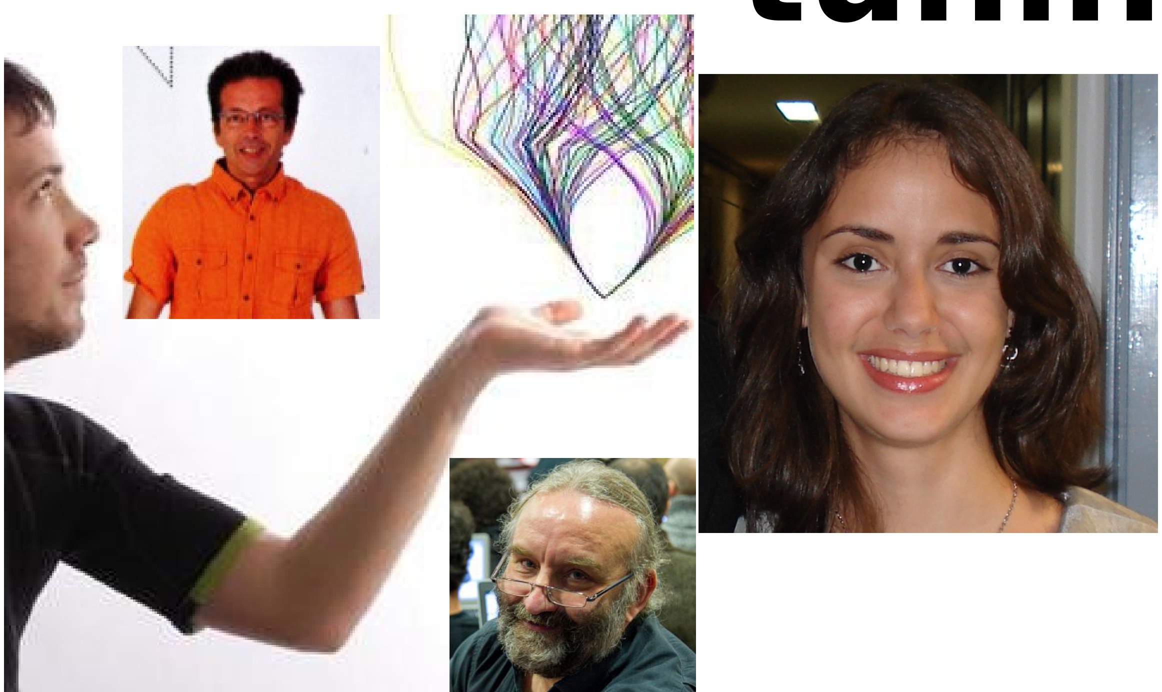
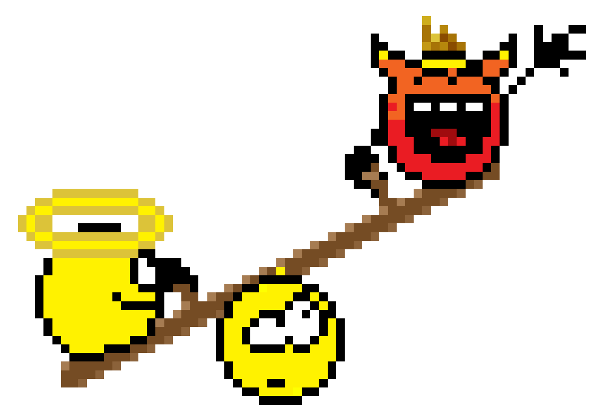
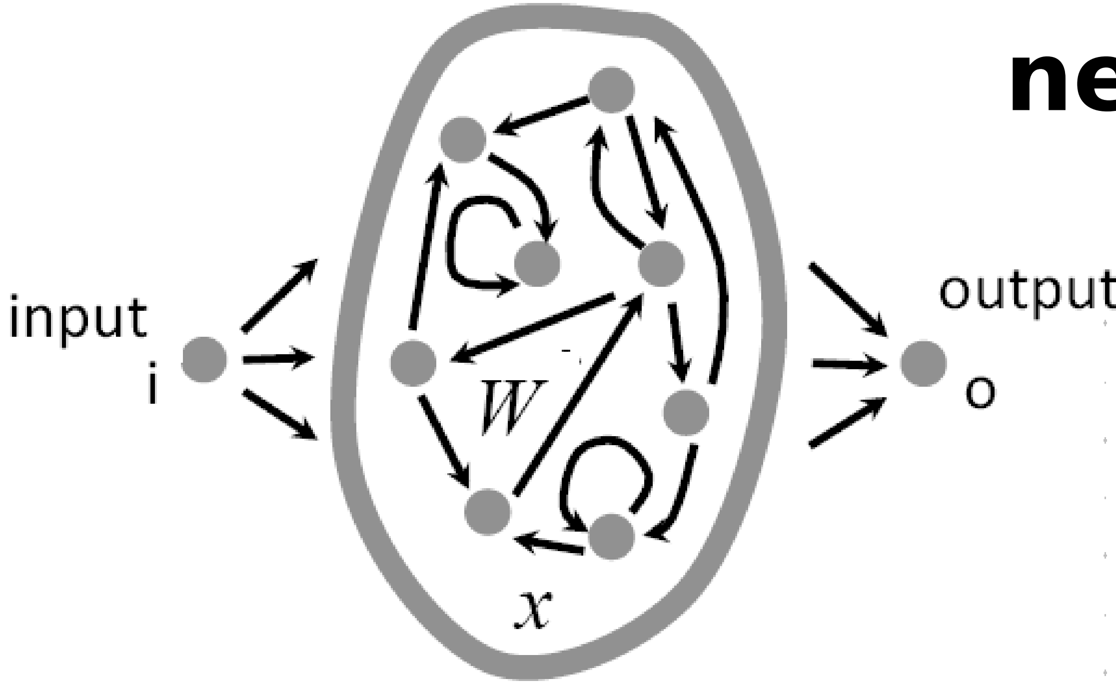
