dynoNet: A neural network architecture for learning dynamical systems

Marco Forgione, Dario Piga
IDSIA Dalle Molle Institute for Artificial Intelligence
SUPSI-USI, Manno, Switzerland
{marco.forgione, dario.piga}@supsi.ch

January 13, 2021

Abstract

This paper introduces a network architecture, called *dynoNet*, utilizing linear dynamical operators as elementary building blocks. Owing to the dynamical nature of these blocks, *dynoNet* networks are tailored for sequence modeling and system identification purposes. The back-propagation behavior of the linear dynamical operator with respect to both its parameters and its input sequence is defined. This enables end-to-end training of structured networks containing linear dynamical operators and other differentiable units, exploiting existing deep learning software. Examples show the effectiveness of the proposed approach on well-known system identification benchmarks.

1 Introduction

1.1 Contribution

This paper introduces *dynoNet*, a neural network architecture tailored for sequence modeling and dynamical system learning (a.k.a. *system identification*). The network is designed to process time series of arbitrary length and contains causal *linear time-invariant* (LTI) dynamical operators as building blocks. These LTI layers are parametrized in terms of rational transfer functions, and thus apply *infinite impulse response* (IIR) filtering to their input sequence. In the *dynoNet* architecture, the LTI layers are combined with *static* (i.e., memoryless) non-linearities which can be either elementary *activation functions* applied channel-wise; fully connected feed-forward neural networks; or other differentiable operators (e.g, polynomials). Both the LTI and the static layers defining a *dynoNet* are in general *multi-input-multi-output* (MIMO) and can be interconnected in an arbitrary fashion.

Overall, the *dynoNet* architecture can represent rich classes of non-linear, causal dynamical relations. Moreover, *dynoNet* networks can be

trained *end-to-end* by plain *back-propagation* using standard *deep learning* (DL) software. Technically, this is achieved by introducing the LTI dynamical layer as a differentiable operator, endowed with a well-defined forward and backward behavior and thus compatible with reverse-mode automatic differentiation [3]. Special care is taken to devise closed-form expressions for the forward and backward operations that are convenient from a computational perspective.

A software implementation of the linear dynamical operator based on the *PyTorch* DL framework [16] has been developed and is available in the GitHub repository `https://github.com/forgi86/dynonet.git`.

1.2 Related works

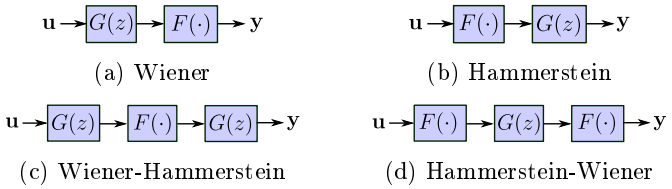
To the best of our knowledge, LTI blocks with an IIR have never been considered as differentiable operators for back-propagation-based training to date. Among the layers routinely applied in DL, 1-D convolution [19] is the closest match. In particular, the 1D causal convolution layer[2, 1] corresponds to the filtering of an input sequence through a causal *finite impulse response* (FIR) dynamical system. The *dynoNet* architecture may be seen as a generalization of the causal 1D *convolutional neural network* (CNN) enabling IIR filtering, owing to the description of the dynamical layers as rational transfer functions. This representation allows modeling long-term (actually infinite) time dependencies with a smaller number of parameters with respect to 1D convolutional networks. Furthermore, filtering through rational transfer function can be implemented by means of recurrent linear difference equations. While this operation is not as highly parallelizable as FIR filtering, the total number of computations required is generally lower.

The *dynoNet* architecture has also analogies with *recurrent neural network* (RNN) [8] architectures. As in RNNs, a dynamic dependency is built exploiting recurrence equations. However, in an RNN the basic computational unit is a neural cell (e.g., Elman, LSTM, GRU) that processes a single time step. The network's computational graph is then built by repeating the same cell for all the steps of the timeseries. Processing long timeseries through an RNN is often computational expensive and presents limited opportunities for parallelization, due to the sequential structure of the computational graph. Conversely, in *dynoNet* a lightweight (linear) recurrence equation is "baked into" the elementary LTI blocks, that naturally operate on time series in a vectorized fashion. While internally these layers require certain sequential operations (details are given in Section 3), the overall computational burden is sensibly lower than the one of typical RNNs. Moreover, the computations performed by the static layers of a *dynoNet* are highly parallelizable, as they are independent for each time step. Therefore, more complex transformations may be included in the static layers.

Thus, compared to 1D convolutional and recurrent neural architectures, *dynoNet* is characterized by an intermediate level of computational complexity and flexibility.

In the system identification literature, particular cases of the *dynoNet* architecture have been widely studied in the last decades within the so-

called *block-oriented* modeling framework [7]. In most of the contributions, shallow architectures based on *single-input single-output* (SISO) blocks are considered. For instance: the Wiener model is defined as the series connection of an LTI dynamical model G followed by a static non-linearity F; the Hammerstein model is based on the reverse connection, with a static non-linearity F followed by an LTI block G; the Wiener-Hammerstein (WH) model combines two SISO LTI blocks interleaved by a static SISO non-linearity in a sequential structure G-F-G; and the Hammerstein-Wiener (HW) has structure F-G-F. See Figure 1 for a visual representation of the aforementioned structures.



(a) Wiener (b) Hammerstein

(c) Wiener-Hammerstein (d) Hammerstein-Wiener

Figure 1: Classic block-oriented architectures. All blocks are SISO.

