Training Recurrent Neural Networks by Sequential Least Squares and the Alternating Direction Method of Multipliers

Alberto Bemporad*

October 18, 2022

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 \( \ell_1 \), \( \ell_0 \), 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.

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.

1 Introduction

In the last few years we have witnessed a tremendous attention towards machine learning methods in several fields, including control. The use of neural networks (NNs) for modeling dynamical systems, already popular in the nineties [10, 28], is flourishing again, mainly due to the wide availability of excellent open-source packages for automatic differentiation (AD). AD greatly simplifies setting up and solving the training problem via numerical optimization, in which the unknowns are the weights and bias terms of the NN. While the attention has been mainly on the use of feedforward NNs to 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 the fact that they are intrinsically state-space models.

While minimizing the one-step-ahead output prediction error of the NNARX model leads to a (nonconvex) unconstrained optimization problem with a separable objective...
function, training RNNs is more difficult. This is because the hidden states and parameters of the network are coupled by nonlinear equality constraints. The training problem is usually solved by gradient descent methods in which backpropagation through time (or its approximate truncated version [40]) is used to evaluate derivatives. To enable the use of minibatch stochastic gradient descent (SGD) methods, in [3] the authors propose to split the experiment in multiple sections where the initial state of each section is written as a NN function of a finite set of past inputs and outputs.

The main issue of gradient-descent methods is their well-known slow-rate of convergence. To improve speed and quality of training, as well as 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) [15, 25, 36] or recursive least squares (RLS) [41] were proposed based on minimizing the mean-squared error (MSE) loss. These results were recently extended in [4] to handle rather arbitrary convex and twice-differentiable loss terms when training RNNs whose state-update and output functions are multi-layer NNs, showing a drastic improvement in terms of computation efficiency and quality of the trained model with respect to gradient-descent methods. For the special case of shallow RNNs in which the states are known functions of the output data and MSE loss, a combination of gradient descent and recursive least squares was proposed in [24] to optimize the sequence of outputs and weights, respectively.

In this paper we propose an offline training method based on two main contributions. First, to handle arbitrary convex and twice-differentiable loss terms, we propose the use of 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 [38] is that the required Jacobian matrices, whose evaluation is usually the most expensive part of the training algorithm, can be computed in parallel, rather than sequentially, 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 [13, 20, 33].

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 in the past for training NNs, mainly feedforward NNs [31] but also recurrent ones [30]. However, existing methods entirely rely on ADMM to solve the learning problem, such as to gain efficiency by parallelizing numerical operations. In this paper, ADMM is combined with the smooth nonlinear optimizer based on sequential LS mentioned above to handle non-smooth terms, therefore allowing handling sparsification (via $\ell_1$, $\ell_0$, or group-Lasso penalties) and to impose (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 the identification of control-oriented black-box models. See also [14] for the use of 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, provides better-quality solutions than classical gradient-descent methods, and is competitive with respect to EKF-based approaches. The algorithm is also applicable to
the simpler case of training feedforward neural networks.

The paper is organized as follows. After formulating the training problem in Section 2, we present the sequential least-squares approach to solve 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 is shown in three nonlinear identification problems in Section 5.

## 2 Problem formulation

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

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

where \(k\) denotes the sample instant, \(x_k \in \mathbb{R}^{n_x}\) is the vector of hidden states, \(f_x : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \rightarrow \mathbb{R}^{n_x}\) and \(f_y : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \rightarrow \mathbb{R}^{n_y}\) are 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, as follows [4]:

\[
\begin{align*}
  v_{1k}^x &= A_1^x [v_{1k}^x] + b_1^x \\
  v_{2k}^x &= A_2^x f_1^x (v_{1k}^x) + b_2^x \\
  &\vdots \\
  v_{L_x k}^x &= A_{L_x}^x f_{L_x-1}^x (v_{(L_x-1)k}^x) + b_{L_x}^x \\
  x_{k+1} &= v_{L_x k}^x \\

  v_{1k}^y &= A_1^y [v_{1k}^y] + b_1^y \\
  v_{2k}^y &= A_2^y f_1^y (v_{1k}^y) + b_2^y \\
  &\vdots \\
  v_{L_y k}^y &= A_{L_y}^y f_{L_y-1}^y (v_{(L_y-1)k}^y) + b_{L_y}^y \\
  \hat{y}_k &= f_y^y (v_{L_y k}^y) 
\end{align*}
\]  

(2a) (2b)

where \(L_x - 1 \geq 0\) and \(L_y - 1 \geq 0\) are the number of hidden layers of the state-update and output functions, respectively, \(v_{i}^y \in \mathbb{R}^{n_{v_i}}, i = 1, \ldots L_x - 1, v_{L_x}^y \in \mathbb{R}^{n_{v_{L_x}}}, \) and \(v_{i}^x \in \mathbb{R}^{n_{v_i}}, i = 1, \ldots L_y - 1, \) the values of the corresponding inner layers, \(f_{i}^x : \mathbb{R}^{n_{v_i}} \rightarrow \mathbb{R}^{n_{v_{i+1}}}, i = 1, \ldots L_x - 1, \) \(f_{i}^y : \mathbb{R}^{n_{v_i}} \rightarrow \mathbb{R}^{n_{v_{i+1}}}, i = 1, \ldots L_y - 1\) the corresponding activation functions, and \(f_y^y : \mathbb{R}^{n_{v_{L_y}}} \rightarrow \mathbb{R}^{n_y}\) the output function, e.g., \(f_y^y (v_{L_y}^y) = v_{L_y}^y\) to model numerical outputs, or \([f_y^y (v_{L_y}^y)]_i = 1/(1 + e^{v_{L_y}^y}_i)\), \(i = 1, \ldots, n_y\), for binary outputs. Note that the RNN [4] can be made strictly causal by simply setting \(v_1^y (k) = A_1^y x_k + b_1^y\) in (2b).

The training problem we want to solve is:

\[
\begin{align*}
  &\min_{z} \quad r(x_0, \theta_x, \theta_y) + g(\theta_x, \theta_y) + \sum_{k=0}^{N-1} \ell(y_k, f_y(x_k, u_k, \theta_y)) \\
  \text{s.t.} \quad &x_{k+1} = f(x_k, u_k, \theta_x), \quad k = 0, \ldots, N - 2
\end{align*}
\]  

(3a) (3b)

where \(z = [x'_0 \ldots x'_{N-1}, \theta'_x \theta'_y] \in \mathbb{R}^{n}\) is the overall optimization vector, \(n = N n_x + n_{\theta_x} + n_{\theta_y}\), \(\ell : \mathbb{R}^{n_y} \times \mathbb{R}^{n_y} \rightarrow \mathbb{R}\) is a loss function that we assume strongly convex and twice differentiable with respect to its second argument, \(r : \mathbb{R}^{n_x} \times \mathbb{R}^{n_{\theta_x}} \times \mathbb{R}^{n_{\theta_y}} \rightarrow \mathbb{R}\)
is a regularization term (such as an $\ell_2$-regularizer) that we also assume strongly convex and twice differentiable, and $g : \mathbb{R}^{n_{a}x} \times \mathbb{R}^{n_{o}x} \to \mathbb{R} \cup \{+\infty\}$ is a possibly non-smooth and non-convex regularization term. For simplicity, in the following we assume that $r$ is separable with respect to $x_{0}$, $\theta_{x}$, and $\theta_{y}$, i.e., $r(x_{0}, \theta_{x}, \theta_{y}) = r_{x}(x_{0}) + r_{\theta_{x}}^{\theta_{y}}(\theta_{x}) + r_{\theta_{y}}^{\theta_{y}}(\theta_{y})$, and that in case of multiple outputs $n_{y} > 1$, the loss function $\ell$ is also separable, i.e., $\ell(y_{\hat{i}}, \hat{y}_{i}) = \sum_{i=1}^{n_{y}} \ell_{i}(y_{ki}, f_{y}(x_{k}, u_{k}, \theta_{y}))$, where the subscript $i$ denotes the $i$th component of the output signal $y_{k}$ and the corresponding loss function.