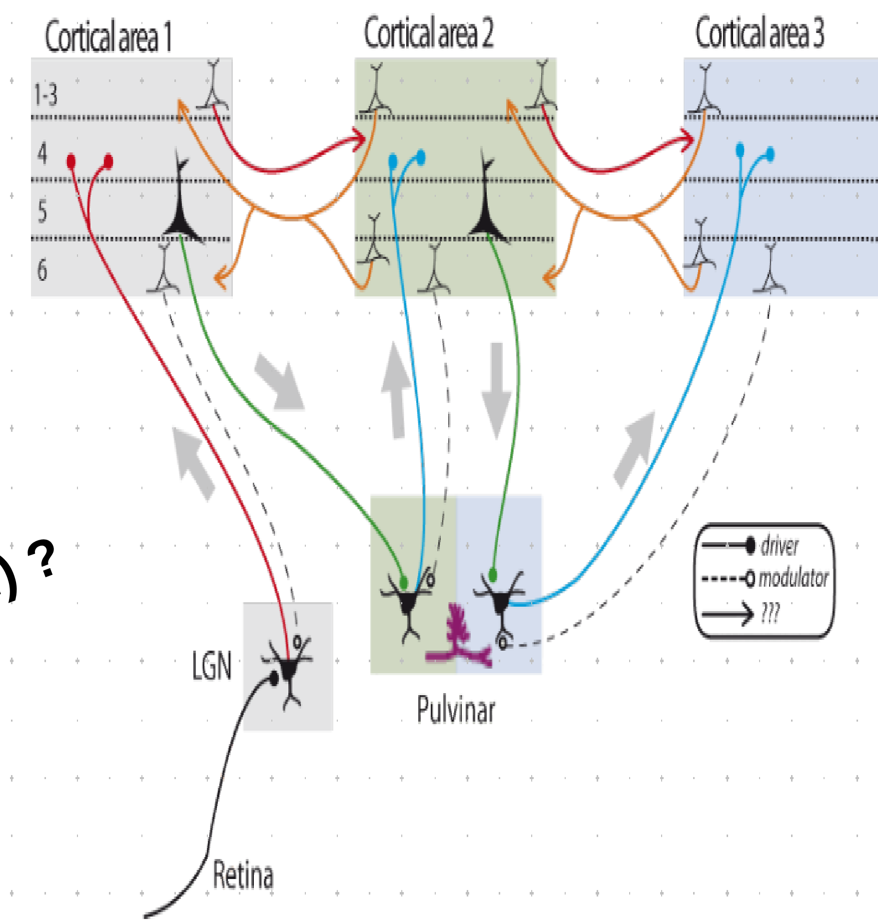
Recurrent neural network weight estimation though backward tuning