One exception is the deep architecture consisting in the repeated sequential connection of SISO blocks F-G-F-G ... described in [20] and called generalized Hammerstein-Wiener (Figure 2, left panel). Furthermore, the parallel Wiener-Hammerstein model [17] extends the classic WH model beyond the strictly SISO case. Indeed, the parallel WH model has the same G-F-G as the basic WH mentioned above. However, the first linear block is single-input-multi-output; the static non-linearity F is multi-input-multi-output; and the second linear block is multi-input-single-output (Figure 2, right panel). Overall, the parallel WH model describes an input/output SISO dynamical system, but it leverages on an inner MIMO structure to provide additional flexibility. From a DL perspective, the parallel WH extends the representation capabilities of the plain WH network by including several "neurons" in a single hidden layer, while the generalized HW model aims to the same result by stacking several layers, each one consisting of a single "neuron".

The *dynoNet* architecture encompasses all the previous block-oriented

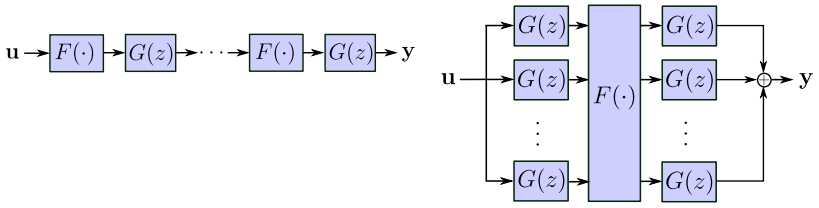


Figure 2: Generalized Hammerstein-Wiener (left) and Parallel Wiener-Hammerstein (right) model structures.

models as special cases. Other structures containing, e.g., multiple MIMO blocks and *skip connection* can be described within the *dynoNet* modeling framework. More importantly, the existing training methods for block-oriented models are custom-made for each specific architecture, requiring for instance analytic expressions of the Jacobian of the loss with respect to the training parameters. Conversely, the derivation of the differentiable dynamical layer allows us to train arbitrary *dynoNet* architectures using the plain back-propagation algorithm.

1.3 Representational power of *dynoNet* architectures

The *dynoNet* framework for dynamical systems is at least as expressive as the block-oriented models in Figures 1 and 2, and causal CNN architectures, being a generalization of the two approaches.

Formal results on the representational power of block-oriented models are given in [15, 5]. It is proven that certain block-oriented models are general approximators of non-linear *fading memory* operators. In particular, the property holds for "Parallel Wiener" architectures consisting in a single-input-multi-output LTI block followed by a multi-input-single-output static non-linearity.

Note that fading memory operators may describe a wide range of non-linear phenomena, but they exclude the cases of time-varying, chaotic, finite-escape-time, unstable, and multiple-equilibria systems. While *dynoNet* can also describe certain unstable systems (rational transfer functions may have unstable dynamics) it is hard to further characterize the *dynoNet* model structure from a system theoretic perspective.

From a practical perspective, the effectiveness of deep CNN architectures for different system identification and time series modeling tasks has been demonstrated in several contributions [19, 2, 1]. Even though mathematically it is not clear whether increasing the number of hidden layers extends the class of dynamics that can be represented by CNNs, experimentally it has been observed that deeper networks are able to learn more complex dependencies than shallower ones, for a given number of training parameters and for a given computational effort.

While the *dynoNet* framework does not increase the theoretical representation power of CNNs dramatically (one could argue that any stable *dynoNet* architecture may be approximated arbitrarily well with a causal CNN, by including convolutional units with a sufficiently long memory), it allows representing long-term dependencies with a smaller number of parameters. This may lead to substantial advantages in terms of generalization properties, memory requirements, training and inference time of the models.

1.4 Notation

The following notation will be used throughout the paper. The entries of an n-length vector are specified by subscript integer indices running from 0 to n–1, unless stated otherwise. The bold-face notation is reserved for real-

valued T-length vectors, generally representing time series with T samples. For instance, $\mathbf{u} \in \mathbb{R}^T$ is a T-length vector with entries $\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{T-1}$.

Time reversal The time reversal of a T-length vector $\mathbf{u} \in \mathbb{R}^T$ is denoted as $\text{flip}(\mathbf{u})$ and defined as

$$\left(\text{flip}(\mathbf{u})\right)_t = \mathbf{u}_{T-t-1}, \qquad t = 0, 1, \ldots, T-1 \tag{1}$$

Convolution The convolution between vectors $x \in \mathbb{R}^{n_x}$ and $y \in \mathbb{R}^{n_y}$ is defined as

$$(x * y)_i = \sum_{j=\max(0, i+1-n_y)}^{\min(i, n_x-1)} x_j y_{i-j}, \qquad i = 0, 1, \ldots, n_x + n_y - 1. \tag{2}$$

Cross-correlation The cross-correlation between vectors $x \in \mathbb{R}^{n_x}$ and $y \in \mathbb{R}^{n_y}$ is defined as

$$(x \star y)_i = \sum_{j=\max(i,0)}^{\min(n_x+i-1, n_y-1)} x_{j-i} y_j, \qquad i = -n_x + 1, \ldots, n_y - 1. \tag{3}$$

2 Linear dynamical operator

The input-output relation of an individual (SISO) dynamical layer in the *dynoNet* architecture is described by the dynamical rational operator $G(q)$ as follows:

$$y(t) = G(q)u(t) = \frac{B(q)}{A(q)} u(t), \tag{4a}$$

where $A(q)$ and $B(q)$ are polynomials in the *time delay operator* q^{-1} ($q^{-1}u(t) = u(t-1)$), i.e.,

$$A(q) = 1 + a_1 q^{-1} + \cdots + a_{n_a} q^{-n_a}, \tag{4b}$$

$$B(q) = b_0 + b_1 q^{-1} + \cdots + b_{n_b} q^{-n_b}, \tag{4c}$$

and $u(t) \in \mathbb{R}$ and $y(t) \in \mathbb{R}$ are the input and output sequence values at time index t.