Problem (3) can be rewritten as the following unconstrained nonlinear programming problem in condensed form

$$\min_{x_{0}, \theta_{x}, \theta_{y}} V(x_{0}, \theta_{x}, \theta_{y}) + g(\theta_{x}, \theta_{y})$$

(4)

where we get

$$V(x_{0}, \theta_{x}, \theta_{y}) = r(x_{0}, \theta_{x}, \theta_{y}) + \sum_{k=0}^{N-1} \ell(y_{k}, f_{y}(x_{k}, u_{k}, \theta_{y}))$$

(5)

by replacing $x_{k+1} = f_{x}(x_{k}, u_{k}, \theta_{x})$ iteratively. Note that Problem (4) can be highly nonconvex and present several local minima, see, e.g., recent studies reported in [27].

2.1 Multiple training traces and initial-state encoder

For simplicity of notation, in this paper we focus on training a RNN model (2) 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 $(x_{0,1}, \ldots, x_{0,M}, \theta_{x}, \theta_{y})$. In alternative, we can parameterize

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

(6)

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 $n_{p}$ past inputs as suggested in [3, 21], 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 $f_{x0} : \mathbb{R}^{n_{v}} \times \mathbb{R}^{n_{a}o} \to \mathbb{R}^{n_{x}}$ is a feedforward neural network defined as in (2), parameterized by the vector $\theta_{x0} \in \mathbb{R}^{n_{a}o}$ to be learned jointly with $\theta_{x}$, $\theta_{y}$. The regularization term in this case is defined as $r(\theta_{x0}, \theta_{x}, \theta_{y})$.

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 $x_{k}^{h}$, $k = 1, \ldots, N-1$, and predicted outputs $\hat{y}_{k}^{h}$. 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 $x_{k+1}^{h+1}, \theta_{x}^{h+1}, \theta_{y}^{h+1}$ by solving a least-squares approximation of problem (3) as described in the next paragraphs.

3.1 Linearization of the dynamic constraints

To get a quadratic approximation of problem (4) we first linearize the RNN dynamic constraints (3b) around $x_{k}^{h}$ by approximating $x_{k+1}^{h} + p_{xk+1} = f_{x}(x_{k}^{h} + p_{x}, u_{k}, \theta_{x}^{h} + p_{\theta_{x}}) \approx
\begin{align*}
f_x(x_k^h, u_k, \theta_k^h) + \nabla_x f_x(x_k^h, u_k, \theta_k^h) p_{x_k} + \nabla_{\theta_x} f_x(x_k^h, u_k, \theta_k^h) p_{\theta_x}.
\end{align*}

Since \( x_{k+1}^h = f_x(x_k^h, u_k, \theta_x^h) \), we can write the linearized state-update equations as
\begin{align}
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}.
\end{align}

The dynamics (7) can be condensed to
\begin{align}
p_{x_k} = M_{kx} p_{x_0} + M_{k\theta_x} p_{\theta_x}.
\end{align}

where matrices \( M_{kx} \in \mathbb{R}^{n_x \times n_x} \) and \( M_{k\theta_x} \in \mathbb{R}^{n_x \times n_{\theta_x}}, \ k = 0, \ldots, N - 2, \) are recursively defined as follows:
\begin{align*}
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{align*}

In case \( x_0 \) is parameterized as in (6), by approximating
\begin{align*}
x_0^h \approx f_{x_0}(v_0, \theta_{x_0}^h) + \nabla_{\theta_x} f_{x_0}(v_0, \theta_{x_0}^h) p_{\theta_x},
\end{align*}

where \( p_{\theta_x} \) is the increment of \( \theta_{x_0} \) to be computed, it is enough to replace \( p_{x_0} \) with \( p_{\theta_x} \) in (8) and \( M_{0x} = \nabla_{\theta_x} f_{x_0}(v_0, \theta_{x_0}^h) \) in (9).

### 3.2 Quadratic approximation of the cost function

Let us now find a quadratic approximation of the cost function \( f \). First, regarding the regularization term \( r \), we take the 2nd-order Taylor expansions
\begin{align}
r_x(x_0 + p_{x_0}) &\approx \frac{1}{2} p_{x_0}^T \nabla^2 r_x(x_0) p_{x_0} + \nabla r_x(x_0) p_{x_0} + r(x_0) \\
r_{\theta_x}(\theta_{x_0} + p_{\theta_{x_0}}) &\approx \frac{1}{2} p_{\theta_{x_0}}^T \nabla^2 r_{\theta_x}(\theta_{x_0}) p_{\theta_{x_0}} + \nabla r_{\theta_x}(\theta_{x_0}) p_{\theta_{x_0}} + r_{\theta_x}(\theta_{x_0}) \\
r_{\theta_y}(\theta_{y_0} + p_{\theta_{y_0}}) &\approx \frac{1}{2} p_{\theta_{y_0}}^T \nabla^2 r_{\theta_y}(\theta_{y_0}) p_{\theta_{y_0}} + \nabla r_{\theta_y}(\theta_{y_0}) p_{\theta_{y_0}} + r_{\theta_y}(\theta_{y_0})
\end{align}

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
\begin{align}
\frac{1}{2} \left\| \begin{bmatrix} L_{r_x} p_{x_0} + (L_{r_x}')^{-1} \nabla r_x(x_0) \\ L_{r_{\theta_x}} p_{\theta_x} + (L_{r_{\theta_x}}')^{-1} \nabla r_{\theta_x}(\theta_{x_0}) \\ L_{r_{\theta_y}} p_{\theta_y} + (L_{r_{\theta_y}}')^{-1} \nabla r_{\theta_y}(\theta_{y_0}) \end{bmatrix} \right\|^2
\end{align}

where \( L_{r_x} L_{r_{\theta_x}} = \nabla^2 r_x(\theta_x) \) and \( L_{r_{\theta_y}} L_{r_{\theta_y}} = \nabla^2 r_{\theta_y}(\theta_y) \) are Cholesky factorizations of the corresponding Hessian matrices. Note that in the case of standard \( \ell_2 \)-regularization \( r_x(x_0) = \frac{\theta}{2} \| x_0 \|^2, \ r_{\theta_x}(\theta_{x_0}) = \frac{\theta}{2} \| \theta_x \|^2, \ r_{\theta_y}(\theta_{y_0}) = \frac{\theta}{2} \| \theta_y \|^2 \), we simply have \( L_{r_x} = \sqrt{\theta} I, \ L_{r_{\theta_x}} = \sqrt{\theta} I, \ L_{r_{\theta_y}} = \sqrt{\theta} I \), \( \nabla_x r_x(x_0) = \sqrt{\theta} x_0, \ r_{\theta_x}(\theta_{x_0}) = \sqrt{\theta} \theta_x, \ r_{\theta_y}(\theta_{y_0}) = \sqrt{\theta} \theta_y \), and hence (10) becomes
\begin{align}
\frac{1}{2} \left\| \begin{bmatrix} \sqrt{\theta} p_{x_0} + \sqrt{\theta} x_0 \\ \sqrt{\theta} p_{\theta_x} + \sqrt{\theta} \theta_x \\ \sqrt{\theta} p_{\theta_y} + \sqrt{\theta} \theta_y \end{bmatrix} \right\|^2
\end{align}
Regarding the loss terms penalizing the $i$th output-prediction error at step $k$, we have that

\[
\begin{align*}
\nabla_x \ell_i(y_{k i}, f_{y i}(x^h_k, u_k, \theta^h_y)) &= \nabla_x f_{y i}(x^h_k, u_k, \theta^h_y) \ell'_{ik} \\
\nabla_{\theta_y} \ell_i(y_{k i}, f_{y i}(x^h_k, u_k, \theta^h_y)) &= \nabla_{\theta_y} f_{y i}(x^h_k, u_k, \theta^h_y) \ell'_{ik}
\end{align*}
\]

where $\ell'_{ik} \triangleq \frac{d\ell_i(y_{k i}, f_{y i}(x^h_k, u_k, \theta^h_y))}{dy_{k i}}$, $\ell'_{ik} \in \mathbb{R}$, and hence, by differentiating again with respect to $[\theta^h_y]$ and setting $\ell''_{ik} \triangleq \frac{d^2\ell_i(y_{k i}, f_{y i}(x^h_k, u_k, \theta^h_y))}{dy_{k i}^2}$, $\ell''_{ik} \in \mathbb{R}$, $\ell''_{ik} \geq 0$, we get

\[
\nabla_{[\theta^h_y]}^2 \ell_i(y_{k i}, f_{y i}(x^h_k, u_k, \theta^h_y)) = \left( \nabla_{[\theta^h_y]} f_{y i}(x^h_k, u_k, \theta^h_y) \right)' \ell''_{ik} + \nabla_{[\theta^h_y]} f_{y i}(x^h_k, u_k, \theta^h_y) \ell''_{ik}
\]