Recurrent neural network weight estimation



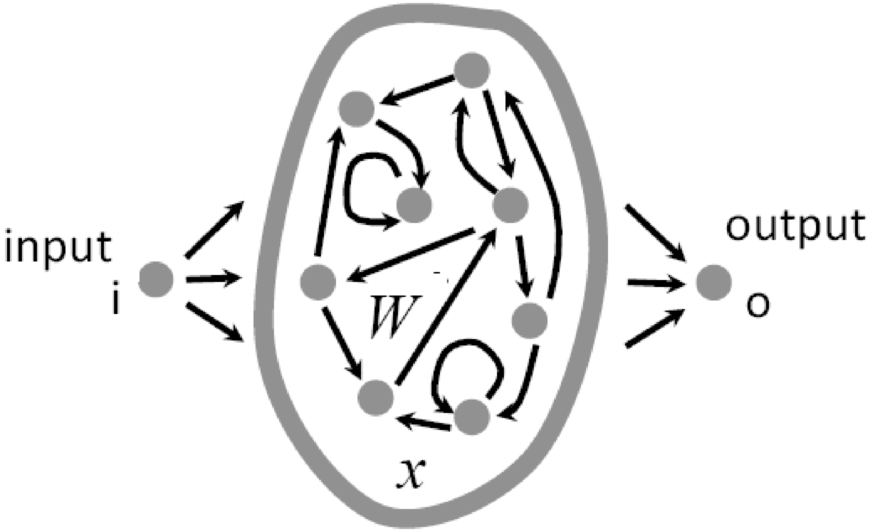


Still unsolved problem in the general case !

Feed-forward deep-learning ?

Only readout tuned reservoir computing ?

Predefined architecture (Elman, LSTM, RBM) ?



A new notation

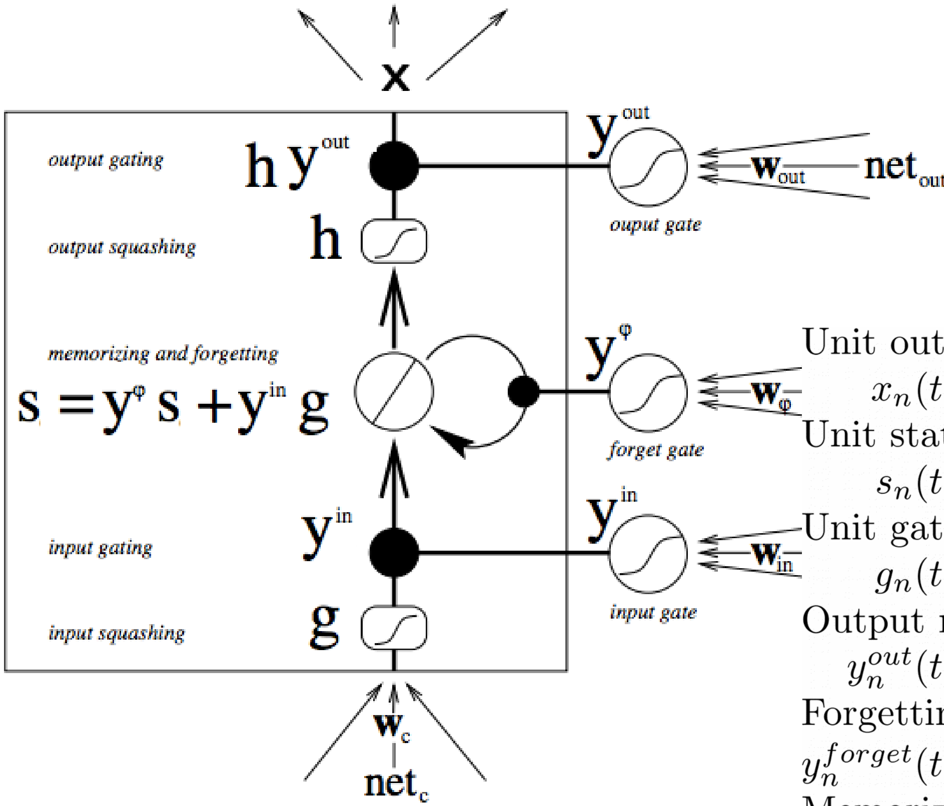
$$
\begin{aligned}
x_n(t) &= \Phi_{n0t}\left(\cdots, x_{n'}(t'), \cdots, i_m(s), \cdots\right) \\
&+ \textstyle\sum_{d=1}^{D_n} W_{nd}\, \Phi_{ndt}\left(\cdots, x_{n'}(t'), \cdots, i_m(s), \cdots\right) \\
o_n(t) &= x_n(t), n < N_0
\end{aligned}
$$

+ Unit firmware $\Phi_0()$ **and unit learnware** $\sum_d W_d\, \Phi_d()$

+ Intermediate variables to obtain a linear problem

+ Homogeneous notion of hidden/output units

A general notation



Unit output:

$$x_n(t) = \zeta_{[0,1]}\left(y_n^{out}(t)\right)\,\zeta_{[-1,1]}\left(s_n(t)\right)$$