The filtering operation through $G(q)$ in (4a) is equivalent to the input/output equation:

$$A(q)y(t) = B(q)u(t). \tag{5}$$

Based on the definitions of $A(q)$ and $B(q)$, (5) is equivalent to the recurrence equation:

$$y(t) = b_0 u(t) + b_1 u(t-1) + \cdots + b_{n_b} u(t-n_b) - a_1 y(t-1) \cdots - a_{n_a} y(t-n_a). \tag{6}$$

Parameters The tunable parameters of $G(q)$ are the coefficients of the polynomials $A(q)$ and $B(q)$. For convenience, these coefficients are collected in vectors $a = [a_1 \; a_2 \ldots \; a_{n_a}] \in \mathbb{R}^{n_a}$ and $b = [b_0 \; b_1 \; \ldots \; b_{n_b}] \in \mathbb{R}^{n_b+1}$. Note that the first element of vector a has index 1, while for all other vectors in this paper the starting index is 0.

Initial condition In this paper, the operator $G(q)$ is always initialized from rest, namely the values of $u(t)$ and $y(t)$ for $t < 0$ are all taken equal to zero. Then, given an input sequence $\{u(t),\ t \geq 0\}$, (6) provides an univocal expression for the output sequence $\{y(t),\ t \geq 0\}$.

Finite-length sequences In practice, the operator $G(q)$ in a *dynoNet* operates on finite-length sequences. Let us stack the input and output samples $u(t)$ and $y(t)$ in vectors $\mathbf{u} \in \mathbb{R}^T$ and $\mathbf{y} \in \mathbb{R}^T$, respectively. With a slight abuse of notation, the filtering operation in (4) applied to $\mathbf{u}$ is denoted as

$$\mathbf{y} = G(q)\mathbf{u}.$$

The operation above is also equivalent to the convolution

$$\mathbf{y}_i = (\mathbf{g} * \mathbf{u})_i, \qquad i = 0, 1, \ldots, T-1, \tag{7}$$

where $\mathbf{g} \in \mathbb{R}^T$ is a vector containing the first T samples of the operator's *impulse response*. The latter is defined as the output sequence generated by (6) for an input $u(\cdot)$ such that $u(0) = 1$ and $u(t) = 0, \forall t \geq 1$.

MIMO extension In the MIMO case, the input $u(t) \in \mathbb{R}^p$ and output $y(t) \in \mathbb{R}^m$ at time t are *vectors* of size p and m, respectively. The MIMO linear dynamical operator with p input and m output channels may be represented as a $m \times p$ MIMO transfer function matrix $G(q)$ whose element $G_{kh}(q)$ is a SISO rational transfer function such as (4). The components $y_k(t)$ of the output sequence $y(t)$ at time t are defined as

$$y_k(t) = \sum_{h=0}^{p-1} G_{kh}(q) u_h(t), \qquad k = 0, 1, \ldots m-1. \tag{8}$$

The derivations for the dynamical layer are presented in the following in a SISO setting to avoid notation clutter. Extension to the MIMO case is straightforward and only requires repetition of the same operations for the different input/output channels. The computations for the different input/output channels are independent and therefore may be performed in parallel.

Note that the software implementation of the operator available in our on-line GitHub repository fully supports the MIMO case.

3 Dynamical operator as a deep learning layer

In this section, the forward and backward operations required to integrate the linear dynamical operator in a DL framework are derived. The computational cost of these operations as measured by the number of multiplications to be executed is also reported. Furthermore, the possibility of parallelizing these computations is analyzed.

In the rest of this paper, the linear dynamical operator interpreted as a differentiable layer for use in DL is also referred to as G-block. In our

software implementation, the G-block is implemented in the *PyTorch* DL framework as a class extending `torch.autograd.Function`, based on the forward and backward operations derived in the following.

3.1 Forward operations

The forward operations of a G-block embedded in a computational graph are represented by solid arrows in Figure 3. In the forward pass, the block filters an input sequence $\mathbf{u} \in \mathbb{R}^T$ through a dynamical system $G(q)$ with structure (4) and parameters $a = [a_1 \ \dots \ a_{n_a}]$ and $b = [b_0 \ b_1 \ \dots \ b_{n_b}]$. The block output is a vector $\mathbf{y} \in \mathbb{R}^T$ containing the filtered sequence:

$$\mathbf{y} = G.\text{forward}(\mathbf{u}, b, a) = G(q)\mathbf{u}. \tag{9}$$

The input $\mathbf{u}$ of the G-block may be either the training input sequence or the result of previous operations in the computational graph, while the output $\mathbf{y}$ is an intermediate step towards the computation of a scalar output $\mathcal{L}$. The exact operations leading to $\mathcal{L}$ are not relevant in this discussion, and thus they are not further specified.

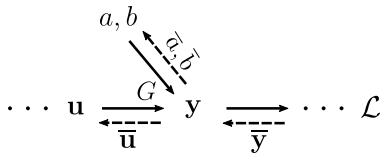


Figure 3: Forward and backward operations of a G-block within a computational graph.

When the filtering operation (9) is implemented using (6), the computational cost of the G-block forward pass corresponds to $T(n_b + n_a + 1)$ multiplications. These multiplications can be parallelized for the $n_b + n_a + 1$ different coefficients at a given time step, but need to be performed sequentially for the T time samples due to the recurrent structure of (6).

