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

Convegno Nazionale Automatica
Cagliari, 9–11 Settembre 2020

Motivations

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

Recurrent NNs

General state-space models

- High representational capacity
- Hard to parallelize
- Numerical issues in training

1D Convolutional NNs

Dynamics through FIR blocks

- Lower capacity, several params
- Fully parallelizable
- Fast, well-behaved training

We introduce *dynoNet*: an architecture using linear dynamical operators parametrized as rational transfer functions as building blocks.

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

Motivations

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

Recurrent NNs

General state-space models

- High representational capacity
- Hard to parallelize
- Numerical issues in training

1D Convolutional NNs

Dynamics through FIR blocks

- Lower capacity, several params
- Fully parallelizable
- Fast, well-behaved training

We introduce *dynoNet*: an architecture using linear dynamical operators parametrized as rational transfer functions as building blocks.

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

Motivations

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

Recurrent NNs

General state-space models

- High representational capacity
- Hard to parallelize
- Numerical issues in training

1D Convolutional NNs

Dynamics through FIR blocks

- Lower capacity, several params
- Fully parallelizable
- Fast, well-behaved training

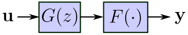
We introduce *dynoNet*: an architecture using linear dynamical operators parametrized as rational transfer functions as building blocks.

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

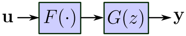
Related works

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

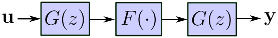
Wiener



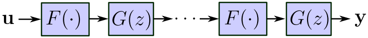
Hammerstein



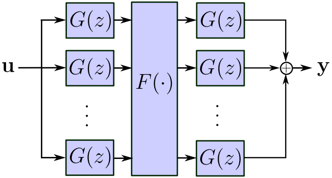
Wiener-Hammerstein



Generalized Hammerstein-Wiener



Parallel Wiener-Hammerstein

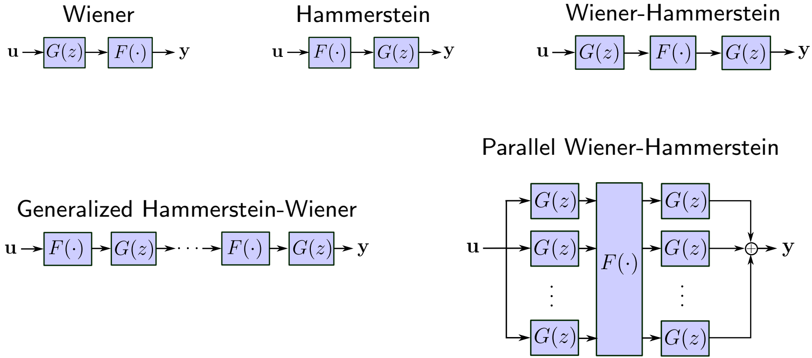


extensively studied in System Identification.

Training with 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 $F(\cdot)$:

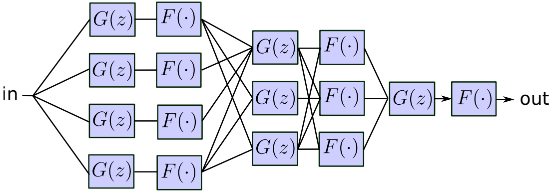


extensively studied in System Identification.
Training with specialized algorithms requiring, e.g. analytic expressions of gradients/jacobians.

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 exploiting Deep Learning software

a *dynoNet* network

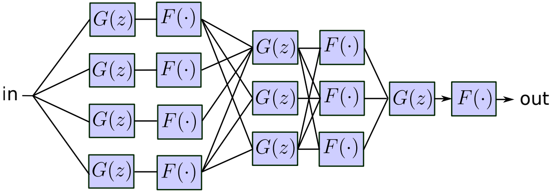


Technical challenge: back-propagation through the transfer function!
No hint in the literature, no ready-made implementation available.

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 exploiting Deep Learning software

a *dynoNet* network



Technical challenge: back-propagation through the transfer function!
No hint in the literature, no ready-made implementation available.

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.
The operation has to be efficient!

Transfer function (SISO)

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

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

Equivalent to the recurrence equation:

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

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.
The operation has to be efficient!

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.
The operation has to be efficient!

Forward pass

In back-propagation-based training, the user defines a 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!

Computational cost: $\mathcal{O}(T)$.

Forward pass

In back-propagation-based training, the user defines a 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!

