Deep learning with transfer functions: New applications in system identification

Dario Piga, Marco Forgione, Manas Mejari

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

19th IFAC symposium System Identification: learning models for decision and control

Motivations

Two main classes of neural network structures for 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
- Well-posed training

We have recently introduced *dynoNet*: an architecture using linear transfer functions as building blocks.

M. Forgione and D. Piga. *dynoNet*: A Neural Network architecture for learning dynamical systems. *International Journal of Adaptive Control and Signal Processing*, 2021

Motivations

Two main classes of neural network structures for 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
- Well-posed training

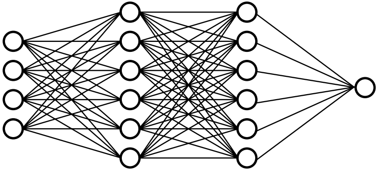
We have recentily introduced *dynoNet*: an architecture using linear transfer functions as building blocks.

M. Forgione and D. Piga. *dynoNet*: A Neural Network architecture for learning dynamical systems. *International Journal of Adaptive Control and Signal Processing*, 2021

The *dynoNet* architecture

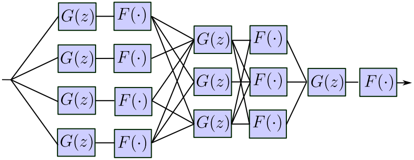
Deep learning with the LTI transfer functions.

A feed-forward neural network



$\Rightarrow$

A dynoNet

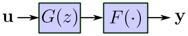


- An extension of feed-forward neural networks, with dynamical blocks
- An extension of 1D CNNs, with Infinite Impulse Response dynamics
- An extension of block-oriented models, with arbitrary connections

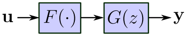
Related works

Block-oriented architectures extensively studied in System Identification. Interconnection of transfer functions $G(z)$ and static non-linearities $F(\cdot)$:

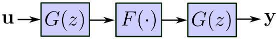
Wiener



Hammerstein

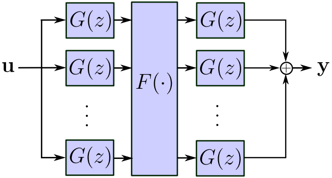


Wiener-Hammerstein



Generalized Hammerstein-Wiener

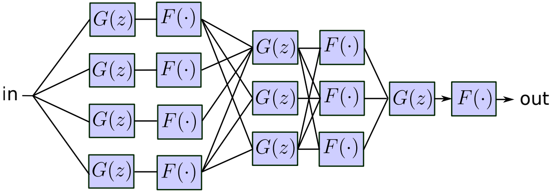
Parallel Wiener-Hammerstein



Related works

- *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

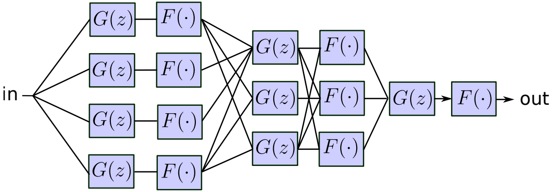


Technical challenge: back-propagation through the transfer function!
No hint in the literature, no ready-made implementation available.

Related works

- *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

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

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

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 $\boldsymbol{u} \in \mathbb{R}^N$ to $\boldsymbol{y} \in \mathbb{R}^N$

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

Our goal is to provide G with a back-propagation behavior.
The operation has to be efficient!

Transfer function

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

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

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 $\boldsymbol{u} \in \mathbb{R}^N$ to $\boldsymbol{y} \in \mathbb{R}^N$

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

Our goal is to provide G with a back-propagation behavior.
The operation has to be efficient!

Transfer function

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

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

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 $\boldsymbol{u} \in \mathbb{R}^N$ to $\boldsymbol{y} \in \mathbb{R}^N$

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

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 $\boldsymbol{u}$, a, and b and needs to compute $\boldsymbol{y}$:

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

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

The forward pass for G is easy: it is just the linear recurrence equation:

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

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

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 $\boldsymbol{u}$, a, and b and needs to compute $\boldsymbol{y}$:

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

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

The forward pass for G is easy: it is just the linear recurrence equation:

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

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

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

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

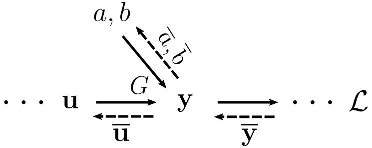
The chain rule is the basic tool, but certain tricks may be used to speed up the operation. Computational cost also $\mathcal{O}(N)$, all details in the paper...