3.2 Backward operations

The backward operations are illustrated in Figure 3 with dashed arrows. In the backward pass, G receives the vector $\overline{\mathbf{y}} \in \mathbb{R}^T$ containing the partial derivatives of the loss $\mathcal{L}$ w.r.t. $\mathbf{y}$, namely:

$$\overline{\mathbf{y}}_t = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t}, \quad t = 0, \dots, T-1. \tag{10}$$

Given $\overline{\mathbf{y}}$, the G-block has to compute the derivatives of the loss $\mathcal{L}$ w.r.t. its differentiable inputs b, a, and $\mathbf{u}$. Overall, the backward operation has the following structure:

$$\overline{b}, \overline{a}, \overline{\mathbf{u}} = G.\text{backward}(\mathbf{u}, b, a, \overline{\mathbf{y}}), \tag{11}$$

where

$$\overline{b}_j = \frac{\partial \mathcal{L}}{\partial b_j}, \qquad j = 0, \ldots, n_b \tag{12a}$$

$$\overline{a}_j = \frac{\partial \mathcal{L}}{\partial a_j}, \qquad j = 1, \ldots, n_a \tag{12b}$$

$$\overline{\mathbf{u}}_\tau = \frac{\partial \mathcal{L}}{\partial \mathbf{u}_\tau}, \qquad \tau = 0, 1, \ldots, T-1. \tag{12c}$$

Numerator coefficients b Application of the chain rule leads to:

$$\overline{b}_j = \sum_{t=0}^{T-1} \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} \frac{\partial \mathbf{y}_t}{\partial b_j} = \sum_{t=0}^{T-1} \overline{\mathbf{y}}_t \frac{\partial \mathbf{y}_t}{\partial b_j}$$

The required sensitivities $\tilde{b}_j(t) = \frac{\partial \mathbf{y}_t}{\partial b_j}$, $j = 0, 1, \ldots, n_b$, can be obtained in closed-form through additional filtering operations[11]. Specifically, an expression for $\tilde{b}_j(t)$ is derived by differentiating the left and hand side of Eq. (5) w.r.t b_j. This yields:

$$A(q)\tilde{b}_j(t) = u(t-j), \tag{13}$$

or equivalently:

$$\tilde{b}_j(t) = \frac{1}{A(q)} u(t-j). \tag{14}$$

Thus, $\tilde{b}_j(t)$ can be computed by filtering the input vector $u(t)$ through the linear filter $\frac{1}{A(q)}$. Furthermore, the following condition holds:

$$\tilde{b}_j(t) = \begin{cases} \tilde{b}_0(t-j), & t-j \geq 0 \\ 0, & t-j < 0. \end{cases} \tag{15}$$

Then, one only needs to compute $\tilde{b}_0(t)$ by simulating the recursive equation (13). The other sensitivities $\tilde{b}_j(t)$, $j = 1, \ldots, n_b$, are obtained through simple shifting operations according to (15).

Exploiting expression (15) for $\tilde{b}_j(t)$, the j-th component of $\overline{b}$ is obtained as:

$$\overline{b}_j = \sum_{t=j}^{T-1} \overline{\mathbf{y}}_t \tilde{b}_0(t-j). \tag{16}$$

This operation corresponds to the dot product of $\overline{\mathbf{y}}$ with shifted version of the sensitivity $b_0(t)$.

Overall, the computation of $\overline{b}$ requires: (i) filtering $\mathbf{u}$ through $\frac{1}{A(q)}$, which entails Tn_a multiplications and (ii) the $n_b{+}1$ dot products defined in (16), totaling $T(n_b{+}1) - n_b$ multiplications. As for the filtering, the operations have to be performed sequentially for the different time steps due to the recursive structure of (13). After completion of the filtering, the dot product operations may be performed in parallel.

Denominator coefficients a Following the same rationale above, we obtain a closed-form expression for the sensitivities $\tilde{a}_j(t) = \frac{\partial \mathbf{y}_t}{\partial a_j}$, $j = 1, 2, \ldots, n_a$, by differentiating the left and right hand side of Eq. (5) with respect to a_j. This yields:

$$y(t-j) + A(q)\frac{\partial y(t)}{\partial a_j} = 0,$$

or equivalently

$$\tilde{a}_j(t) = -\frac{1}{A(q)} y(t-j). \tag{17}$$

Then, $\tilde{a}_j(t)$ can be obtained by filtering the output $\mathbf{y}$ through the linear filter $-\frac{1}{A(q)}$. Furthermore, the following condition holds:

$$\tilde{a}_j(t) = \begin{cases} \tilde{a}_1(t-j+1), & t-j+1 \geq 0 \\ 0, & t-j+1 < 0. \end{cases} \tag{18}$$

The j-th component of $\overline{a}$ is obtained as:

$$\overline{a}_j = \sum_{t=j-1}^{T-1} \overline{\mathbf{y}}_t \tilde{a}_1(t-j+1). \tag{19}$$

The back-propagation for the denominator coefficients requires: (i) the filtering operation (17), which involves Tn_a multiplications; and (ii) the n_a dot products defined in (19), totaling $Tn_a - n_a + 1$ multiplications.

Input time series u Application of the chain rule yields:

$$\overline{\mathbf{u}}_\tau = \frac{\partial \mathcal{L}}{\partial \mathbf{u}_\tau} = \sum_{t=0}^{T-1} \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} \frac{\partial \mathbf{y}_t}{\partial \mathbf{u}_\tau} = \sum_{t=0}^{T-1} \overline{\mathbf{y}}_t \frac{\partial \mathbf{y}_t}{\partial \mathbf{u}_\tau} \tag{20}$$

From (7), the following expression for $\frac{\partial \mathbf{y}_t}{\partial \mathbf{u}_\tau}$ holds:

$$\frac{\partial \mathbf{y}_t}{\partial \mathbf{u}_\tau} = \begin{cases} \mathbf{g}_{t-\tau}, & t-\tau \geq 0 \\ 0, & t-\tau < 0. \end{cases} \tag{21}$$

Plugging the expression above for $\frac{\partial \mathbf{y}_t}{\partial \mathbf{u}_\tau}$ into (20), we obtain

$$\overline{\mathbf{u}}_\tau = \sum_{t=\tau}^{T-1} \overline{\mathbf{y}}_t \mathbf{g}_{t-\tau}$$

By definition, the expression above corresponds to the following cross-correlation operation:

$$\overline{\mathbf{u}}_\tau = (\mathbf{g} \star \overline{\mathbf{y}})_\tau, \qquad \tau = 0, 1, \ldots T-1. \tag{22}$$

However, direct implementation of (22) requires a number of operations *quadratic* in T.