Computational cost: $\mathcal{O}(T)$.

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 $\overline{\mathcal{L}} \equiv \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$ 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: $\overline{\mathbf{u}}, \overline{a}, \overline{b}$:

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

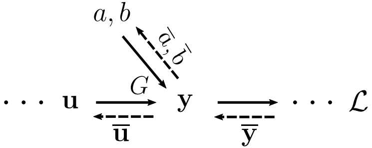
Chain rule of calculus is the basic ingredient, but certain tricks may be used to speed up the operation. Let us see an example...

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 $\overline{\mathcal{L}} \equiv \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$ 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: $\overline{\mathbf{u}}, \overline{a}, \overline{b}$:

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



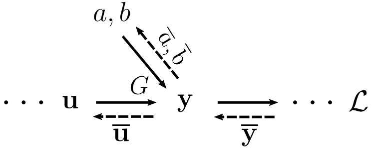
Chain rule of calculus is the basic ingredient, but certain tricks may be used to speed up the operation. Let us see an example...

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 $\overline{\mathcal{L}} \equiv \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$ 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: $\overline{\mathbf{u}}, \overline{a}, \overline{b}$:

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



Chain rule of calculus is the basic ingredient, but certain tricks may be used to speed up the operation. Let us see an example...

Backward pass for $\mathbf{u}$

Compute $\overline{\mathbf{u}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{u}}$ from $\overline{\mathbf{y}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$.

- Applying the chain rule:

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

  where g is the impulse response of G.
- From the expression above, by definition:

$$\overline{\mathbf{u}} = \mathbf{g} \star \overline{\mathbf{y}},$$

  where $\star$ is cross-correlation. This implementation has cost $\mathcal{O}(T^2)$
- It is equivalent to filtering $\overline{\mathrm{y}}$ through G in reverse time, and flipping the result. Implemented this way, the cost is $\mathcal{O}(T)$!

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

All details also for $\overline{a}$ and $\overline{b}$ in the *dynoNet* arXiv paper...

Backward pass for $\mathbf{u}$

Compute $\overline{\mathbf{u}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{u}}$ from $\overline{\mathbf{y}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$.

- Applying the chain rule:

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

  where $\mathbf{g}$ is the impulse response of G.

- From the expression above, by definition:

$$\overline{\mathbf{u}} = \mathbf{g} \star \overline{\mathbf{y}},$$

  where $\star$ is cross-correlation. This implementation has cost $\mathcal{O}(T^2)$

- It is equivalent to filtering $\overline{\mathbf{y}}$ through G in reverse time, and flipping the result. Implemented this way, the cost is $\mathcal{O}(T)$!

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

All details also for $\overline{a}$ and $\overline{b}$ in the *dynoNet* arXiv paper. . .

Backward pass for $\mathbf{u}$

Compute $\overline{\mathbf{u}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{u}}$ from $\overline{\mathbf{y}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$.

- Applying the chain rule:

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

  where $\mathbf{g}$ is the impulse response of G.

- From the expression above, by definition:

$$\overline{\mathbf{u}} = \mathbf{g} \star \overline{\mathbf{y}},$$

  where $\star$ is cross-correlation. This implementation has cost $\mathcal{O}(T^2)$

- It is equivalent to filtering $\overline{\mathrm{y}}$ through G in reverse time, and flipping the result. Implemented this way, the cost is $\mathcal{O}(T)$!

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

All details also for $\overline{a}$ and $\overline{b}$ in the *dynoNet* arXiv paper. . .

Backward pass for $\mathbf{u}$

Compute $\overline{\mathbf{u}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{u}}$ from $\overline{\mathbf{y}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$.

- Applying the chain rule:

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

  where $\mathbf{g}$ is the impulse response of G.

- From the expression above, by definition:

$$\overline{\mathbf{u}} = \mathbf{g} \star \overline{\mathbf{y}},$$

  where $\star$ is cross-correlation. This implementation has cost $\mathcal{O}(T^2)$

- It is equivalent to filtering $\overline{\mathbf{y}}$ through G in reverse time, and flipping the result. Implemented this way, the cost is $\mathcal{O}(T)$!

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

All details also for $\overline{a}$ and $\overline{b}$ in the *dynoNet* arXiv paper...

Backward pass for $\mathbf{u}$

Compute $\overline{\mathbf{u}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{u}}$ from $\overline{\mathbf{y}} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$.