(we have omitted the dependence of $\ell'$, $\ell''$ on the iteration $h$ for simplicity). By neglecting the Hessian of the output function $f_{y i}$ (as it is common in Gauss-Newton methods), the minimization of $\ell_i(y_{k i}, f_{y i}(x^h_k + p_{x_k}, u_k, \theta^h_y + p_{\theta_y}))$ can be approximated by the minimization of

\[
\frac{1}{2} \left[ \ell''_{ik} \right] \frac{1}{2} \left[ \nabla_{[\theta^h_y]} f_{y i}(x^h_k + p_{x_k}, u_k, \theta^h_y + p_{\theta_y}) \right]' \ell''_{ik} + \ell''_{ik} \frac{1}{2} \left[ \nabla_{[\theta^h_y]} f_{y i}(x^h_k, u_k, \theta^h_y) \right]' \ell''_{ik}
\]

By strong convexity of the loss function $\ell$, this can be further rewritten as the minimization of the least-squares term

\[
\frac{1}{2} \left\| \ell''_{ik} \right\|^2 \left( \nabla_{[\theta^h_y]} f_{y i}(x^h_k, u_k, \theta^h_y) \right)' \ell''_{ik} + \ell''_{ik} \left( \nabla_{[\theta^h_y]} f_{y i}(x^h_k, u_k, \theta^h_y) \right)' \ell''_{ik} \left( \nabla_{[\theta^h_y]} f_{y i}(x^h_k, u_k, \theta^h_y) \right)' \ell''_{ik}
\]

that, by exploiting (8), is equivalent to

\[
\frac{1}{2} \left\| (\ell''_{ik})^2 \left[ \nabla_{x_k} \ell'_{ik}(y_{k i}, f_{y i}(x^h_k, u_k, \theta^h_y)) \right] \right\|^2_2
\]

Finally, the minimization of the quadratic approximation of (5) can be recast as the linear LS problem

\[
p^h = \arg \min_p \frac{1}{2} \| A^h p - b^h \|^2_2
\]

where $p = \begin{bmatrix} p_{x_k} \\ p_{x_k} \\ p_{\theta_y} \end{bmatrix}$, $A^h$ and $b^h$ are constructed by collecting the least-squares terms from (11) and (13), $A^h \in \mathbb{R}^{n_A \times n_p}$, $b^h \in \mathbb{R}^{n_A}$, $n_A = N \theta + n_x + n_{\theta_x} + n_{\theta_y}$, and $n_p = n_x + n_{\theta_x} + n_{\theta_y}$.

We remark that most of the computation effort is usually spent in computing the Jacobian matrices $\nabla_{x_k} f_{y i}(x_k^h, u_k, \theta^h_y)$ and $\nabla_{[\theta^h_y]} f_{y i}(x_k^h, u_k, \theta^h_y)$. 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 [38] to solve for $p^h$ in (14), as typically done in classical training methods based on gradient-descent.

Note also that for training feedforward neural networks we set $n_x = 0$, $v^y_i(k) = A^y_i u_k + b^y_i$ in (22), and only optimize with respect to $\theta_y$. In this case, the operations performed at each iteration $h$ to predict $\hat{y}_k$ and compute $V^h$ can be also parallelized.

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

\[
p^h = -\nabla V(x_k^h, \theta^h_x, \theta^h_y) = \begin{bmatrix} \nabla_{x_k} r_{x_k}(x_k^h) \\ \nabla_{\theta_x} r_{\theta_x}(\theta^h_x) \\ \nabla_{\theta_y} r_{\theta_y}(\theta^h_y) \end{bmatrix} + \sum_{k=0}^{N-1} \sum_{i=1}^{n_y} \ell'_{ik} \begin{bmatrix} \nabla_{x_k} f_{y i}(x_k^h, \theta^h_y) \end{bmatrix}' M_{x_k} + \sum_{k=0}^{N-1} \sum_{i=1}^{n_y} \ell'_{ik} \begin{bmatrix} \nabla_{\theta_y} f_{y i}(x_k^h, \theta^h_y) \end{bmatrix}' M_{\theta_y}
\]

since, from (9), $\nabla_{x_k} x_k = M_{x_k}$ and $\nabla_{\theta_x} x_k = M_{\theta_x}$. 

\[
6
\]
3.2.1 Sequential unconstrained quadratic programming

The formulation (14) requires storing matrix \( A^h \) and vector \( b^h \). In order to decrease memory requirements, especially in the case of a large number \( N \) of training data, an alternative method is to incrementally construct the matrices \( H^h = (A^h)'(A^h) \in \mathbb{R}^{n_p \times n_p} \) and \( c^h = -(A^h)'b^h \in \mathbb{R}^{n_p} \) of the normal equation \( H^hp^h + c^h = 0 \) associated with (14). Although this approach might be computationally more expensive, it only requires storing \( n_p^2 + 3n_p \) numbers. A further advantage of solving the normal equation is that the convexity of \( \ell \) with respect to \( \hat{y} \) and of \( r(x, \theta_x, \theta_y) \) do not have to be strong, as instead required to express the quadratic approximation of \( f \) as a squared norm from (13) and (10), respectively, as long as the computed \( H^h \) is nonsingular.

3.2.2 Recursive least-squares

A further alternative to precomputing and storing the matrices \( A^h, b^h \) is to exploit (13) and use the following recursive least-square (RLS) formulation of (14)

\[
\begin{align*}
P_k &= P_{k-1} - \frac{P_{k-1}\phi_k P_{k-1}}{1 + \phi_k P_{k-1}\phi_k} \\
p_k &= p_{k-1} + P_k \phi_k(\eta_k - \eta_k' P_k \phi_k)
\end{align*}
\]

(16)

for \( k = 0, \ldots, N-1 \) where \( \phi_k = \sqrt{\ell_{ik}} \begin{bmatrix} M_{ix} & \nabla_x f_i \\ M_{ix} & \nabla_y f_i \end{bmatrix} \), \( \eta_k = \ell_{ik}'/\sqrt{\ell_{ik}}. \) The RLS iterations in (16) provide the optimal solution \( p^h \triangleq p_{N-1} \) (as well as \( P_{N-1} = ((A^h)'A^h)^{-1} \)) when the matrix \( P_k \in \mathbb{R}^{n_p \times n_p} \) and vector \( p_k \) are initialized, respectively, as

\[
P_{-1} = \begin{bmatrix} I_{n_x} & 0 & 0 \\ 0 & I_{n_{\theta_x}} & 0 \\ 0 & 0 & I_{n_{\theta_y}} \end{bmatrix}, \quad p_{-1} = \begin{bmatrix} x_0^h \\ \theta_x^h \\ \theta_y^h \end{bmatrix}
\]

(17)

Different algorithms were proposed in the literature to implement RLS efficiently, such as the inverse QR method proposed in [1], in which a triangular matrix \( R_k \) is updated recursively starting from \( R_{-1} = (P_{-1})^{1/2} \).