In order to obtain a more efficient solution, we observe that:

$$\overline{\mathbf{u}}_0 = \sum_{t=0}^{T-1} \overline{\mathbf{y}}_t \mathbf{g}_t, \qquad \overline{\mathbf{u}}_1 = \sum_{t=1}^{T-1} \overline{\mathbf{y}}_t \mathbf{g}_{t-1}, \qquad \overline{\mathbf{u}}_2 = \sum_{t=2}^{T-1} \overline{\mathbf{y}}_t \mathbf{g}_{t-2},$$
$$\ldots, \qquad \overline{\mathbf{u}}_{T-2} = \overline{\mathbf{y}}_{T-2}\mathbf{g}_0 + \overline{\mathbf{u}}_{T-1}\mathbf{g}_1, \qquad \overline{\mathbf{u}}_{T-1} = \overline{\mathbf{y}}_{T-1}\mathbf{g}_0.$$

Since $\mathbf{g}$ represents the impulse response of the filter $G(q)$, the vector $\overline{\mathbf{u}}$ may also be obtained by filtering the vector $\overline{\mathbf{y}}$ in reverse time through $G(q)$, and then reversing the result, i.e.,

$$\overline{\mathbf{u}} = \mathrm{flip}\big(G(q)\mathrm{flip}(\overline{\mathbf{y}})\big). \tag{23}$$

Neglecting the flipping operations, the computational cost of the backward pass for $\mathbf{u}$ implemented using (23) is *linear* in T. Indeed, it is equivalent to the filtering of a T-length vector through $G(q)$, which requires $T(n_b + n_a + 1)$ multiplications.

4 Special cases of linear dynamical operators

In this section, two special cases of the linear dynamical operators, namely the finite impulse response and the second-order structures are analyzed. The first case is interesting as it corresponds to the convolutional block of a standard 1D CNN, which is generalized by the *dynoNet* architecture. The latter is useful in practice as: (*i*) higher-order systems may always be described as the sequential connection of first- and second-order dynamics; (*ii*) the coefficients of a second-order system can be readily reparametrized in order to enforce stability of the dynamical blocks and thus of the whole *dynoNet* network.

4.1 Finite impulse response structure

A finite impulse response (FIR) dynamical operator has structure

$$G(q) = b_0 + b_1 q^{-1} + \cdots + n_{n_b} q^{-n_b}. \tag{24}$$

In the FIR structure, there are no denominator coefficients a. Furthermore, the numerator coefficients b correspond to the system's non-zero impulse response coefficients. For these reasons, the formulas derived in Section 3 required to define the forward and backward behavior of a general G-block simplify significantly in the FIR case.

Forward operations The forward pass operation is equivalent to

$$\mathbf{y} = G(q)\mathbf{u}, \tag{25}$$

which is equivalent to the convolution

$$\mathbf{y}_i = (\mathbf{g} * \mathbf{u})_i, \qquad i = 0, 1, \ldots, T-1 \tag{26a}$$
$$= (b * \mathbf{u})_i, \qquad i = 0, 1, \ldots, T-1. \tag{26b}$$

Using (26b) for the implementation, the forward operation of a FIR G-block requires $T(n_b + 1)$ fully parallelizable multiplications.

Backward operations The sensitivities of the block output $\mathbf{y}$ with respect to the numerator coefficients b are given by

$$\tilde{b}_j(t) = \frac{\partial \mathbf{y}_t}{\partial b_j} = \mathbf{u}_{t-j}, \qquad j = 0, 1, \dots, n_b.$$

Applying the chain rule, we obtain

$$\overline{b}_j = \frac{\partial \mathcal{L}}{\partial b_j} = \sum_{t=0}^{T} \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} \frac{\partial \mathbf{y}_t}{\partial b_j} = \sum_{t=j}^{T} \overline{\mathbf{y}}_t \mathbf{u}_{t-j}.$$

The latter can also be written as

$$\overline{b}_j = (\mathbf{u} \star \overline{\mathbf{y}})_j, \qquad j = 0, 1, \dots, n_b.$$

Thus, computing $\overline{b}$ requires $T(n_b + 1)$ fully parallelizable multiplications.

As for the back-propagation operations with respect to the input time series $\mathbf{u}$, Equation (21) of the main paper still holds in the FIR case. Applying this equation to the FIR case, we obtain:

$$\overline{\mathbf{u}}_\tau = (b \star \overline{\mathbf{y}})_\tau, \qquad \tau = 0, 1, \dots, T-1.$$

This operation also requires $T(n_b + 1)$ fully parallelizable multiplications.

The formulas presented above for the FIR structure are very similar to the ones used in 1D-CNNs. In most deep learning frameworks, however, the cross-correlation is implemented as forward operation. Following the same rationale above, convolution operations appears then in the backward computations.

Discussion A significant limitation of the FIR structure is that a large number of coefficient b is required to represent an LTI dynamics whose impulse response decays slowly. On the other hand, all operations can be performed in parallel as they are independent for each time step, owing to the non-recurrent structure. Furthermore, the FIR representation defines by construction a stable LTI dynamics, which could have numerical advantages in training.

4.2 Second-order structure

A second-order dynamical operator has structure

$$G(q) = \frac{B(q)}{A(q)} = \frac{b_0 + b_1 q^{-1} + b_2 q^{-1}}{1 + a_1 q^{-1} + a_2 q^{-2}}. \tag{27}$$

The second-order structure is interesting as (i) higher-order systems may always be described as the sequential connection of first- and second-order dynamics and (ii) the coefficients of a second-order system can be readily re-parametrized to enforce stability of the block, and thus of the entire *dynoNet* network.