Unit state:

$$s_n(t) = \zeta_{[0,1]}\left(y_n^{forget}(t)\right)\,s_n(t-1) + \zeta_{[0,1]}\left(y_n^{in}(t)\right)\,\zeta_{[-1,1]}\left(g_n(t)\right)$$

Unit gate:

$$g_n(t) = \sum_{n'} W^g_{nn'}\,x_{n'}(t-1) + \sum_m W^g_{nm}\,i_m(t-1)$$

Output modulation:

$$y_n^{out}(t) = W^o_n\,s_n(t-1) + \sum_{n'} W^o_{nn'}\,y^c_{n'}(t-1) + \sum_m W^o_{nm}\,i_m(t-1)$$

Forgetting modulation:

$$y_n^{forget}(t) = W^f_n\,s_n(t-1) + \sum_{n'} W^f_{nn'}\,y^c_{n'}(t-1) + \sum_m W^f_{nm}\,i_m(t-1)$$

Memorizing modulation:

$$y_n^{in}(t) = W^i_n\,s_n(t-1) + \sum_{n'} W^i_{nn'}\,y^c_{n'}(t-1) + \sum_m W^i_{nm}\,i_m(t-1)$$

+ Any unit (LSTM, SoftMax, LNL, Spiking, …) fits

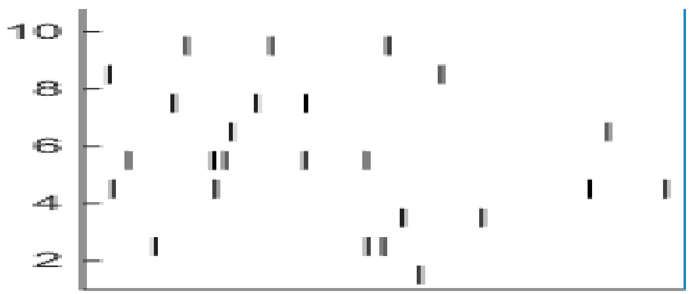
+ Includes adjustable leak, variable architecture, …

+ Simply requires the definition of $\Phi()$ and $\partial\Phi()$

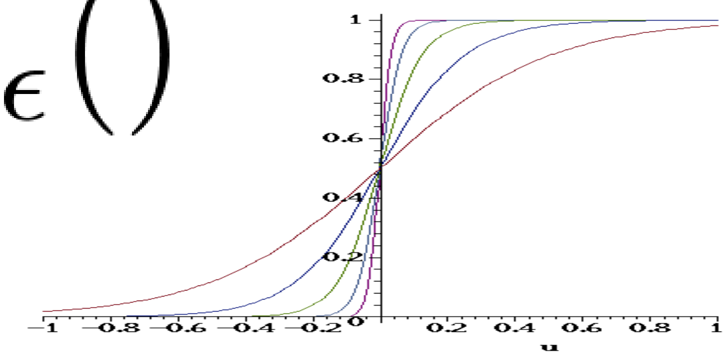
A general notation

+ Includes event based units

$$
\begin{aligned}
x_n(t) &= \gamma_n \left(\left(1 - H_\epsilon\left(x_n(t-1) - 1\right)\right) x_n(t-1)\right. \\
&+ \textstyle\sum_{n'=0}^{N} W_{nn'}\, H_\epsilon\left(x_{n'}(t-1) - 1\right) + \sum_{m=0}^{N} W_{nm}\, i_m(t-1),
\end{aligned}
$$



$$H_\epsilon()$$



+ Approximate spiking unit, with adjustable sharpness

+ Yields trivial definition of « complex functions »
- **latches and constant state memory,**
- **long term transform,**
- **conditional sequence generation,**
- **...**