3.3 Line search

To enforce the decrease of the cost function (3a) across iterations \( h = 0, 1, \ldots \), a possibility is to update the solution as

\[
\begin{bmatrix} x_{k+1}^h \\ \theta_{x}^{k+1} \\ \theta_{y}^{k+1} \end{bmatrix} = \begin{bmatrix} x_k^h \\ \theta_x^h \\ \theta_y^h \end{bmatrix} + \alpha_h \begin{bmatrix} p_{x_0}^h \\ p_{\theta_x}^h \\ p_{\theta_y}^h \end{bmatrix}
\]

(18)

and set \( x_{k+1}^h = f(x_{k+1}^h, u_k, \theta_{x}^{k+1}), k = 1, \ldots, N-1 \). We choose the step-size \( \alpha_h \) by finding the largest \( \alpha \) satisfying the Armijo condition

\[
V^{h+1} \leq V^h + c_1 \alpha(\nabla V^h)'p^h
\]

(19)

where \( V^{h+1}, V^h \) are the costs defined as in (3) 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.,
$c_1 = 10^{-4}$ [23] Chapter 3.1]. In this paper we use the common approach of starting with $\alpha = 1$ and then change $\alpha \leftarrow \sigma \alpha$, $\sigma \in (0, 1)$ until (19) is satisfied. We set a bound $N_\sigma$ on the number of line-search steps that can be performed, setting $\alpha = 0$ in the case of line-search failure. Since $V(x_0 + \alpha p^h_0, \theta_x + \alpha p^h_{\theta_x}, \theta_y + \alpha p^h_{\theta_y}) \approx V(x_0, \theta^h_x, \theta^h_y) + \frac{1}{2} \|A^h \alpha p^h - b^h\|^2_2 - \frac{1}{2} \|b^h\|^2_2$ for small values of $\alpha$, the directional derivative in (19) can be computed from $A^h$, $b^h$:

$$(\nabla V^h) p^h = \lim_{\alpha \rightarrow 0} \frac{V(x_0 + \alpha p^h_0, \theta_x + \alpha p^h_{\theta_x}, \theta_y + \alpha p^h_{\theta_y}) - V(x_0, \theta^h_x, \theta^h_y)}{\alpha} = -(b^h)' A^h p^h$$

Note that when the loss function $\ell(y, \hat{y}) = \frac{1}{2} \|y - \hat{y}\|^2_2$ and the regularization terms $r_x, r_{\theta_y}, r_{\theta_z}$ are quadratic, the loss $V(x_0, \theta^h_x, \theta^h_y) = \frac{1}{2} (b^h)' b^h$ by construction.

The overall nonlinear least-squares solution method for training recurrent neural networks described in the previous sections is summarized in Algorithm 1.

Algorithm 1 belongs to the class of Generalized Gauss-Newton (GGN) methods [22] applied to solve problem (5), with the addition of line search. In fact, following the works described in the previous sections is summarized in Algorithm 1.

Algorithm 1 takes a second-order approximation of $\Phi_0$ and a linearization of $F(w)$ during the iterations. In [22], local convergence under the full step $x_h$ is shown under additional assumptions on $\Phi_0$ and the neglected second-order derivatives of $F$, while in Algorithm 1 we force monotonic decrease of the loss function $V$ at each iteration $h$ by line search, hence obtaining a “damped” GGN algorithm. Note that the latter may fail ($\alpha_h = 0$) in the case $p^h$ is not a descent direction or $N_\sigma$ is not large enough. In such a case, Algorithm 1 would stop immediately as $V^h - V^{h-1} = 0$.

3.4 Initialization

In this paper we initialize $x^0 = 0$, set zero initial bias terms, and draw the remaining components of $\theta_x^0$ and $\theta_y^0$ from the normal distribution with zero mean and standard deviation defined as in [7], further multiplied by the quantity $\sigma_0 \leq 1$. The rationale to select small initial values of $\theta_x^0$, $\theta_y^0$ is to avoid that the initial dynamics $f_x$ used to compute $x^k_0$ for $k = 1, \ldots, N - 1$ 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 $x_1, \ldots, x_{N-1}$ in the condensing approach (4) suitable for the current RNN training setting, contrarily to solving nonlinear MPC problems in which the initial state $x_0$ and the model coefficients $\theta_x$, $\theta_y$ are fixed and may therefore lead to excite unstable linearized model responses, with consequent numerical issues [5]. Note also that ideas to enforce open-loop stability could be introduced in our setting by adopting ideas as in [12], where the authors propose to also learn a Lyapunov function while learning the model.
Algorithm 1 Sequential least squares with line search.

**Input:** Training dataset \(\{u_k, y_k\}_{k=0}^{N-1}\), initial guess \((x_0^0, \theta_x^0, \theta_y^0)\), maximum number \(E\) of epochs, tolerance \(\epsilon_V > 0\), line-search parameters \(c_1 > 0, \sigma \in (0,1), N_\sigma \geq 1\).

1. Simulate (1) to get \(x_{k+1}^0, k = 0, \ldots, N - 1\), and the initial cost \(V_0^0\) from (5);
2. \(h \leftarrow 0\);
3. repeat:
   3.1. Construct and solve the linear least-squares problem (14) to get a descent direction \(p^h\);
   3.2. Find step-length \(\alpha_h \in [0,1]\) by line-search using Armijo rule (19) and get the new cost \(V_{h+1}^h\);
   3.3. Update \(x_{h+1}^0, \theta_x^{h+1}, \theta_y^{h+1}\) as in (18);
   3.4. \(h \leftarrow h + 1\);
4. until \(V^h - V^{h-1} \leq \epsilon_V\) or \(h \geq E\);
5. end.

**Output:** RNN parameters \(\theta_x^h, \theta_y^h\) and initial hidden state \(x_0^h\).

### 3.5 A Levenberg-Marquardt variant

Trust-region methods are an alternative family of approaches to line-search for forcing monotonicity of the objective function. In solving nonlinear least-squares problems, the well-known Levenberg-Marquardt (LM) algorithm [13, 20] is often used due to its good performance. In LM methods, (14) is changed to

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

where the regularization term \(\lambda_h\) is tuned at each iteration \(h\) to meet the condition \(V^h \leq V^{h-1}\). It is easy to show that tuning \(\lambda_h\) is equivalent to tuning the trust-region radius, see e.g. [23, Chapter 10]. Clearly, setting \(\lambda_h = 0\) corresponds to the full Generalized Gauss-Newton step, as in line-search when \(\alpha_h = 1\), and \(\lambda_h \rightarrow \infty\) corresponds to \(p^h \rightarrow 0\) \((\alpha_h \rightarrow 0)\). Several methods exist to update \(\lambda_h\) (and possibly generalizing \(\|p\|_2^2\) to \(\|D^hp\|_2^2\) for some weight matrix \(D^h\)) [33]. In this paper, we take the simplest approach suggested in [33, 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 \(h + 1\). The approach is summarized in Algorithm 2.

Similarly to Algorithm 1 in Algorithm 2 we have set a bound \(N_\lambda\) on the number of least-squares problems that can be solved at each epoch \(h\). In the case an improvement \(V^h \leq V^{h-1}\) has not been found, Algorithm 2 stops due to Step 3.6.

The main difference between performing line-search and computing instead LM iterations is that at each epoch \(h\) the former only solves the linear least-squares problems (14), the latter requires solving up to \(N_\lambda\) problems (20). Note that, however, both only construct \(A^h\) and \(b^h\) once at each iteration \(h\), which, as remarked earlier, is usually the dominant
Algorithm 2 Sequential least squares, Levenberg-Marquardt variant.

**Input:** Training dataset \( \{ u_k, y_k \}_{k=0}^{N-1} \), initial guess \( (x_0^0, \theta_x^0, \theta_y^0) \), maximum number \( E \) of epochs, tolerance \( \epsilon_V > 0 \), initial value \( \lambda_0 > 0 \), LM parameters \( c_2, c_3 > 1 \), number \( N_\lambda \geq 1 \) of search iterations.

