dynoNet: a neural network architecture for learning dynamical systems

Marco Forgione, Dario Piga

[1]IDSIA Dalle Molle Institute for Artificial Intelligence SUPSI-USI, Lugano, Switzerland

July 20, 2020

Motivations

Two main classes of neural network structures for sequence modeling and system identification:

Recurrent NNs

General state-space models

- General state-space models
- High representational capacity
- Difficulties in training

1D Convolutional NNs

Dynamics through FIR blocks

- Lower capacity
- Several parameters
- Fast, well-behaved training

We introduce *dynoNet*: a neural network architecture using linear dynamical operators (rational transfer functions) as building blocks.

- Extends 1D Convolutional NNs to Infinite Impulse Response dynamics
- Can be trained by plain back-propagation

Motivations

Two main classes of neural network structures for sequence modeling and system identification:

Recurrent NNs

General state-space models

- General state-space models
- High representational capacity
- Difficulties in training

1D Convolutional NNs

Dynamics through FIR blocks

- Lower capacity
- Several parameters
- Fast, well-behaved training

We introduce *dynoNet*: a neural network architecture using linear dynamical operators (rational transfer functions) as building blocks.

- Extends 1D Convolutional NNs to Infinite Impulse Response dynamics
- Can be trained by plain back-propagation

Motivations

Two main classes of neural network structures for sequence modeling and system identification:

Recurrent NNs

General state-space models

- General state-space models
- High representational capacity
- Difficulties in training

1D Convolutional NNs

Dynamics through FIR blocks

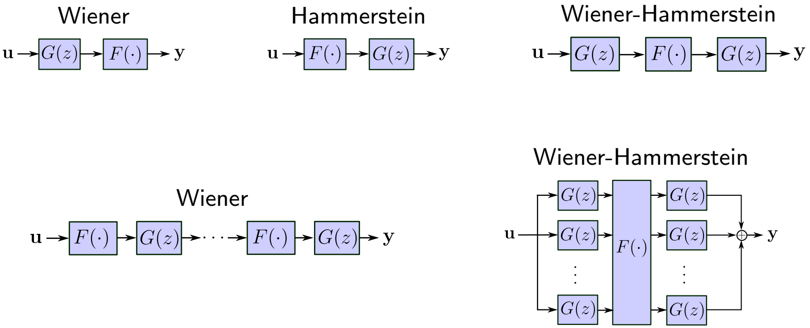
- Lower capacity
- Several parameters
- Fast, well-behaved training

We introduce *dynoNet*: a neural network architecture using linear dynamical operators (rational transfer functions) as building blocks.

- Extends 1D Convolutional NNs to Infinite Impulse Response dynamics
- Can be trained by plain back-propagation

Related works

Block-oriented architectures consist in the interconnection of transfer functions $G(z)$ and static non-linearities $F(\cdot)$:

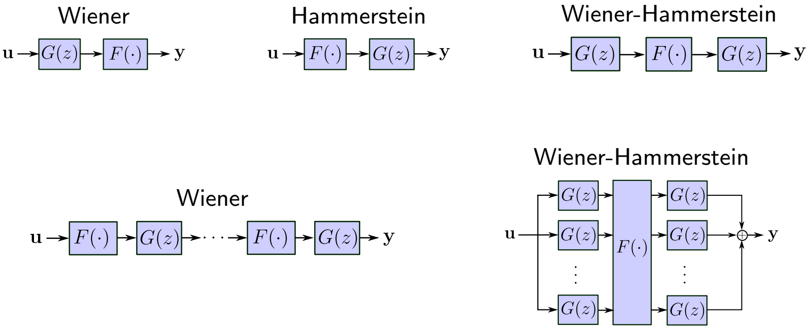


extensively studied in System Identification.

Training through specialized algorithms requiring, e.g. analytic expressions of gradients/jacobians.

Related works

Block-oriented architectures consist in the interconnection of transfer functions $G(z)$ and static non-linearities $F(\cdot)$:



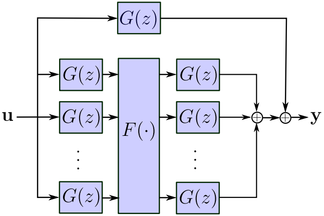
extensively studied in System Identification.

Training through specialized algorithms requiring, e.g. analytic expressions of gradients/jacobians.

dynoNet

- *dynoNet* generalizes block-oriented models to arbitrary connection of MIMO blocks $G(z)$ and $F(\cdot)$
- More importantly, training is performed using a general approach
- Plain back-propagation for gradient computation

a *dynoNet* network



Technical challenge: back-propagation through the transfer function!
No hint in the literature, no implementation in Deep Learning toolboxes.