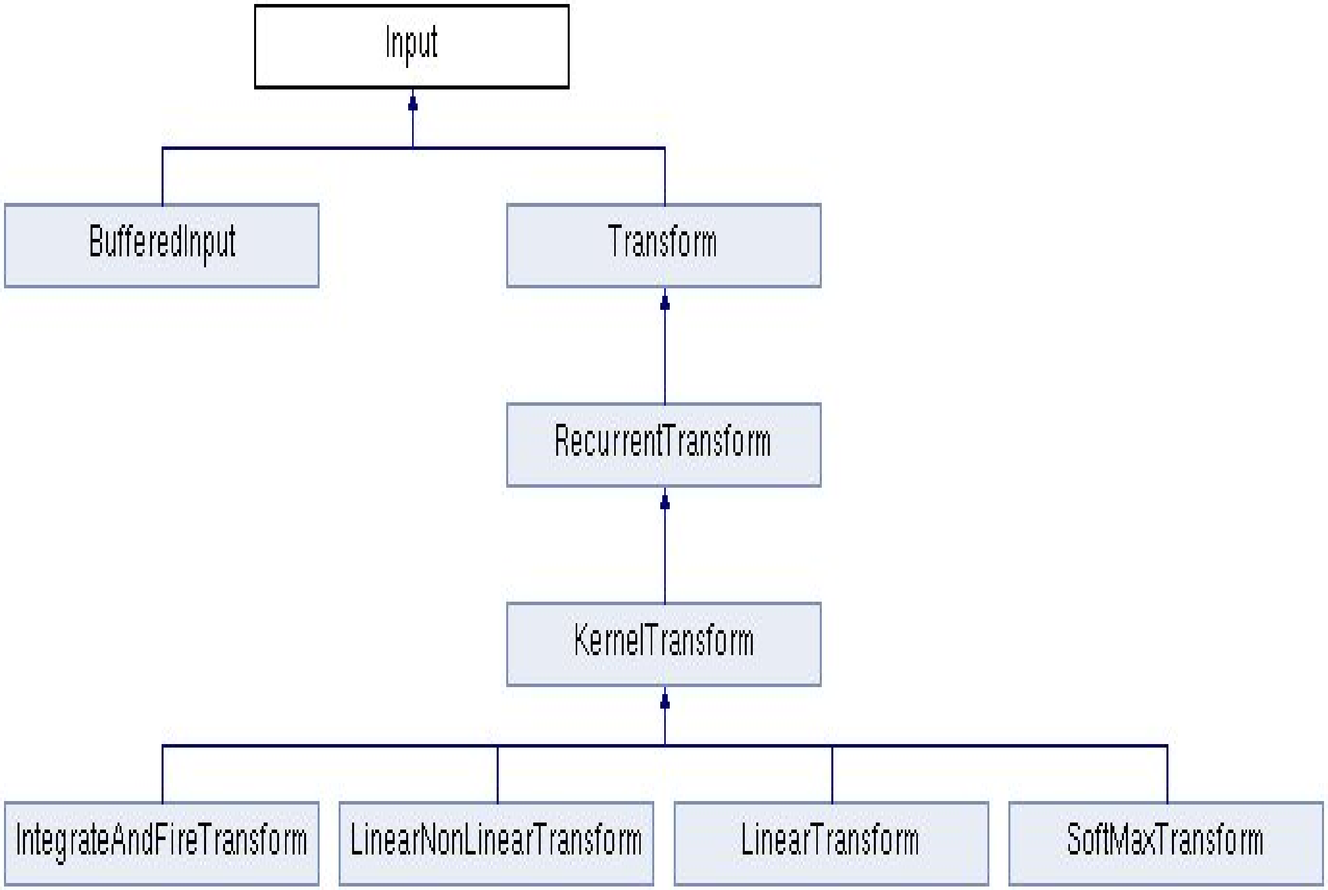
Mnemonas'semiware

Implementation of a Mnemonas semiware

A simple implementation

https://vthierry.github.io/mnemonas

virtual unsigned int	**getKernelDimension** (unsigned int n) const Returns the recurrent kernel dimension. More...
virtual double	**getKernelValue** (unsigned int n, unsigned int d, double t) const Returns the recurrent kernel value. More...
virtual double	**getKernelDerivative** (unsigned int n, unsigned int d, double t, unsigned int n_, double t_) const Returns the recurrent kernel derivative with respect to a recurrent state value. More...



A simple implementation

```
unsigned int getKernelDimension(unsigned int n) const
{
  return n < N ? 2 + N + input.getN() : 1;
}
double getKernelValue(unsigned int n, unsigned int d, double t) const
{
  if(n < N) {
    if(d == 0)
      return 0;
    if(d == 1)
      return (1 - zeta(get(n + N, t))) * get(n, t - 1);
    if(d == 2)
      return 1;
    d -= 3;
    if(d < N)
      return zeta(get(d + N, t));
    d -= N;
    if(d < input.getN())
      return input.get(d, t - 1);
    return 0;
  } else
    return d == 1 ? get(n - N, t - 1) - 1 : 0;
}
double getKernelDerivative(unsigned int n, unsigned int d, double t, unsigned int n_, double t_) const
{
  return n < N ?
           (d == 1 ?
            (n_ == n && t_ == t - 1 ? 1 - zeta(get(n + N, t)) :
             n_ == n + N && t_ == t ? -dzeta(get(n_, t)) * get(n, t - 1) : 0) :
            3 <= d && d < N + 3 && n_ == N + d - 3 && t_ == t ? dzeta(get(n_, t)) : 0) :
    d == 1 && n_ == n - N && t_ == t - 1 ? 1 : 0;
}
```