1. Simulate \( 1 \), get \( x_{k+1}^0, k = 0, \ldots, N - 1 \), and initial cost \( V^0 \) from \( 5 \);
2. \( h \leftarrow 0; \)
3. repeat:
   3.1. Construct the least-squares problem \( 14 \);
   3.2. \( j \leftarrow 0; \)
   3.3. repeat:
      3.3.1. Solve the regularized linear least-squares problem \( 20 \) and get a candidate \( p_h \);
      3.3.2. Update \( x_{h+1}^0, \theta_{x}^{h+1}, \theta_{y}^{h+1} \) as in \( 18 \) when \( \alpha_h = 1; \)
      3.3.3. Get new cost \( V^{h+1} \);
      3.3.4. if \( V^{h+1} \leq V^h \) decrease \( \lambda_h \leftarrow \lambda_h/c_3; \)
      3.3.5. else increase \( \lambda_h \leftarrow c_2 \lambda_h; \)
      3.3.6. set \( j \leftarrow j + 1; \)
   3.4. until \( V^{h+1} \leq V^h \) or \( j = N_\lambda; \)
   3.5. \( h \leftarrow h + 1; \)
   3.6. if \( V^h > V^{h-1} \) go to Step \( 5 \)
4. until \( V^h - V^{h-1} \leq \epsilon_V \), or \( h \geq E; \)
5. end.

**Output:** RNN parameters \( \theta_{x}^h, \theta_{y}^h \) and initial hidden state \( x_0^h \).

Computation effort due to the evaluation of the required Jacobian matrices \( \nabla_x f_x, \nabla_{\theta_x} f_x, \nabla_x f_y, \nabla_{\theta_y} f_y \).

4 Non-smooth regularization

We now want to solve the full problem \( 3 \) by considering also the additional non-smooth regularization term \( g(\theta_x, \theta_y) \). Consider again the condensed loss function \( 5 \) and rewrite problem \( 3 \) according to the following split

\[
\begin{align*}
\min_{x_0, \theta_x, \theta_y, \nu_x, \nu_y} & \quad V(x_0, \theta_x, \theta_y) + g(\nu_x, \nu_y) \\
\text{s.t.} & \quad \theta_x = \nu_x, \quad \theta_y = \nu_y
\end{align*}
\] (21)
We solve problem (21) by executing the following scaled ADMM iterations

\[
\begin{align*}
\begin{bmatrix} x_0(t+1) \\ \theta_x(t+1) \\ \theta_y(t+1) \end{bmatrix} &= \arg\min_{x_0,\theta_x,\theta_y} V(x_0, \theta_x, \theta_y) + \frac{\rho}{2} \left\| \begin{bmatrix} \theta_x - \nu_x(t) + w_x(t) \\ \theta_y - \nu_y(t) + w_y(t) \end{bmatrix} \right\|^2_2 \\
\begin{bmatrix} \nu_x(t+1) \\ \nu_y(t+1) \end{bmatrix} &= \arg\min_{\nu_x,\nu_y} \frac{1}{\rho} g(\nu_x, \nu_y) + \frac{1}{2} \left\| \begin{bmatrix} \nu_x - \theta_x(t+1) - w_x(t) \\ \nu_y - \theta_y(t+1) - w_y(t) \end{bmatrix} \right\|^2_2 \\
\begin{bmatrix} w_x(t+1) \\ w_y(t+1) \end{bmatrix} &= \text{prox}_{\frac{1}{\rho} g}(\theta_x(t+1) + w_x(t), \theta_y(t+1) + w_y(t)) \\
\end{align*}
\tag{22a}
\end{align*}
\]

for \( t = 0, \ldots, N_{\text{ADMM}} - 1 \), where “prox” denotes the proximal operator. Problem (22a) is solved by running either Algorithm 1 or 2 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 (14) or (20), \( \ell_2 \)-regularization term \( \frac{1}{2} \left\| \sqrt{\rho} \begin{bmatrix} \theta_x^0 \\ \theta_y^0 \end{bmatrix} - \sqrt{\rho} \begin{bmatrix} \nu_x(t) - \theta_x(t) \\ \nu_y(t) - \theta_y(t) \end{bmatrix} \right\|^2_2 \) introduced in the ADMM iteration (22a). Note also that for \( t > 0 \) Step 1 of Algorithms 1 and 2 can be omitted, since the initial simulated trajectory \( x_{k+1}^0 \) is available from the last execution of the ADMM step (22a), \( \forall k = 0, \ldots, N - 2 \), and \( V_0 \) can be obtained by adjusting the cost computed at the last iteration of the previous execution of Algorithm 1 or 2 to take into account that \( \nu_x, \nu_y, w_x, w_y \) have possibly changed their value.

Our numerical experiments show that setting \( E = 1 \) is a good choice, which corresponds to just computing one sequential LS iteration to solve (22a) approximately. The overall algorithm, that we call NAILS (nonconvex ADMM iterations and least squares with line search) in the case Algorithm 1 is used, or NAILM (nonconvex ADMM iterations and least squares with Levenberg-Marquardt variant) when instead Algorithm 2 is used, is summarized in Algorithm 3. Note that NAILS/NAILM processes the training dataset for a maximum of \( E \cdot N_{\text{ADMM}} \) epochs.

**Algorithm 3** 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 \( x_0^0, \theta_x^0, \theta_y^0 \), number \( N_{\text{ADMM}} \) 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. \[ \begin{bmatrix} \nu_x^0 \\ \nu_y^0 \end{bmatrix} \leftarrow \begin{bmatrix} \theta_x^0 \\ \theta_y^0 \end{bmatrix}, \quad \begin{bmatrix} w_x^0 \\ w_y^0 \end{bmatrix} \leftarrow 0; \]
2. **for** \( t = 0, 1, \ldots, N_{\text{ADMM}} - 1 \) **do:**
   1. **run** Algorithm 1 or 2 to update \( x_0(t+1), \theta_x(t+1), \) and \( \theta_y(t+1) \) as in (22a);
   2. **update** \( \begin{bmatrix} \nu_x(t+1) \\ \nu_y(t+1) \end{bmatrix} \) as in (22b) and \( \begin{bmatrix} w_x(t+1) \\ w_y(t+1) \end{bmatrix} \) as in (22c);
   3. \( t \leftarrow t + 1; \)
3. **end.**

**Output:** RNN parameters \( \theta_x = \nu_x(t), \theta_y = \nu_y(t) \) and initial hidden state \( x_0(t) \).
In NAILS, if RLS are used in alternative to forming and solving (14), since
\[ \arg \min_{\alpha \in \mathbb{R}} \frac{1}{2} (\rho_1 (\alpha - \alpha_1)^2 + \rho_2 (\alpha - \alpha_2)^2) = \arg \min_{\alpha \in \mathbb{R}} \frac{\rho_1 + \rho_2}{2} \left( \alpha - \frac{\rho_1 \alpha_1 + \rho_2 \alpha_2}{\rho_1 + \rho_2} \right)^2 \]
for any \( \rho_1, \rho_2 > 0 \) and \( \alpha_1, \alpha_2 \in \mathbb{R} \), we can initialize
\[
P_{-1} = \begin{bmatrix} \frac{\rho_1}{\rho_1 + \rho_2} I_{n_x} & 0 & 0 \\ 0 & \frac{\rho_2}{\rho_1 + \rho_2} I_{n_y} & 0 \\ 0 & 0 & \frac{1}{\rho_1 + \rho_2} I_{n_y} \end{bmatrix}, \quad p_{-1} = \begin{bmatrix} x_0^h \\ \frac{\rho_2}{\rho_1 + \rho_2} (\nu_x(t) + \theta x(t) - \omega_x(t)) \\ \frac{\rho_2}{\rho_1 + \rho_2} (\nu_y(t) + \theta y(t) - \omega_y(t)) \end{bmatrix}
\]
The final model parameters are given by \( \theta x = \nu_x(t), \theta y = \nu_y(t), x_0(t) \), where \( t = N_{ADMM} \) 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 \( \nu x_0 \).

Algorithm 3 can be modified to return the initial state and model obtained after each ADMM step that provide the best loss \( \sum \ell(y_k, \hat{y}_k) \) observed during the execution of the algorithm on training data or, if available, on a separate validation dataset.

Note that the iterations (22) 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 [9]. The reader is referred to [8, 9, 32, 37] for convergence conditions and/or alternative ADMM formulations.

We finally remark that, in the absence of the non-smooth regularization term \( g \), Algorithm 3 simply reduces to a smooth GNN method running with tolerance \( \epsilon V \) for a maximum number \( E \) of epochs.