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 $\overline{\mathcal{L}} \equiv \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$ and goes backward.
- Each operator must be able to "push back" derivatives from its outputs to its inputs

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

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



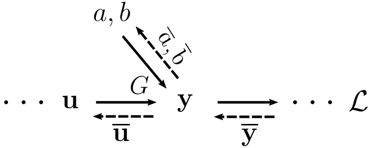
The chain rule is the basic tool, but certain tricks may be used to speed up the operation. Computational cost also $\mathcal{O}(N)$, all details in the paper...

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 $\overline{\mathcal{L}} \equiv \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$ and goes backward.
- Each operator must be able to "push back" derivatives from its outputs to its inputs

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

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



The chain rule is the basic tool, but certain tricks may be used to speed up the operation. Computational cost also $\mathcal{O}(N)$, all details in the paper...

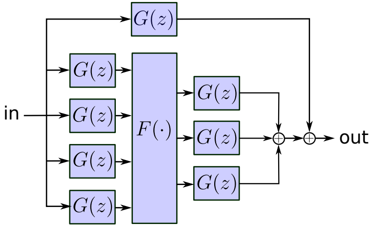
PyTorch implementation

https://github.com/forgi86/dynonet

https://github.com/forgi86/sysid-transfer-functions-pytorch

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*. The derivatives are obtained through back-propagation.

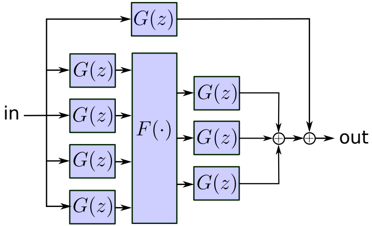
PyTorch implementation

`https://github.com/forgi86/dynonet`

`https://github.com/forgi86/sysid-transfer-functions-pytorch`

Use case:

dynoNet architecture



Python code

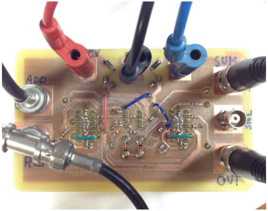
```
G1 = LinearMimo(1,  4, ...) # a SIMO tf
F = StaticNonLin(4, 3, ...) # a static NN
G2 = LinearMimo(3, 1, ...) # a MISO tf
G3 = LinearMimo(1, 1, ...) # a SISO tf

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

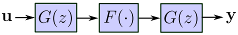
Any gradient-based optimization algorithm can be used to train the *dynoNet*. The derivatives are obtained through back-propagation.

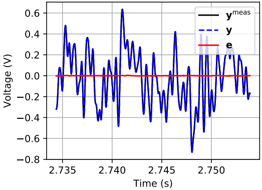
Experimental results

On public benchmarks at www.nonlinearbenchmark.org.

Wiener-Hammerstein System





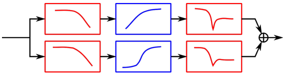


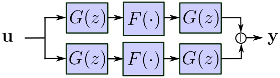
Test performance: $\text{fit} = 99.5\%$

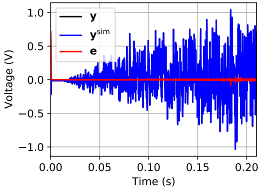
Experimental results

On public benchmarks at `www.nonlinearbenchmark.org`.

Parallel Wiener-Hammerstein





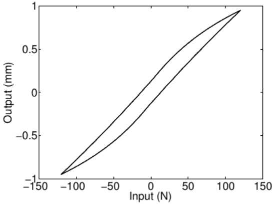


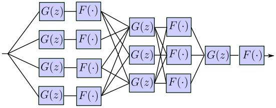
Test performance: $\text{fit} = 98.4\%$

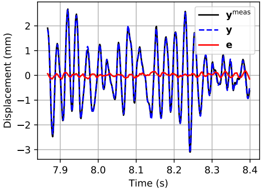
Experimental results

On public benchmarks at `www.nonlinearbenchmark.org`.

Bouc-Wen Hysteretic System







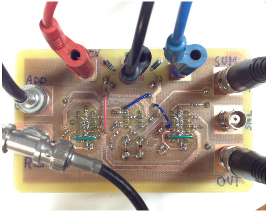
Test performance: $\mathrm{fit} = 93.2\%$

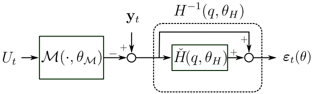
Experimental results

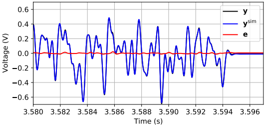
On public benchmarks at `www.nonlinearbenchmark.org`.

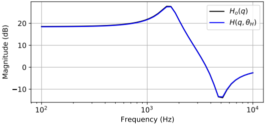
Wiener-Hammerstein System

modified training dataset, additive colored noise on the output









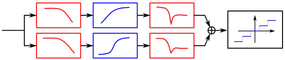
Training with WH model + noise whitening filter (Neural PEM)

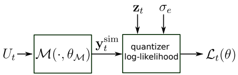
Test performance: $\text{fit} = 96.9\%$

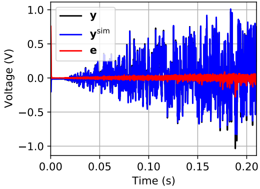
Experimental results

On public benchmarks at `www.nonlinearbenchmark.org`.

Parallel Wiener-Hammerstein
modified training dataset, 5-level quantized measurements







Training with a loss corresponding to ML for quantized measurements:
Test performance: $\text{fit} = 91.9\%$

Conclusions

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

- Extends 1D-Convolutional NNs and block-oriented models
- Training through plain back-propagation, at cost $\mathcal{O}(N)$
- Implementation available on-line

Current and future work:

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

Conclusions

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

- Extends 1D-Convolutional NNs and block-oriented models
- Training through plain back-propagation, at cost $\mathcal{O}(N)$
- Implementation available on-line

Current and future work:

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

Thank you.
Questions?

marco.forgione@idsia.ch

Backward pass for $\boldsymbol{u}$

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

- Applying the chain rule:

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

  where $\boldsymbol{g}$ is the impulse response of G.
- From the expression above, by definition:

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

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

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

All details (also for $\overline{a}$ and $\overline{b}$) in the paper...

Backward pass for $\boldsymbol{u}$

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

- Applying the chain rule:

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

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

- From the expression above, by definition:

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

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

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

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

All details (also for $\overline{a}$ and $\overline{b}$) in the paper...

Backward pass for $\boldsymbol{u}$

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

- Applying the chain rule:

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

  where $\boldsymbol{g}$ is the impulse response of G.
- From the expression above, by definition:

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

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

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

All details (also for $\overline{a}$ and $\overline{b}$) in the paper...

Backward pass for $\boldsymbol{u}$

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

- Applying the chain rule:

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

  where $\boldsymbol{g}$ is the impulse response of G.
- From the expression above, by definition:

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

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

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

All details (also for $\overline{a}$ and $\overline{b}$) in the paper...

Backward pass for $\boldsymbol{u}$