$$
\begin{aligned}
x_n(t) &= \gamma_n \left[\left(1 - \zeta_{[0,1]}(x_{n_1}(t))\right) x_n(t-1) \right] \\
&+ \alpha_n + \textstyle\sum_{n'=0}^{N} W_{nn'}\, \zeta_{[0,1]} \left(x_{n'_1}(t-1) \right) \\
&+ \textstyle\sum_{m=0}^{N} W_{nm}\, i_m(t-1) \\
x_{n_1}(t) &= \frac{1}{\epsilon} \left(x_n(t-1) - 1 \right)
\end{aligned}
$$

$$\mathbf{W} = \arg\min_{\mathbf{W},\mathbf{x}} \mathcal{L}(\mathbf{W},\mathbf{x})$$

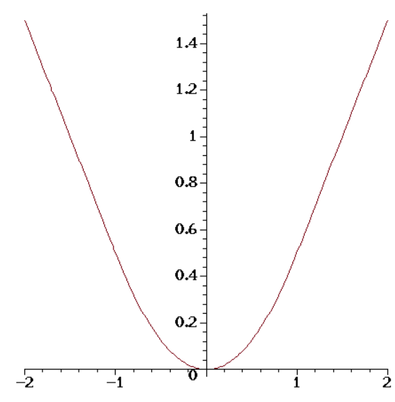
A clean estimation

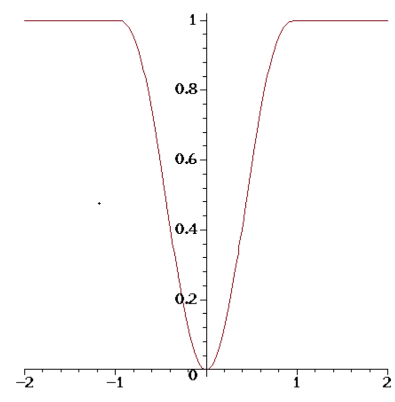
$$\begin{array}{llll} \mathcal{L}(\mathbf{W},\mathbf{x}) & & & \\ & \sum_{nt} & \rho_{nt}(x_n(t)) & \text{desired values} \\ - & \sum_{nt} & \varepsilon_{nt}\,(\hat{x}_n(t) - x_n(t)) & \text{network dynamic constraint} \\ + & \mathcal{R}(\mathbf{W}) & & \text{regularization} \end{array}$$

e.g. :

$$\rho_{nt}(\hat{x}_n(t)) = \frac{\kappa_{nt}}{2}(\hat{x}_n(t) - \bar{o}_n(t))^2$$

Robust profile $\rho_{nt}()$





Unsupervised criterion

+ sparseness

+ orthogonality

+ …/…

Sparse weights

$$\mathcal{R}(\mathbf{W}) = \sum_{nd} \frac{\nu_{nd}}{\epsilon + |\hat{W}_{nd}|} W_{nd}^2$$