4.1 \( \ell_1 \)-regularization

A typical instance of regularization is the \( \ell_1 \)-penalty
\[
g(\theta x, \theta y) = \tau x \| \theta x \|_1 + \tau y \| \theta y \|_1
\]
to attempt pruning an overly-parameterized network structure, with \( \tau x, \tau y \geq 0 \). In this case, the update in (22b) simply becomes
\[
\nu x(t + 1) = S_{\tau x} \left( \theta x(t + 1) + w x(t) \right) \\
\nu y(t + 1) = S_{\tau y} \left( \theta y(t + 1) + w y(t) \right)
\]
where \( S_\alpha \) is the soft-threshold operator
\[
[S_\alpha(w)]_i = \begin{cases} 
  w_i + \alpha & \text{if } w_i \leq -\alpha \\
  0 & \text{if } |w_i| \leq \alpha \\
  w_i - \alpha & \text{if } w_i \geq \alpha 
\end{cases}
\]

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 [42] that attempt zeroing entire subsets of parameters. In particular, the order \( n x \) of the RNN can be penalized by creating \( n x \) groups \( \theta^{i x} y \) of variables, \( \theta^{i x} y \in \mathbb{R}^{n x}, n g = n x^i + n y^i + n_{x-1} x^i + 1, i = 1, \ldots, n x, \) each one stacking the \( i \)-th columns of \( A^{i x}_t \) and \( A^{i y}_t \), the \( i \)-th
row of $A_{Lx}$, and the $i$th entry of $b_{Lx}$. In this case, in (21) we only introduce the splitting $	heta_i^g = \nu_i^g$ on those variables and penalize

$$g(\nu_i^g) = \tau_g \sum_{i=1}^{n_x} \|\nu_i^g\|_2$$

where $\tau_g \geq 0$ is the group-Lasso penalty. Then, vectors $\nu_i^g(t + 1)$ are updated as in (24) by replacing $S_{\tau_g}$ with the block soft thresholding operator $S_{\frac{\tau_g}{\rho}} : \mathbb{R}^{n_g} \rightarrow \mathbb{R}^{n_g}$ defined as

$$S_{\frac{\tau_g}{\rho}}(w) = \begin{cases} (1 - \frac{\alpha}{\|w\|_2})w & \text{if } \|w\|_2 > \alpha \\ 0 & \text{if } \|w\|_2 \leq \alpha \end{cases}$$

The larger the penalty $\tau_g$, the larger is expected the number of state indices $i$ such that the $i$th column of $A_{Lx}$, the $i$th row of $A_{Lx}$, and the $i$th entry of $b_{Lx}$ are all zero. As a consequence, the corresponding $i$th hidden state has no influence in the model.

### 4.3 $\ell_0$-regularization

The $\ell_0$-regularization term

$$g(\theta_x, \theta_y) = \tau_x \|\theta_x\|_0 + \tau_y \|\theta_y\|_0$$

with $\tau_x^0, \tau_y^0 \geq 0$ also admits the explicit proximal “hard-thresholding” operator

$$[\text{prox}_{\frac{\tau_g}{\rho}}\|x\|_0]_i = \begin{cases} 0 & \text{if } x_i^2 < \frac{2\rho}{\tau} \\ x_i & \text{if } x_i^2 \geq \frac{2\rho}{\tau} \end{cases}$$

and can be used in alternative to (23).

### 4.4 Integer constraints

Constraints on $\theta_x, \theta_y$ that can expressed as mixed-integer linear inequalities

$$A_{MI} \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} \leq b_{MI}, \quad \theta_x, \theta_y \in \{0, 1\}, \quad \forall i \in I_x, j \in I_y$$

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

$$g(\theta_x, \theta_y) = \begin{cases} 0 & \text{if } (26) \text{ is satisfied} \\ +\infty & \text{otherwise} \end{cases}$$

In this case, $\nu_{x}^{h+1}, \nu_{y}^{h+1}$ can be computed from (22b) by solving the mixed-integer quadratic programming (MIQP) problem

$$\min_{\nu_x, \nu_g} \frac{1}{2} \left\| \begin{bmatrix} \nu_x - \theta_x(t+1) - w_x(t) \\ \nu_y - \theta_y(t+1) - w_y(t) \end{bmatrix} \right\|_2^2$$

s.t. $A_{MI} \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} \leq b_{MI}$

$$\theta_x, \theta_y \in \{0, 1\} \quad \forall i \in I_x, j \in I_y$$

Note that in the special case $n_{MI} = 0$, (27) has the following explicit solution

$$\nu_{xi}(t + 1) = \text{round}(\theta_{xi}(t + 1) + w_{xi}(t)), \quad i \in I_x$$

$$\nu_{yj}(t + 1) = \text{round}(\theta_{yj}(t + 1) + w_{yj}(t)), \quad j \in I_y$$

13
Figure 1: MSE loss (5) evaluated on training data as a function of training time on different runs

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

Binary constraints can be extended to more general quantization constraints \(\theta_{x_i}, \theta_{y_j} \in Q \triangleq \{q_1, \ldots, q_{n_Q}\} \subset \mathbb{R}, \forall i \in I_x, \forall j \in I_y\). In fact, these can be handled as in (28) by setting \(\nu_{x_i}(t + 1)\) equal to the value in \(Q\) that is closest to \(\theta_{x_i}(t + 1) + w_{x_i}^h\), and similarly for \(\nu_{y_j}(t + 1)\).

5 Numerical experiments

We test NAILS and NAILM on three nonlinear system identification problems. The hyperparameters of the Algorithm 3 are summarized in Table 1. All computations have been carried out in MATLABR2022b on an Apple M1 Max CPU, with the library CasADi [2] used to generate 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 (6) 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

\[
\min_{x_0} \rho_x \|x_0\|_2^2 + \sum_{k=0}^{N_0} \ell(y_k, \hat{y}_k) \tag{29}
\]

by using the particle swarm optimizer (PSO) PSwarm [34] with initial population of 2\(n_x\) samples, each component of \(x_0\) constrained in \([-3, 3]\), and with \(N_0 = 100\).

5.1 RNN training with smooth quadratic loss

We first test NAILS/NAILM against the EKF approach [4] and gradient-descent based on the AMSGrad algorithm [26], for which we compute the gradient as in (15), on real-world data from the nonlinear magneto-rheological fluid damper benchmark proposed in [35]. We use \(N = 2000\) data samples for training and 1499 samples for testing, obtained from the System Identification (SYS-ID) Toolbox for MATLAB R2021a [16], where a nonlinear autoregressive (NLARX) model identification algorithm is employed for black-box nonlinear modeling.

We consider a RNN model (2) with \(n_x = 4\) hidden states and shallow state-update and output network functions (\(L_x = L_y = 2\)) with \(n_x^1 = n_y^1 = 4\) neurons, hyperbolic-tangent activation functions \(f_x^1, f_y^1\), and linear output function \(f_y^2\), parameterized by \(\theta_x \in \mathbb{R}^{14}, \theta_y \in \mathbb{R}^{29}\). The CPU time to solve (29) is approximately 30 ms in all tests.
In (3) we use the quadratic loss \( \ell(y_k, \hat{y}_k) = \frac{1}{N} \| y_k - \hat{y}_k \|_2^2 \) and quadratic regularization \( r(x_0, \theta_x, \theta_y) = \frac{1}{2}(\rho_x \| x_0 \|_2^2 + \rho_\theta \| \theta_x \|_2^2 + \rho_\theta \| \theta_y \|_2^2) \) with \( \rho_x = 1, \rho_\theta = 0.1 \). As \( g(\theta_x, \theta_y) = 0 \), NAILS and NAILM only execute Algorithm 1 once with parameters \( \epsilon_V = 10^{-6} \) and, respectively, \( c_1 = 10^{-4}, \sigma = 0.5, N_\sigma = 20 \), and \( c_2 = 1.5, c_3 = 5, N_\lambda = 20 \). The initial condition is \( x_0 = 0 \) and the components of \( \theta^0_x, \theta^0_y \) are randomly generated as described in Section 3.4.

Table 2 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 − \( \| Y - \hat{Y} \|_2^2 / \| Y - \bar{y} \|_2^2 \)) 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 Figure 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).