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

- Applying the chain rule:

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

  where $\boldsymbol{g}$ is the impulse response of G.
- From the expression above, by definition:

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

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

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

All details (also for $\overline{a}$ and $\overline{b}$) in the paper...

Backward pass for b

Compute $\overline{b} \equiv \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}}$ from $\overline{\boldsymbol{y}} \equiv \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}}$.

- Applying the chain rule:

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

- The forward sensitivities $\frac{\partial \boldsymbol{y}_t}{\partial b_j}$ may be obtained in closed form through additional filtering operations. From the definition:

$$A(q)y(t) = B(q)u(t)$$

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

Differentiating w.r.t. b_j:

$$\frac{\partial y(t)}{\partial b_j} + a_1 \frac{\partial y(t-1)}{\partial b_j} \cdots + a_{n_a} \frac{\partial y(t-n_a)}{\partial b_j} = u(t-j)$$

$$A(q)\frac{\partial y(t)}{\partial b_j} = u(t-j)$$

Actually, just one filtering is required as $\frac{\partial y(t)}{\partial b_j} = \frac{\partial y(t-j)}{\partial b_0}$

Backward pass for b

Compute $\overline{b} \equiv \frac{\partial \mathcal{L}}{\partial b}$ from $\overline{\boldsymbol{y}} \equiv \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}}$.

- Applying the chain rule:

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

- The forward sensitivities $\frac{\partial \boldsymbol{y}_t}{\partial b_j}$ may be obtained in closed form through additional filtering operations. From the definition:

$$A(q)y(t) = B(q)u(t)$$

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

Differentiating w.r.t. b_j:

$$\frac{\partial y(t)}{\partial b_j} + a_1 \frac{\partial y(t-1)}{\partial b_j} \cdots + a_{n_a} \frac{\partial y(t-n_a)}{\partial b_j} = u(t-j)$$

$$A(q)\frac{\partial y(t)}{\partial b_j} = u(t-j)$$

Actually, just one filtering is required as $\frac{\partial y(t)}{\partial b_j} = \frac{\partial y(t-j)}{\partial b_0}$

Backward pass for b

Compute $\overline{b} \equiv \frac{\partial \mathcal{L}}{\partial b}$ from $\overline{\boldsymbol{y}} \equiv \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}}$.

- Applying the chain rule:

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

- The forward sensitivities $\frac{\partial \boldsymbol{y}_t}{\partial b_j}$ may be obtained in closed form through additional filtering operations. From the definition:

$$A(q)y(t) = B(q)u(t)$$

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

Differentiating w.r.t. b_j:

$$\frac{\partial y(t)}{\partial b_j} + a_1 \frac{\partial y(t-1)}{\partial b_j} \cdots + a_{n_a} \frac{\partial y(t-n_a)}{\partial b_j} = u(t-j)$$

$$A(q) \frac{\partial y(t)}{\partial b_j} = u(t-j)$$

Actually, just one filtering is required as $\frac{\partial y(t)}{\partial b_j} = \frac{\partial y(t-j)}{\partial b_0}$

Backward pass for b

Compute $\overline{b} \equiv \frac{\partial \mathcal{L}}{\partial b}$ from $\overline{\boldsymbol{y}} \equiv \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}}$.

- Applying the chain rule:

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

- The forward sensitivities $\frac{\partial \boldsymbol{y}_t}{\partial b_j}$ may be obtained in closed form through additional filtering operations. From the definition:

$$A(q)y(t) = B(q)u(t)$$

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

Differentiating w.r.t. b_j:

$$\frac{\partial y(t)}{\partial b_j} + a_1 \frac{\partial y(t-1)}{\partial b_j} \cdots + a_{n_a} \frac{\partial y(t-n_a)}{\partial b_j} = u(t-j)$$

$$A(q) \frac{\partial y(t)}{\partial b_j} = u(t-j)$$

Actually, just one filtering is required as $\frac{\partial y(t)}{\partial b_j} = \frac{\partial y(t-j)}{\partial b_0}$

Backward pass for b

Compute $\overline{b} \equiv \frac{\partial \mathcal{L}}{\partial b}$ from $\overline{\boldsymbol{y}} \equiv \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}}$.

- Applying the chain rule:

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

- The forward sensitivities $\frac{\partial \boldsymbol{y}_t}{\partial b_j}$ may be obtained in closed form through additional filtering operations. From the definition:

$$A(q)y(t) = B(q)u(t)$$

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

Differentiating w.r.t. b_j:

$$\frac{\partial y(t)}{\partial b_j} + a_1 \frac{\partial y(t-1)}{\partial b_j} \cdots + a_{n_a} \frac{\partial y(t-n_a)}{\partial b_j} = u(t-j)$$

$$A(q) \frac{\partial y(t)}{\partial b_j} = u(t-j)$$

Actually, just one filtering is required as $\frac{\partial y(t)}{\partial b_j} = \frac{\partial y(t-j)}{\partial b_0}$