A clean estimation

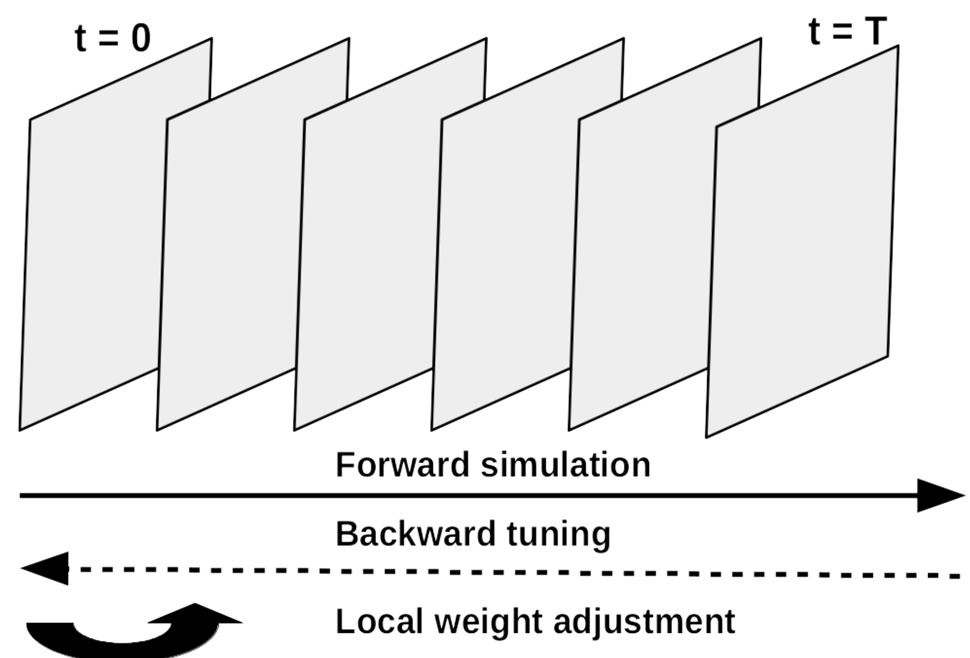
The three steps

1 → **forward simulation**

2 → **backward tuning**

3 → **weight adjustment**

from the normal equations.



+ distributed local computation

+ closed form at each step

+ relaxation between steps 2 & 3

$$\nabla_{\varepsilon_{nt}} \mathcal{L} = \hat{x}_n(t) - x_n(t)$$

$$\nabla_{x_{n'}(t')} \mathcal{L} = -\varepsilon_{n't'} + \rho'_{n't'} + \sum_{\substack{nt,\ t' < t \leq t' + R \\ \text{or } t' = t, n < n'}} \beta^{nt}_{n't'} \varepsilon_{nt}$$

$$\nabla_{W_{nd}} \mathcal{L} = \sum_{n'', W_{n''d} = W_{nd}} \sum_t \phi_{n''dt} \varepsilon_{n''t} + \nabla_{W_{nd}} \mathcal{R}$$

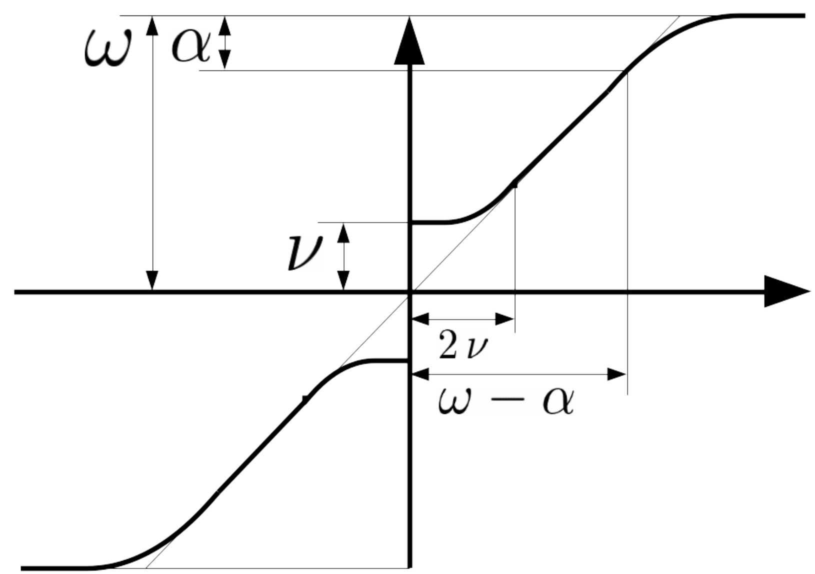
Backward tuning

A clean estimation

- **+ backward propagation of the output and hidden unit related error**
- **+ backward error vanishing or error explosion avoided**
- **+ simple linear calculation**

$$\varepsilon_{n't'} \simeq \rho'_{nt} + g\left(\sum_{nt} \beta^{nt}_{n't'}\, \varepsilon_{nt}\right)$$

tiny error bias + huge error saturation :



$$\beta^{nt}_{n't'} = \frac{\partial \phi_{n0t}}{\partial x_{n'}(t')} + \sum_{d=1}^{D_n} W_{nd}\, \frac{\partial \phi_{ndt}}{\partial x_{n'}(t')}$$

2nd order weight adjustment

A clean estimation

$$\nabla_{W_{nd}} \mathcal{L} = 0 \Rightarrow \sum_{n''} b_{n'',d} = \sum_{n''} \sum_{d'=1}^{D_n} A_{n'',d\,d'} W_{nd'}$$

$$\left\{ \begin{array}{rcl} b_{n,d} & = & \sum_t \phi_{ndt} \left(\varepsilon_{nt} + \kappa_{nt} \left(x_n(t) - \phi_{n0t} \right) \right) + \nabla_{W_{nd}} \mathcal{R}(\tilde{\mathbf{W}}), \\ A_{n,d\,d'} & = & \sum_t \kappa_{nt} \phi_{ndt} \phi_{nd't}. \end{array} \right. \qquad \kappa_{nt} = 1$$