It is apparent that NAILS and NAILM obtain similar quality of fit and are computa-
Table 3: Silverbox benchmark: RMSE and BFR obtained by different methods. The numbers reported in brackets refer to the results reported in the corresponding original papers.

| Identification method          | RMSE (mV) | BFR    |
|-------------------------------|-----------|--------|
| ARX  (ml) [18]                | 16.29 [4.40] | 69.22 [73.79] |
| NLARX (ms) [18]               | 8.42 [4.20]  | 83.67 [92.06] |
| NLARX (mlc) [18]              | 1.75 [1.70]   | 96.67 [96.79] |
| NLARX (ms8c50) [18]           | 1.05 [0.30]   | 98.01 [99.43] |
| Recurrent LSTM model [17]     | 2.20       | 95.83  |
| State-space encoder [3] (nx = 4) [1.40] | [97.35]  |
| NAILM (this paper)            | 0.35       | 99.33  |

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 [39] for a detailed description. The first 40000 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 (2) with nx = 8 hidden states, no I/O feedthrough (A^y_1 ∈ \mathbb{R}^{n^y_1 × n^x} in (2b)), 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 with two hidden layers of 4 neurons each mapping the vector v_0 of past na = 8 outputs and nb = 8 inputs to the initial state x_0, hyperbolic-tangent activation functions, and linear output functions. The total number of parameters is n_{θ_x} + n_{θ_y} + n_{θ_a} = 296 + 225 + 128 = 649, that we train using NAILM on E = 150 epochs with ρ = 0.01 and regularization \frac{1}{2} ||θ_a||_2^2 on the parameters defining f_x0. For comparison, we consider the autoregressive models considered in [18], 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 [16] using the same commands reported in [18]. Note that the latter two models contain explicitly the nonlinear term y^3_{k-1}, 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 are shown in Table 3 which also includes the results reported in the recent paper [3] and those obtained by the LSTM model proposed in [17]. The results reproduced here differ from those reported originally in the papers, that we show included in brackets (the results originally reported in [17] have been omitted, as they were obtained on a reduced subset of test and training data).

The resulting model can be retrieved at http://cse.lab.imtlucca.it/~bemporad/shared/silverbox/rnn888.zip.
Figure 2: Fluid damper benchmark: BRF and sparsity of $\theta_x, \theta_y$ for different $\ell_1$-regularization coefficients $\tau$ (mean results over 20 runs from different random initial weights)

| training algorithm | BFR training | BFR test | sparsity % | CPU time (s) |
|--------------------|--------------|----------|------------|--------------|
| NAILS              | 91.00 (1.66) | 87.71 (2.67) | 65.1 (6.5) | 11.4         |
| NAILM              | 91.32 (1.19) | 87.80 (1.86) | 64.1 (7.4) | 11.7         |
| EKF                | 89.27 (1.48) | 86.67 (2.71) | 47.9 (9.1) | 13.2         |
| AMSGrad            | 91.04 (0.47) | 88.32 (0.80) | 12.6 (7.1) | 64.0         |
| Adam               | 90.47 (0.34) | 87.79 (0.44) | 8.3 (3.5)  | 63.9         |
| DiffGrad           | 90.05 (0.64) | 87.34 (1.14) | 7.4 (4.5)  | 63.9         |

Table 4: Fluid damper benchmark: mean BRF, sparsity of $[\theta_x', \theta_y']'$ (with standard deviation in parentheses), and mean CPU time given by different solution methods when solving $\ell_1$-regularized training for $\tau = 0.2$ (results obtained over 20 runs from different random initial weights)

5.2 RNN training with $\ell_1$-regularization

We consider again the fluid damper benchmark dataset and add an $\ell_1$-regularization term $g(\theta_x, \theta_y)$ as in (23) and use NAILS to train the same RNN model structure (2), for different values of $\tau_x = \tau_y \equiv \tau$. Figure 2 shows the resulting BRF. Figure 2 also shows the percentage of zero coefficients in the resulting parameter vector $[\theta_x, \theta_y] \in \mathbb{R}^{73}$, i.e., its sparsity.

The benefit of using $\ell_1$-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 $\ell_1$-regularization terms (Adam [11], AMSGrad [26], DiffGrad [6]), we report in Table 4 the results obtained by solving the training problem for $\tau = 0.2$ using the EKF approach [4] and some of the SGD algorithms mostly used in machine-learning packages. NAILS/NAILM is run for $E = 250$, SGD for $E = 2000$, and EKF for $E = 50$ epochs. While all methods get a similar BFR, NAILS and NAILM are more effective in both sparsifying the model-parameter vector and execution time.

For the Silverbox benchmark problem, following the analysis performed in [19], Figure 3 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 $\ell_1$-regularization term with weight $\tau$ between
Figure 3: Silverbox benchmark: RMSE vs number of parameters. Models ml, ms, mlc, and ms8c50 [18], LSTM model [17], and RNN models generated by NAILM for values of the $\ell_1$-regularization parameter $\tau$ between 0 and 20.

| BFR    | NAILS | 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) |

Table 5: Fluid damper benchmark with quantized model coefficients: mean BRF and CPU time (standard deviation) over 20 runs

$10^{-4}$ and 20. The figure shows the effectiveness of NAILM in trading off model simplicity vs quality of fit.

5.3 RNN training with quantization

Consider again the fluid damper benchmark dataset and let $\theta_x, \theta_y$ be only allowed to take values in the finite set $Q$ 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 $n_x = n_y = 6$. We train the model using NAILS to handle the non-smooth and nonconvex regularization term $g(\theta_x, \theta_y) = 0$ if $\theta_x \in Q^{n_x}, \theta_y \in Q^{n_y}, g(\theta_x, \theta_y) = +\infty$ otherwise, with $E = 1, N_{ADMM} = 200, \rho = 10$, without changing the remaining parameters. For comparison, we solve the same problem without non-smooth regularization term $g$ by running Algorithm 1 for $E = N_{ADMM}$ epochs and then quantize the components of the resulting vectors $\theta_x, \theta_y$ to their closest values in $Q$.

The mean and standard deviation of the BRF and corresponding CPU time obtained over 20 runs from different random initial weights is reported in Table 5. It is apparent that 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 $\tau_g$ described in Section 4.2 to control the model order $n_x$ when training a model on the fluid damper benchmark dataset. We consider the same settings as in Section 5.1 except for the model-order $n_x = 8, n_x = n_y = 6$ 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 $\tau_g$ are reported in Figure 4, 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

\[
x(k + 1) = \begin{bmatrix} 8 & 2 \\ 0 & 1 \end{bmatrix} \text{diag}(x(k))(0.9 + 0.1 \sin(x(k))) + \begin{bmatrix} -1 \\ -1 \end{bmatrix} u(k)(1 - u^3(k)) + \xi(k)
\]

\[
y(k) = \begin{cases} 1 & \text{if } \left\lfloor \frac{\sqrt{y(k) + \frac{1}{3}x^3(k)}}{0.5} \right\rfloor - 4 + \zeta(k) \geq 0 \\ 0 & \text{otherwise} \end{cases}
\]

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 $\xi_i(k), \zeta(k) \sim N(0, \sigma_i^2), i = 1, 2, 3$, for different values of the standard deviation $\sigma_i$. The first $N = 1000$ samples are used for training, the rest for testing. NAILS and NAILM are used to train a RNN model [2] with no feedthrough, one hidden layer ($L_x = L_y = 2$) with $n_1^x = n_1^y = 5$ neurons and hyperbolic-tangent activation function, and sigmoid output function $f_2^y(y) = 1/(1 + e^{-A_2^y [x(k) u(k)]})$. 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 $\ell(y(k), \hat{y}) = \sum_{i=1}^{n_y} -y_i(k) \log(\epsilon + \hat{y}_i) - (1 - y_i(k)) \log(1 + \epsilon - \hat{y}_i)$, where $\hat{y} = f_y(x_k, \theta_y)$ and $\epsilon = 10^{-4}$, quadratic regularization with $\rho_x = 0.1$, $\rho_y = 0.01$, 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 4 shows the final accuracy (%) achieved on training and test data for different levels $\sigma_n$ 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).
| $\sigma_n$ | 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 6: Nonlinear system with binary outputs: mean final accuracy (%) (standard deviation) over 20 runs on training and test data

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 focused on recurrent neural networks of the form [2], 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 and white-box models, and static models.