Transfer function (SISO)

Transforms an input sequence $u(t)$ to an output $y(t)$ according to:

$$y(t) = G(q)u(t) = \frac{b_0 + b_1 q^{-1} + \cdots + b_{n_b} q^{-n_b}}{1 + a_1 q^{-1} + \cdots + a_{n_a} q^{-n_a}} u(t)$$

Equivalent to the recurrence equation:

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

For our purposes, G is a vector operator with coefficients a, b, transforming $\mathbf{u} \in \mathbb{R}^T$ to $\mathbf{y} \in \mathbb{R}^T$

$$\mathbf{y} = G(\mathbf{u}; a, b)$$

Our goal is to provide G with a back-propagation behavior.
Then, we can use G within a Deep Learning optimization engine!

Transfer function (SISO)

Transforms an input sequence $u(t)$ to an output $y(t)$ according to:

$$y(t) = G(q)u(t) = \frac{b_0 + b_1 q^{-1} + \cdots + b_{n_b} q^{-n_b}}{1 + a_1 q^{-1} + \cdots + a_{n_a} q^{-n_a}} u(t)$$

Equivalent to the recurrence equation:

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

For our purposes, G is a vector operator with coefficients a, b, transforming $\mathbf{u} \in \mathbb{R}^T$ to $\mathbf{y} \in \mathbb{R}^T$

$$\mathbf{y} = G(\mathbf{u}; a, b)$$

Our goal is to provide G with a back-propagation behavior.
Then, we can use G within a Deep Learning optimization engine!

Transfer function (SISO)

Transforms an input sequence $u(t)$ to an output $y(t)$ according to:

$$y(t) = G(q)u(t) = \frac{b_0 + b_1 q^{-1} + \cdots + b_{n_b} q^{-n_b}}{1 + a_1 q^{-1} + \cdots + a_{n_a} q^{-n_a}} u(t)$$

Equivalent to the recurrence equation:

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

For our purposes, G is a vector operator with coefficients a, b, transforming $\mathbf{u} \in \mathbb{R}^T$ to $\mathbf{y} \in \mathbb{R}^T$

$$\mathbf{y} = G(\mathbf{u}; a, b)$$

Our goal is to provide G with a back-propagation behavior.
Then, we can use G within a Deep Learning optimization engine!

Forward pass

In back-propagation-based training, the user defines the network's computational graph producing a loss $\mathcal{L}$ (to be minimized).

In the forward pass, the loss $\mathcal{L}$ is computed.
G receives $\mathbf{u}$, a, and b and needs to compute $\mathbf{y}$:

$$\mathbf{y} = G.\mathrm{forward}(\mathbf{u}; a, b).$$

$$\begin{tikzcd} & a, b \arrow[dr] & & \\ \cdots\ \mathbf{u} \arrow[rr, "G"] & & \mathbf{y} \arrow[r] & \cdots\ \mathcal{L} \end{tikzcd}$$

The forward pass for G is easy: it is just the filtering operation!

Forward pass

In back-propagation-based training, the user defines the network's computational graph producing a loss $\mathcal{L}$ (to be minimized).

In the forward pass, the loss $\mathcal{L}$ is computed.
G receives $\mathbf{u}$, a, and b and needs to compute $\mathbf{y}$:

$$\mathbf{y} = G.\mathrm{forward}(\mathbf{u}; a, b).$$

$$\begin{tikzcd} & a, b \arrow[dr] & & & \\ \cdots \ \mathbf{u} \arrow[rr, "G"] & & \mathbf{y} \arrow[r] & \cdots \ \mathcal{L} \end{tikzcd}$$

The forward pass for G is easy: it is just the filtering operation!

Backward pass

- In the backward pass, derivatives of $\mathcal{L}$ w.r.t. the training variables are computed. Notation: $\overline{x} = \frac{\partial \mathcal{L}}{\partial x}$.
- The procedure starts from $\overline{\mathcal{L}} \equiv 1$ and goes backward.
- Each operator must be able to "push back" derivatives from its outputs to its inputs

G receives $\overline{\mathbf{y}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$ and is responsible for computing: $\overline{\mathbf{u}}, \overline{a}, \overline{b}$:

$$\overline{\mathbf{u}}, \overline{a}, \overline{b} = G.\mathrm{backward}(\overline{\mathbf{y}}; a, b).$$

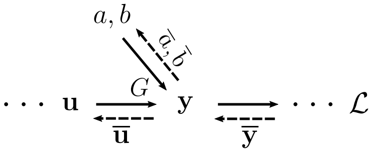
By defining these backward operations, we can use G in Deep Learning!
All technical details are in the *dynoNet* arXiv paper...