- Applying the chain rule:

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

  where $\mathbf{g}$ is the impulse response of G.

- From the expression above, by definition:

$$\overline{\mathbf{u}} = \mathbf{g} \star \overline{\mathbf{y}},$$

  where $\star$ is cross-correlation. This implementation has cost $\mathcal{O}(T^2)$

- It is equivalent to filtering $\overline{\mathbf{y}}$ through G in reverse time, and flipping the result. Implemented this way, the cost is $\mathcal{O}(T)$!

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

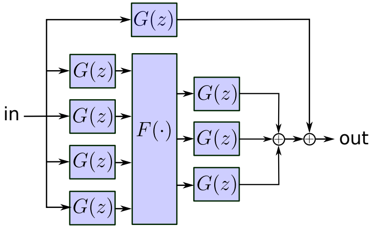
All details also for $\overline{a}$ and $\overline{b}$ in the *dynoNet* arXiv paper...

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 SIMO tf
F = StaticNonLin(4, 3, ...) # a static NN
G2 = LinearMimo(3, 1, ...) # a MISO tf
G3 = LinearMimo(1, 1, ...) # a SISO tf

def model(in_data):
    y1 = G1(in_data)
    z1 = F(y1)
    y2 = G2(z1)
    out = y2 + G3(in_data)
```

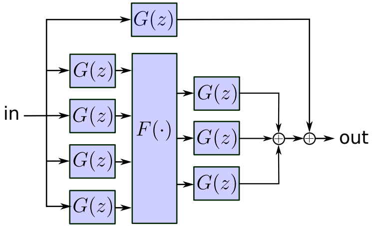
Any gradient-based optimization algorithm can be used to train the *dynoNet* with derivatives readily obtained by back-propagation.

PyTorch implementation

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

Use case:

dynoNet architecture



Python code

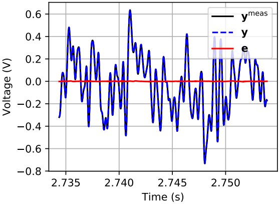
```
G1 = LinearMimo(1,  4, ...) # a SIMO tf
F = StaticNonLin(4, 3, ...) # a static NN
G2 = LinearMimo(3, 1, ...) # a MISO tf
G3 = LinearMimo(1, 1, ...) # a SISO tf

def model(in_data):
    y1 = G1(in_data)
    z1 = F(y1)
    y2 = G2(z1)
    out = y2 + G3(in_data)
```

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

Experimental results

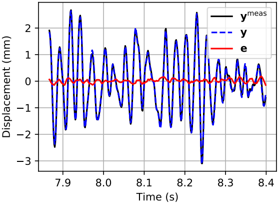
Numerical experiments on public system identification benchmark available at `www.nonlinearbenchmark.org`.

Wiener-Hammerstein



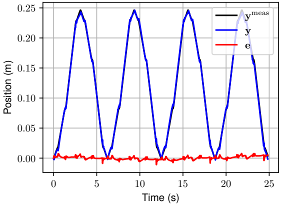
$\mathrm{fit} = 99.5\%$

Bouc-Wen



$\mathrm{fit} = 93.2\%$

EMPS



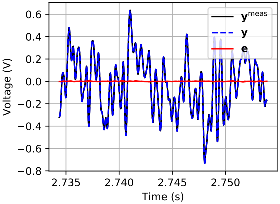
$\mathrm{fit} = 96.8\%$

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

Experimental results

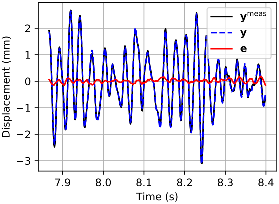
Numerical experiments on public system identification benchmark available at `www.nonlinearbenchmark.org`.

Wiener-Hammerstein



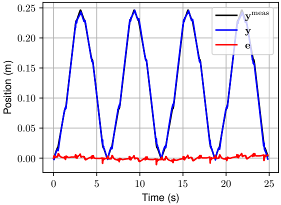
$\text{fit} = 99.5\%$

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 arbitrary interconnections
- Enables training through plain back-propagation, at cost $\mathcal{O}(T)$. No custom algorithm/code required

Current and future work:

- 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 arbitrary interconnections
- Enables training through plain back-propagation, at cost $\mathcal{O}(T)$. No custom algorithm/code required

Current and future work:

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

Thank you.
Questions?

marco.forgione@idsia.ch