Future research will be devoted to further analyze the convergence properties of Algorithms 1 and 2 within the Generalized Gauss-Newton framework, and to establish conditions on the ADMM parameter $\rho$, number of epochs $E$ used when solving (22a), the model structure to learn, and the regularization function $g$ that guarantee convergence of the nonconvex ADMM iterations.

7 Acknowledgement

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

References

[1] S.T. Alexander and A.L. Ghirnikar. A method for recursive least squares filtering based upon an inverse QR decomposition. *IEEE Trans. Signal Processing*, 41(1):20–30, 1993.

[2] J.A.E. Andersson, J. Gillis, G. Horn, J.B. Rawlings, and M. Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36, 2019.

[3] G. Beintema, R. Toth, and M. Schoukens. Nonlinear state-space identification using deep encoder networks. In *Proc. Machine Learning Research*, volume 144, pages 241–250, 2021.

[4] A. Bemporad. Recurrent neural network training with convex loss and regularization functions by extended Kalman filtering. 2021. Submitted for publication. Available on [https://arxiv.org/abs/2111.02673](https://arxiv.org/abs/2111.02673)
[5] A. Bemporad and G. Cimini. Variable elimination in model predictive control based on K-SVD and QR factorization. *IEEE Trans. Automatic Control*, 2022. In press. Also available on arXiv at [http://arxiv.org/abs/2012.10423](http://arxiv.org/abs/2012.10423).

[6] S.H. Dubey, S. Chakraborty, S.K. Roy, S. Mukherjee, S.K. Singh, and B.B. Chaudhuri. diffGrad: an optimization method for convolutional neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 31(11):4500–4511, 2019.

[7] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proc. 19th Int. Conf. Artificial Intelligence and Statistics*, pages 249–256, 2010.

[8] M. Hong, Z.-Q. Luo, and M. Razaviyayn. Convergence analysis of alternating direction method of multipliers for a family of nonconvex problems. *SIAM Journal on Optimization*, 26(1):337–364, 2016.

[9] B. Houska, J. Frasch, and M. Diehl. An augmented Lagrangian based algorithm for distributed nonconvex optimization. *SIAM Journal on Optimization*, 26(2):1101–1127, 2016.

[10] K.J. Hunt, D. Sbarbaro, R. Zbikowski, and P.J. Gawthrop. Neural networks for control systems – A survey. *Automatica*, 28(6):1083–1112, 1992.

[11] D.P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] J.Z. Kolter and G. Manek. Learning stable deep dynamics models. *Advances in neural information processing systems*, 32, 2019.

[13] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of applied mathematics*, 2(2):164–168, 1944.

[14] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Kiu, Q. Qiu, W. Xu, X. Lin, X. Qian, and Y. Wang. E-RNN: Design optimization for efficient recurrent neural networks in FPGAs. In *IEEE Int. Symp. High Performance Computer Architecture*, pages 69–80, 2019.

[15] M.M. Livstone, J.A. Farrell, and W.L. Baker. A computationally efficient algorithm for training recurrent connectionist networks. In *Proc. American Control Conference*, pages 555–561, 1992.

[16] L. Ljung. *System Identification Toolbox for MATLAB*. The Mathworks, Inc., 2001. [https://www.mathworks.com/help/ident](https://www.mathworks.com/help/ident).

[17] L. Ljung, C. Andersson, K. Tiels, and T.B. Schön. Deep learning and system identification. *IFAC-PapersOnLine*, 53(2):1175–1181, 2020.

[18] L. Ljung, Q. Zhang, P. Lindskog, and A. Juditski. Estimation of grey box and black box models for non-linear circuit data. *IFAC Proceedings Volumes*, 37(13):399–404, 2004.
[19] A. Marconato, M. Schoukens, Y. Rolain, and J. Schoukens. Study of the effective number of parameters in nonlinear identification benchmarks. In Proc. 52nd IEEE Conf. Dec. Contr., pages 4308–4313, 2013.

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

[21] D. Masti and A. Bemporad. Learning nonlinear state-space models using autoencoders. Automatica, 129:109666, 2021.

[22] F. Messerer, K. Baumgärtner, and M. Diehl. Survey of sequential convex programming and generalized Gauss-Newton methods. ESAIM. Proceedings and Surveys, 71:64, 2021.

[23] J. Nocedal and S.J. Wright. Numerical Optimization. Springer, 2nd edition, 2006.

[24] R. Parisi, E.D. Di Claudio, A. Rapagetta, and G. Orlandi. Recursive least squares approach to learning in recurrent neural networks. In Proc. Int. Conf. on Neural Networks, volume 2, pages 1350–1354, 1996.

[25] G.V. Puskorius and L.A. Feldkamp. Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks. IEEE Transactions on Neural Networks, 5(2):279–297, 1994.

[26] S.J. Reddi, S. Kale, and S. Kumar. On the convergence of adam and beyond. arXiv preprint arXiv:1904.09237, 2019.

[27] A.H. Ribeiro, K. Tiels, J. Umenberger, T.B. Schön, and L.A. Aguirre. On the smoothness of nonlinear system identification. Automatica, 121:109158, 2020.

[28] J.A.K. Suykens, J.P.L. Vandewalle, and B.L.R. De Moor. Artificial neural networks for modelling and control of non-linear systems. Springer Science & Business Media, 1995.

[29] R. Takapoui, N. Moehle, S. Boyd, and A. Bemporad. A simple effective heuristic for embedded mixed-integer quadratic programming. Int. J. Control, 93(1):2–12, 2020.

[30] Y. Tang, Z. Kan, D. Sun, L. Qiao, K. Xiao, Z. Lai, and D. Li. ADMMiRNN: Training RNN with stable convergence via an efficient ADMM approach. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pages 3–18, 2020.

[31] G. Taylor, R. Burmeister, Z. Xu, B. Singh, A. Patel, and T. Goldstein. Training neural networks without gradients: A scalable admm approach. In International Conference on Machine Learning, pages 2722–2731, 2016.

[32] A. Themelis and P. Patrinos. Douglas–Rachford splitting and ADMM for nonconvex optimization: Tight convergence results. SIAM Journal on Optimization, 30(1):149–181, 2020.

[33] M.K. Transtrum and J.P. Sethna. Improvements to the Levenberg-Marquardt algorithm for nonlinear least-squares minimization. arXiv preprint arXiv:1201.5885, 2012.
[34] A.I.F. Vaz and L.N. Vicente. PSwarm: A hybrid solver for linearly constrained global derivative-free optimization. *Optimization Methods and Software*, 24:669–685, 2009. [http://www.norg.uminho.pt/aivaz/pswarm/](http://www.norg.uminho.pt/aivaz/pswarm/)

[35] J. Wang, A. Sano, T. Chen, and B. Huang. Identification of Hammerstein systems without explicit parameterisation of non-linearity. *International Journal of Control*, 82(5):937–952, 2009.

[36] X. Wang and Y. Huang. Convergence study in extended Kalman filter-based training of recurrent neural networks. *IEEE Transactions on Neural Networks*, 22(4):588–600, 2011.

[37] Y. Wang, W. Yin, and J. Zeng. Global convergence of ADMM in nonconvex nonsmooth optimization. *Journal of Scientific Computing*, 78(1):29–63, 2019.

[38] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[39] T. Wigren and J. Schoukens. Three free data sets for development and benchmarking in nonlinear system identification. In *European Control Conference*, pages 2933–2938, Zürich, Switzerland, 2013.

[40] R.J. Williams and J. Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501, 1990.

[41] Q. Xu, K. Krishnamurthy, B. McMillin, and W. Lu. A recursive least squares training algorithm for multilayer recurrent neural networks. In *Proc. American Control Conference*, volume 2, pages 1712–1716, 1994.

[42] M. Yuan and Y. Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.