System analysis The second-order filter $G(q)$ above is asymptotically stable if and only if the two roots of the characteristic polynomial

$$P(q) = q^2 + a_1 q + a_2 \tag{28}$$

lie within the complex unit circle.

By applying the Jury stability criterion [14], it is possible to show that this property holds in the region of the coefficient space characterized by:

$$|a_1| < 2 \tag{29a}$$
$$|a_1| - 1 < a_2 < a_1. \tag{29b}$$

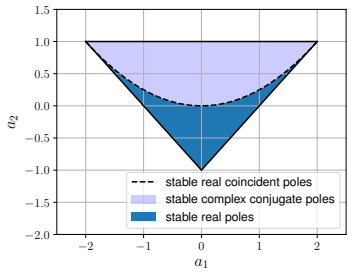


Figure 4: Stability region of a second-order transfer function in the coefficient space.

Furthermore, the two poles are: (i) real and distinct for $a_2 < \frac{a_1^2}{4}$; (ii) complex conjugate for $a_2 > \frac{a_1^2}{4}$; and (iii) real and coincident for $a_2 = \frac{a_1^2}{4}$. The regions of interest in the coefficient space are illustrated in Figure 4.

Stable parametrization An intuitive stable parametrization for second-order dynamical layers is obtained by describing the denominator $A(q)$ in terms of two complex conjugate (or coincident) poles:

$$A(q) = (1 - re^{j\beta}q^{-1})(1 - re^{-j\beta}q^{-1}) = 1 - 2r\cos\beta q^{-1} + r^2 q^{-2},$$

with magnitude r, $0 \leq r < 1$ and phase β, $0 \leq \beta < \pi$. Next, in order to avoid interval constraints on r and β, one can further parametrize r and β in terms of unconstrained variables ρ, ψ as follows:

$$r = \sigma(\rho)$$
$$\beta = \pi\sigma(\psi),$$

where $\sigma(\cdot)$ denotes the sigmoid function and $\rho, \psi \in \mathbb{R}$.

The overall transformation from ρ, ψ to a_1, a_2 is then:

$$a_1 = -2\sigma(\rho)\cos(\pi\sigma(\psi)) \tag{31a}$$
$$a_2 = \sigma(\rho)^2. \tag{31b}$$

Adopting this parametrization, it is possible to train *dynoNet* networks that are stable by design. In practice, the trainable parameters ρ and ψ may be introduced in the computational graph as parents—through the differentiable transformation (31)—of the coefficients a_1, a_2 of a second-order G-block. Then, the variables ρ, ψ can be optimized (along with the numerator coefficients b_0, b_1, b_2 and all other model parameters) using standard unconstrained algorithms, with gradients computed by plain back-propagation. The learned denominator coefficients a_1, a_2 will describe a stable second-order dynamics.

Figure 5 represents the computational graph of a second-order G-block augmented with the stable re-parametrization of the denominator coefficients a_1, a_2 in terms of the unconstrained variables ρ, ψ. Note that the re-parametrization formula (31) included in the computational graph may be implemented using standard differentiable blocks readily available in deep learning software. Thus, back-propagation through this operation does not entail particular difficulties.

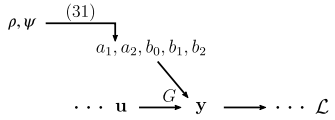


Figure 5: Computational graph of a second-order G-block with stable re-parametrization of the denominator coefficient according to transformation (31).

The parametrization (31) excludes however the case of two distinct real poles. In order to allow this system structure, a slightly more complex parametrization spanning the whole stability region (29) for the coefficients a_1 and a_2 may be used, e.g.:

$$a_1 = 2\tanh(\alpha_1) \tag{32a}$$

$$a_2 = |2\tanh(\alpha_1)| + (2 - |2\tanh(\alpha_1)|)\sigma(\alpha_2) - 1, \tag{32b}$$

where $\alpha_1, \alpha_2 \in \mathbb{R}$ are used as unconstrained optimization variables.

5 Examples

The effectiveness of the *dynoNet* architecture is evaluated on system identification benchmarks publicly available at the website `www.nonlinearbenchmark.org`. All the codes required to reproduce the results in this section are available on the GitHub repository `https://github.com/forgi86/dynonet.git`

The results achieved on the Wiener-Hammerstein [12], the Bouc-Wen [13], and the electro-mechanical positioning System (EMPS) [9] benchmarks are presented in this paper. Other examples are dealt with in the provided codes.

Settings In the following examples, the *dynoNet* is trained by minimizing the mean square of the simulation error and using the Adam algorithm

[10] for gradient-based optimization. The number n of iterations is chosen sufficiently large to reach a cost function plateau. The learning rate λ is adjusted by a rough trial and error. All static non-linearities following the G-blocks are modeled as feed-forward neural networks with a single hidden layer containing 20 neurons and hyperbolic tangent activation function. The numerator and denominator coefficients of the linear dynamical G-blocks are randomly initialized from a uniform distribution with zero mean and range $[-0.01\ 0.01]$, while the feed-forward neural network parameters are initialized according to PyTorch's default strategy. Note that several settings are kept constant across the benchmarks to highlight that limited tuning is needed to obtain state-of-the art identification results using the proposed *dynoNet* architecture.

Hardware setup All computations are performed on a desktop computer equipped with an AMD Ryzen 5 1600x 6-core processor and 32 GB of RAM.

