – a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, *11*(1), 1–36.

Beintema, G., Toth, R., & Schoukens, M. (2021). Nonlinear state-space identification using deep encoder networks. *vol. 144*, In *Proc. machine learning research* (pp. 241–250).

Bemporad, A. (2022). Training recurrent neural networks by sequential least squares and the alternating direction method of multipliers. Uploaded on v3.

Bemporad, A. (2023). Recurrent neural network training with convex loss and regularization functions by extended Kalman filtering. *IEEE Transactions on Automatic Control*, in press.

Bemporad, A., & Cimini, G. (2023). Variable elimination in model predictive control based on K-SVD and QR factorization. *IEEE Transactions on Automatic Control*, *68*(2), 782–797.

Burke, J., Curtis, F., Lewis, A., Overton, M., & Simões, L. (2020). Gradient sampling methods for nonsmooth optimization. In A. B. et al. (Ed.), *Numerical nonsmooth optimization* (pp. 201–225). http://www.cs.nyu.edu/overton/software/hanso/.

Curtis, F., Mitchell, T., & Overton, M. (2017). A BFGS-SQP method for nonsmooth, nonconvex, constrained optimization and its evaluation using relative minimization profiles. *Optimization Methods & Software*, *32*(1), 148–181, http://www.timmitchell.com/software/GRANSO/.

Dubey, S., Chakraborty, S., Roy, S., Mukherjee, S., Singh, S., & Chaudhuri, B. (2019). diffGrad: an optimization method for convolutional neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, *31*(11), 4500–4511.
Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proc. 13th int. conf. artificial intelligence and statistics* (pp. 249–256).
Hong, M., Luo, Z.-Q., & Razaviyayn, M. (2016). Convergence analysis of alternating direction method of multipliers for a family of nonconvex problems. *SIAM Journal on Optimization*, *26*(1), 337–364.
Houska, B., Frasch, J., & Diehl, M. (2016). An augmented Lagrangian based algorithm for distributed nonconvex optimization. *SIAM Journal on Optimization*, *26*(2), 1101–1127.
Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.
Kolter, J., & Manek, G. (2019). Learning stable deep dynamics models. *Advances in Neural Information Processing Systems*, *32*.
Levenberg, K. (1944). A method for the solution of certain non-linear problems in least squares. *Q. Appl. Math.*, *2*(2), 164–168.
Li, Z., Ding, C., Wang, S., Wen, W., Zhuo, Y., Kiu, C., et al. (2019). E-RNN: Design optimization for efficient recurrent neural networks in FPGAs. In *IEEE int. symp. high performance computer architecture* (pp. 69–80).
Ljung, L. (2001). *System identification toolbox for MATLAB*. The Mathworks, Inc., https://www.mathworks.com/help/ident.
Ljung, L., Andersson, C., Tiels, K., & Schön, T. (2020). Deep learning and system identification. *IFAC-PapersOnLine*, *53*(2), 1175–1181.
Ljung, L., Zhang, Q., Lindskog, P., & Juditski, A. (2004). Estimation of grey box and black box models for non-linear circuit data. *IFAC Proceedings Volumes*, *37*(13), 399–404.
Marconato, A., Schoukens, M., Rolain, Y., & Schoukens, J. (2013). Study of the effective number of parameters in nonlinear identification benchmarks. In *Proc. 52nd IEEE conf. dec. contr.* (pp. 4308–4313).
Marquardt, D. (1963). An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, *11*(2), 431–441.
Masti, D., & Bemporad, A. (2021). Learning nonlinear state-space models using autoencoders. *Automatica*, *129*, Article 109666.
Messerer, F., Baumgärtner, K., & Diehl, M. (2021). Survey of sequential convex programming and generalized Gauss-Newton methods. *ESAIM Proceedings and Surveys*, *71*, 64.
Narendra, K., & Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, *1*(1), 4–27.
Nocedal, J., & Wright, S. (2006). *Numerical optimization* (2). Springer.
Puskorius, G., & Feldkamp, L. (1994). Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks. *IEEE Transactions on Neural Networks*, *5*(2), 279–297.
Reddi, S., Kale, S., & Kumar, S. (2019). On the convergence of adam and beyond. arXiv preprint arXiv:1904.09237.
Ribeiro, A., Tiels, K., Umenberger, J., Schön, T., & Aguirre, L. (2020). On the smoothness of nonlinear system identification. *Automatica*, *121*, Article 109158.
Takapoui, R., Moehle, N., Boyd, S., & Bemporad, A. A simple effective heuristic for embedded mixed-integer quadratic programming 93 (1) (2020) 2–12.
Tang, Y., Kan, Z., Sun, D., Qiao, L., Xiao, K., Lai, Z., et al. (2020). ADMMiRNN: Training RNN with stable convergence via an efficient ADMM approach. In *Joint European conference on machine learning and knowledge discovery in databases* (pp. 3–18).
Taylor, G., Burmeister, R., Xu, Z., Singh, B., Patel, A., & Goldstein, T. (2016). Training neural networks without gradients: A scalable ADMM approach. In *International conference on machine learning* (pp. 2722–2731).
Themelis, A., & Patrinos, P. (2020). Douglas–Rachford splitting and ADMM for nonconvex optimization: Tight convergence results. *SIAM Journal on Optimization*, *30*(1), 149–181.
Transtrum, M., & Sethna, J. (2012). Improvements to the Levenberg-Marquardt algorithm for nonlinear least-squares minimization. arXiv preprint arXiv:1201.5885.
Vaz, A., & Vicente, L. (2009). PSwarm: A hybrid solver for linearly constrained global derivative-free optimization. *Optimization Methods & Software*, *24*, 669–685, http://www.norg.uminho.pt/aivaz/pswarm/.
Wang, J., Sano, A., Chen, T., & Huang, B. (2009). Identification of Hammerstein systems without explicit parameterisation of non-linearity. *International Journal of Control*, *82*(5), 937–952.
Wang, Y., Yin, W., & Zeng, J. (2019). Global convergence of ADMM in nonconvex nonsmooth optimization. *Journal of Scientific Computing*, *78*(1), 29–63.
Werbos, P. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560.
Wigren, T., & Schoukens, J. (2013). Three free data sets for development and benchmarking in nonlinear system identification. In *European control conference* (pp. 2933–2938). Zürich, Switzerland.
Williams, R., & Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, *2*(4), 490–501.
Xu, Q., Krishnamurthy, K., McMillin, B., & Lu, W. (1994). A recursive least squares training algorithm for multilayer recurrent neural networks. *2*, In *Proc. American control conference* (pp. 1712–1716).
Yuan, M., & Lin, Y. (2006). Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society. Series B. Statistical Methodology*, *68*(1), 49–67.