Backward pass

- In the backward pass, derivatives of $\mathcal{L}$ w.r.t. the training variables are computed. Notation: $\overline{x} = \frac{\partial \mathcal{L}}{\partial x}$.
- The procedure starts from $\overline{\mathcal{L}} \equiv 1$ and goes backward.
- Each operator must be able to "push back" derivatives from its outputs to its inputs

G receives $\overline{\mathbf{y}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$ and is responsible for computing: $\overline{\mathbf{u}}, \overline{a}, \overline{b}$:

$$\overline{\mathbf{u}}, \overline{a}, \overline{b} = G.\mathrm{backward}(\overline{\mathbf{y}}; a, b).$$



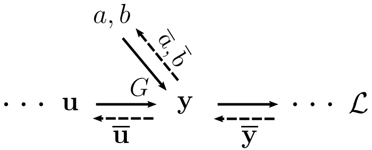
By defining these backward operations, we can use G in Deep Learning!
All technical details are in the *dynoNet* arXiv paper...

Backward pass

- In the backward pass, derivatives of $\mathcal{L}$ w.r.t. the training variables are computed. Notation: $\overline{x} = \frac{\partial \mathcal{L}}{\partial x}$.
- The procedure starts from $\overline{\mathcal{L}} \equiv 1$ and goes backward.
- Each operator must be able to "push back" derivatives from its outputs to its inputs

G receives $\overline{\mathbf{y}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$ and is responsible for computing: $\overline{\mathbf{u}}, \overline{a}, \overline{b}$:

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



By defining these backward operations, we can use G in Deep Learning!
All technical details are in the *dynoNet* arXiv paper...

Backward pass for $\mathbf{u}$

From $\overline{\mathbf{y}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$, compute $\overline{\mathbf{u}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{u}}$.

- Using the chain rule:

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

- From the expression above, by definition:

$$\overline{\mathbf{u}}_\tau = \sum_{t=0}^{T-1} \overline{\mathbf{y}}_t \mathbf{g}_{t-\tau} = \mathbf{g} \star \overline{\mathbf{y}},$$

  where $\mathbf{g}$ contains the impulse response coefficients and $\star$ is the cross-correlation. This is already a valid formula, but it is $\mathcal{O}(T^2)$

- The formula above is equivalent to applying the filter in reverse time!

$$\overline{\mathbf{u}} = \text{flip}\big(G(q)\text{flip}(\overline{\mathbf{y}})\big)$$

  Implemented this way, the cost is $\mathcal{O}(T)$!

Backward pass for $\mathbf{u}$

From $\overline{\mathbf{y}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$, compute $\overline{\mathbf{u}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{u}}$.

- Using the chain rule:

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

- From the expression above, by definition:

$$\overline{\mathbf{u}}_\tau = \sum_{t=0}^{T-1} \overline{\mathbf{y}}_t \mathbf{g}_{t-\tau} = \mathbf{g} \star \overline{\mathbf{y}},$$

  where $\mathbf{g}$ contains the impulse response coefficients and $\star$ is the cross-correlation. This is already a valid formula, but it is $\mathcal{O}(T^2)$

- The formula above is equivalent to applying the filter in reverse time!

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

  Implemented this way, the cost is $\mathcal{O}(T)$!

Backward pass for $\mathbf{u}$

From $\overline{\mathbf{y}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$, compute $\overline{\mathbf{u}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{u}}$.

- Using the chain rule:

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

- From the expression above, by definition:

$$\overline{\mathbf{u}}_\tau = \sum_{t=0}^{T-1} \overline{\mathbf{y}}_t \mathbf{g}_{t-\tau} = \mathbf{g} \star \overline{\mathbf{y}},$$

  where $\mathbf{g}$ contains the impulse response coefficients and $\star$ is the cross-correlation. This is already a valid formula, but it is $\mathcal{O}(T^2)$

- The formula above is equivalent to applying the filter in reverse time!

$$\overline{\mathbf{u}} = \text{flip}\big(G(q)\text{flip}(\overline{\mathbf{y}})\big)$$

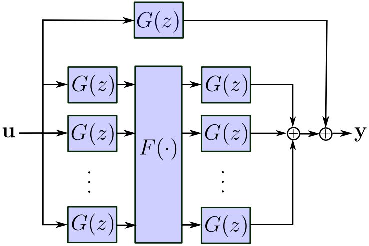
  Implemented this way, the cost is $\mathcal{O}(T)$!