Metrics The identified models are evaluated in terms of the fit and Root Mean Square Error (RMSE) indexes defined as:

$$\text{fit} = 100 \cdot \left(1 - \frac{\sqrt{\sum_{t=0}^{T-1} \left(\mathbf{y}_t^{\text{meas}} - \mathbf{y}_t\right)^2}}{\sqrt{\sum_{t=0}^{T-1} \left(\mathbf{y}_t^{\text{meas}} - \overline{\mathbf{y}}\right)^2}} \right) (\%), \qquad \text{RMSE} = \sqrt{\frac{1}{T} \sum_{t=0}^{T-1} \left(\mathbf{y}_t^{\text{meas}} - \mathbf{y}_t\right)^2},$$

where $\mathbf{y}^{\text{meas}}$ is the measured (true) output vector; $\mathbf{y}$ is the *dynoNet* model's open-loop simulated output vector; and $\overline{\mathbf{y}}$ is the mean value of $\mathbf{y}^{\text{meas}}$, i.e. $\overline{\mathbf{y}} = \frac{1}{T} \sum_{t=0}^{T-1} \mathbf{y}_t^{\text{meas}}$.

5.1 Electronic circuit with Wiener-Hammerstein structure

The experimental setup used in this benchmark is an electronic circuit that behaves by construction as a Wiener-Hammerstein system [12]. Therefore, a simple *dynoNet* architecture corresponding to the WH model structure is adopted. Specifically, the *dynoNet* model has a sequential structure defined by a SISO G-block with $n_a = n_b = 8$; a SISO feed-forward neural network; and a final SISO G-block with $n_a = n_b = 8$.

The model is trained over $n = 40000$ iterations of the Adam algorithm with learning rate $\lambda = 10^{-4}$, by minimizing the MSE on the whole training dataset ($T = 100000$ samples). The total training time is 267 seconds. On the test dataset ($T = 87000$ samples), the *dynoNet* model's performance indexes are fit$=99.5\%$ and RMSE$=1.2$ mV. The measured output $\mathbf{y}^{\text{meas}}$ and simulated output $\mathbf{y}$ on a portion of the test dataset are shown in Figure 6a, together with the simulation error $\mathbf{e} = \mathbf{y}^{\text{meas}} - \mathbf{y}$. While specialized training algorithms for WH systems may provide even superior results on this benchmark (the best published result [18] reports RMSE $= 0.28$ mV), the *dynoNet* model trained by plain back-propagation achieves remarkably good performance.

5.2 Bouc-Wen system

The Bouc-Wen is a nonlinear dynamical system describing hysteretic effects in mechanical engineering and commonly used to assess system identification algorithms. The example in this section is based on the synthetic Bouc-Wen benchmark[13]. The training and test datasets of the benchmark are obtained by numerical simulation of the differential equations:

$$
\begin{aligned}
&m_L\ddot{y}(t) + k_L y(t) + c_L\dot{y}(t) + z(t) = u(t) \\
&\dot{z}(t) = \alpha\dot{y}(t) - \beta\left(\gamma|\dot{y}(t)||z(t)|^{\nu-1}z(t) + \delta\dot{y}(t)|z(t)|^{\nu}\right),
\end{aligned}
$$

where $u(t)$ (N) is the input force; $z(t)$ (N) is the hysteretic force; $y(t)$ (mm) is the output displacement; and all other symbols represent fixed coefficients. The input $u(t)$ and output $y(t)$ signals are available at a sampling frequency $f_s = 750$ Hz. The output $y(t)$ is corrupted by an additive band-limited Gaussian noise with bandwidth 375 Hz and standard deviation $8 \cdot 10^{-3}$ mm. The training and test datsaset for the benchmark are generated using as input independent random phase multisine sequences containing 40960 and 8192 samples, respectively.

We adopt for this benchmark a *dynoNet* architecture with two parallel branches. The first branch has a sequential structure containing: a G-block with 1 input and 8 output channels; a feed-forward network with 8 input and 4 output channels; a G-block with 4 input and 4 output channels; and a feed-forward neural network with 4 input and 1 output channel, while the second branch consists in a single SISO G-block. The model output is the sum of the two branches. All the G-blocks in the first branch are third-order ($n_a = n_b = 3$), while the single G-block in the second branch is second-order ($n_a = n_b = 2$). This model does not have a specific physical motivation and it is chosen to showcase the representational power of *dynoNet*. Furthermore, it does not correspond to any classic block-oriented structure previously considered in the system identification literature.

The model is trained over $n = 10000$ iterations with learning rate $\lambda = 2 \cdot 10^{-3}$, by minimizing the MSE on the whole training dataset. On the test dataset, the model achieves a fit index of 93.2% and a RMSE of $4.52 \cdot 10^{-5}$ mm. Time traces of the measured and simulated *dynoNet* output on a portion of the test dataset are shown in Figure 6b. The results obtained by the *dynoNet* compare favorably with other general black-box identification methods applied to this benchmark. For instance, Nonlinear Finite Impulse Response (NFIR); Auto Regressive with eXogenous input (NARX); and Orthornormal Basis Function (NOBF) model structures are tested on this benchmark [4]. The best results are obtained with the NFIR structure (RMSE $= 16.3 \cdot 10^{-5}$ mm). Superior results are achieved using polynomial nonlinear state-space models [6] trained with an algorithm tailored for the identification of hysteretic systems (RMSE $= 1.87 \cdot 10^{-5}$ mm).