Alberto Bemporad received his Master's degree cum laude in Electrical Engineering in 1993 and his Ph.D. in Control Engineering in 1997 from the University of Florence, Italy. In 1996/97 he was with the Center for Robotics and Automation, Department of Systems Science & Mathematics, Washington University, St. Louis. In 1997–1999 he held a postdoctoral position at the Automatic Control Laboratory, ETH Zurich, Switzerland, where he collaborated as a Senior Researcher until 2002. In 1999–2009 he was with the Department of Information Engineering of the University of Siena, Italy, becoming an Associate Professor in 2005. In 2010–2011 he was with the Department of Mechanical and Structural Engineering of the University of Trento, Italy. Since 2011 he has been a Full Professor at the IMT School for Advanced Studies Lucca, Italy, where he served as the Director of the institute in 20122015. He spent visiting periods at Stanford University, University of Michigan, and Zhejiang University. In 2011 he co-founded ODYS S.r.l., a company specialized in developing model predictive control systems for industrial production. He has published more than 400 papers in the areas of model predictive control, hybrid systems, optimization, automotive control, and is the co-inventor of 21 patents. He is the author or coauthor of various software packages for model predictive control design and implementation, including the Model Predictive Control Toolbox (The Mathworks, Inc.) and the Hybrid Toolbox for MATLAB. He was an Associate Editor of the IEEE Transactions on Automatic Control during 2001–2004 and Chair of the Technical Committee on Hybrid Systems of the IEEE Control Systems Society in 2002–2010. He received the IFAC High-Impact Paper Award for the 2011–14 triennial, the IEEE CSS Transition to Practice Award in 2019, and the 2021 SAE Environmental Excellence in Transportation Award. He has been an IEEE Fellow since 2010.