PyTorch implementation

PyTorch implementation of the G-block in the repository `https://github.com/forgi86/dynonet`.

Use case:

dynoNet architecture



Python code

```
G1 = LinearMimo(1,  4, ...) # a G-block
F1 = StaticNonLin(4, 8, ...) # a NN?
G2 = LinearMimo(8, 1, ...)
G3 = LinearMimo(1, 1, ...)

def model():
    y1 = G1(u)
    z1 = F1(y1)
    y2 = G2(z1)
    ymodel = y2 + G3(u)
```

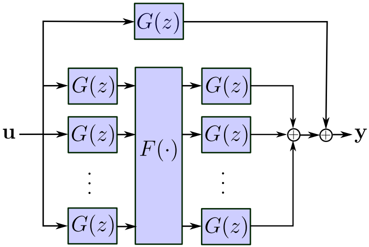
Any gradient-based optimization algorithm can be used to train the *dynoNet* with derivatives readily obtained by back-propagation.

PyTorch implementation

PyTorch implementation of the G-block in the repository
`https://github.com/forgi86/dynonet`.

Use case:

dynoNet architecture



Python code

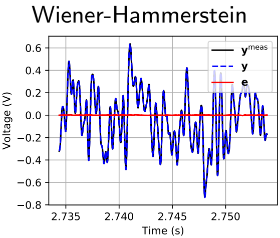
```
G1 = LinearMimo(1,  4, ...) # a G-block
F1 = StaticNonLin(4, 8, ...) # a NN?
G2 = LinearMimo(8, 1, ...)
G3 = LinearMimo(1, 1, ...)

def model():
    y1 = G1(u)
    z1 = F1(y1)
    y2 = G2(z1)
    ymodel = y2 + G3(u)
```

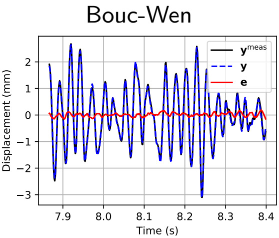
Any gradient-based optimization algorithm can be used to train the *dynoNet* with derivatives readily obtained by back-propagation.

Experimental results

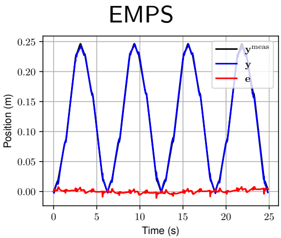
Numerical experiments on public system identification benchmark available at `www.nonlinearbenchmark.org`.



$\text{fit} = 99.5\%$



$\text{fit} = 93.2\%$



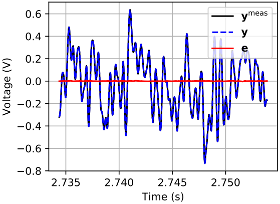
$\text{fit} = 96.8\%$

Compare favorably with state-of-the-art black-box identification techniques.

Experimental results

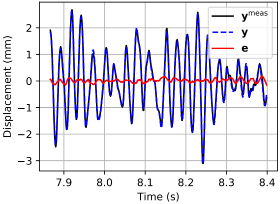
Numerical experiments on public system identification benchmark available at `www.nonlinearbenchmark.org`.

Wiener-Hammerstein



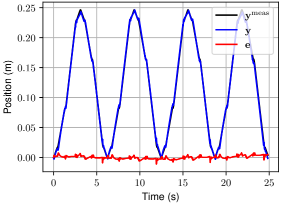
$\text{fit} = 99.5\%$

Bouc-Wen



$\text{fit} = 93.2\%$

EMPS



$\text{fit} = 96.8\%$

Compare favorably with state-of-the-art black-box identification techniques.

Conclusions

A neural network architecture containing linear dynamical operators parametrized as rational transfer functions.

- Extends 1D-Convolutional NNs to Infinite Impulse Response dynamics
- Extends block-oriented dynamical models with generic interconnections
- Enables training through plain back-propagation. No custom algorithm/code required

Future works:

- Estimation/control strategies
- System analysis/model reduction using e.g. linear tools

Conclusions

A neural network architecture containing linear dynamical operators parametrized as rational transfer functions.

- Extends 1D-Convolutional NNs to Infinite Impulse Response dynamics
- Extends block-oriented dynamical models with generic interconnections
- Enables training through plain back-propagation. No custom algorithm/code required

Future works:

- Estimation/control strategies
- System analysis/model reduction using e.g. linear tools

Thank you.
Questions?

marco.forgione@idsia.ch