5.3 Electro-mechanical positioning system

The EMPS is a controlled prismatic joint, which is a common component of robots and machine tools. In the experimental benchmark [9], the

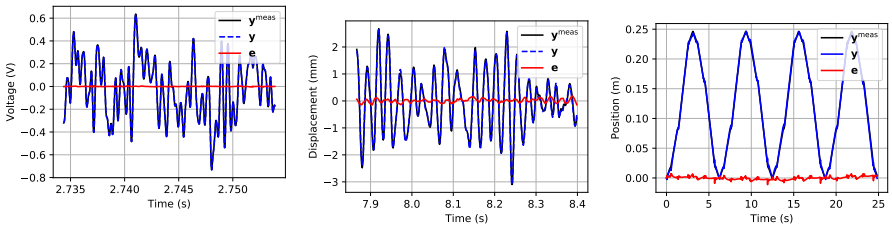
system input is the motor force F (N) and the measured output is the prismatic joint position y (m). A physical model for the system is:

$$\ddot{y}(t) = -\frac{F(t)}{M} - \frac{F_d(\dot{y}(t))}{M},$$

where M (kg) is the joint mass and $F_d(t)$ (N) is the friction affecting the system (comprising both viscous and Coloumb terms). From a system identification perspective, this benchmark is challenging due to (*i*) the unknown (and possibly complex) friction characteristic, (*ii*) the marginally stable (integral) system dynamics, and (*iii*) actuator and sensor behavior affecting the measured data.

We adopt for this benchmark a sequential *dynoNet* structure comprising: a third-order G-block with 1 input and 20 output channels; a feed-forward neural network with 20 input and 1 output channel; and a final integrator block. In this architecture, the final integrator is used to model the integral system dynamics, while the other units have no specific physical meaning and are used as black-box model components.

By training this *dynoNet* model over $n=50000$ iterations on a dataset with $T = 24841$ samples with learning rate $\lambda=1 \cdot 10^{-4}$, we obtain on the test dataset performance indexes fit $= 96.8\%$ and RMSE $= 2.64{\cdot}10^{-3}$ mm. As reference, a fit $= 25.4\%$ for the best linear model was obtained on this benchmark [9]. Time traces of the measured and simulated output on the test dataset are shown in Figure 6c.



(a) Wiener-Hammerstein circuit.

(b) Bouc-Wen.

(c) EMPS.

Figure 6: Measured output $\mathbf{y}^{\text{meas}}$ (black), *dynoNet* simulated output $\mathbf{y}$ (blue), and simulation error $\mathbf{e} = \mathbf{y} - \mathbf{y}^{\text{meas}}$ (red) on the test dataset for the three benchmarks.

6 Conclusions

We have introduced *dynoNet*, a neural network architecture tailored for time series processing and dynamical system learning. The core element of *dynoNet* is a linear *infinite impulse response* dynamical operator described by a rational transfer function. We have derived all the formulas required to integrate the dynamical operator in a back-propagation-based optimization engine for end-to-end training of deep networks. Furthermore,

we have analyzed the computational cost of the forward and backward operations.

The proposed case studies have shown the effectiveness and flexibility of the presented methodologies against well-known system identification benchmarks.

Current and future research activities are devoted to the design of nonlinear state estimators and control strategies for systems modeled by *dynoNet* networks.

Acknowledgments

This work was partially supported by the European H2020-CS2 project ADMITTED, Grant agreement no. GA832003.

References

[1] C. Andersson, A. H. Ribeiro, K. Tiels, N. Wahlström, and T. B. Schön. Deep convolutional networks in system identification. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 3670–3676, 2019.

[2] S. Bai, J.Z Kolter, and V. Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

[3] A.G. Baydin, B.A. Pearlmutter, A.A Radul, and J.M Siskind. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(1):5595–5637, 2017.

[4] J. Belz, T. Münker, T.O. Heinz, G. Kampmann, and O. Nelles. Automatic modeling with local model networks for benchmark processes. *IFAC-PapersOnLine*, 50(1):470–475, 2017.

[5] S. Boyd and L. Chua. Fading memory and the problem of approximating nonlinear operators with Volterra series. *IEEE Transactions on Circuits and Systems*, 32(11):1150–1161, 1985.

[6] A.F. Esfahani, P. Dreesen, K. Tiels, J.P. Noël, and J. Schoukens. Polynomial state-space model decoupling for the identification of hysteretic systems. *IFAC-PapersOnLine*, 50(1):458–463, 2017.

[7] F. Giri and E. Bai, editors. *Block-oriented Nonlinear System Identification*, volume 404 of *Lecture Notes in Control and Information Sciences*. 2010.

[8] K. Greff, R.K. Srivastava, J. Koutník, B.R. Steunebrink, and J. Schmidhuber. LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, 2016.

[9] A. Janot, M. Gautier, and M. Brunot. Data Set and Reference Models of EMPS. In *Nonlinear System Identification Benchmarks*, EINDHOVEN, Netherlands, April 2019.

[10] D.P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[11] L. Ljung. *System Identification: Theory for the User*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 1999.

[12] L. Ljung, J. Schoukens, and J. Suykens. Wiener-Hammerstein benchmark. In *15th IFAC Symposium on System Identification, Saint-Malo, France, July, 2009*, 2009.

[13] J.P. Noël and M Schoukens. Hysteretic benchmark with a dynamic nonlinearity. In *Workshop on Nonlinear System Identification Benchmarks*, pages 7–14, 2016.

[14] Katsuhiko Ogata et al. *Discrete-time control systems*, volume 2. Prentice Hall Englewood Cliffs, NJ, 1995.

[15] G. Palm. On representation and approximation of nonlinear systems. *Biological Cybernetics*, 34(1):49–52, 1979.

[16] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[17] M. Schoukens, A. Marconato, R. Pintelon, G. Vandersteen, and Y. Rolain. Parametric identification of parallel wiener–hammerstein systems. *Automatica*, 51:111–122, 2015.

[18] M. Schoukens, R. Pintelon, and Y. Rolain. Identification of Wiener–Hammerstein systems by a nonparametric separation of the best linear approximation. *Automatica*, 50(2):628–634, 2014.

[19] Z. Wang, W. Yan, and T. Oates. Time series classification from scratch with Deep Neural Networks: A strong baseline. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1578–1585. IEEE, 2017.

[20] A. Wills and B. Ninness. Generalised hammerstein–wiener system estimation and a benchmark application. *Control Engineering Practice*, 20(11):1097–1108, 2012.