+ just a linear system to solve at each step

+ can take weight sharing into account

- speed up by a line search between the current and next weights

- used a 1st order gradient descent as fallback

Implementation

It works

→ **a simple,
highly modular,
normalized*,
fully documented,
open source,
object oriented,
easily forkable,
python wrapped,
self contained,
middle-ware :**

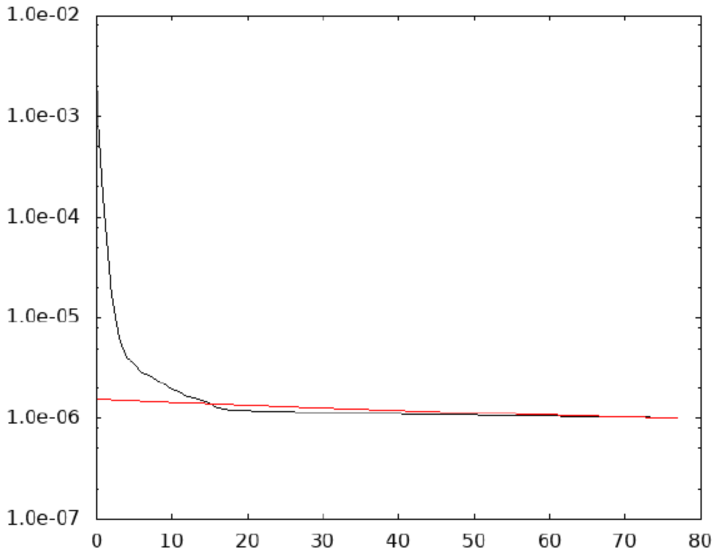
https://vthierry.github.io/mnemonas

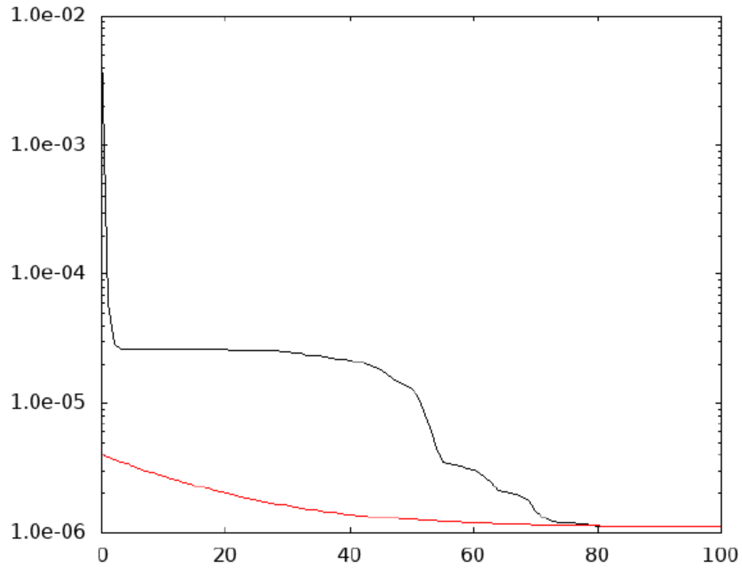
(*) using precise coding rules and beautified source code

Reverse engineering

It works

→ **a root network provides an input/output to be reproduced by a blind network**





Node type	LinearNonLinearTransform				
Number of units	2	4	8	16	32
Number of Iterations	36	101	78	T.B.D.	T.B.D.
Minimal criterion value	9.3e-07	3.0e-06	1.0e-06	T.B.D.	T.B.D.
Exponential decay time	24	23	88	T.B.D.	T.B.D.
Final bias interpolation	2.2e-08	2.6e-06	5.9e-07	T.B.D.	T.B.D.
Node type	IntegrateAndFireTransform				
Number of units	2	4	8	16	32
Number of Iterations	101	101	101	T.B.D.	T.B.D.
Minimal criterion value	9.9e-06	4.7e-06	1.1e-06	T.B.D.	T.B.D.
Exponential decay time	8	36	17	T.B.D.	T.B.D.
Final bias interpolation	9.1e-06	4.6e-06	1.1e-06	T.B.D.	T.B.D.

Input/output approximation

It works

→ **it works with**

different kind of nodes

several robuts criteria

→ **a non reproducable input/output can be approximated**

→ **it also allows global statistical estimation**

That’s all folks