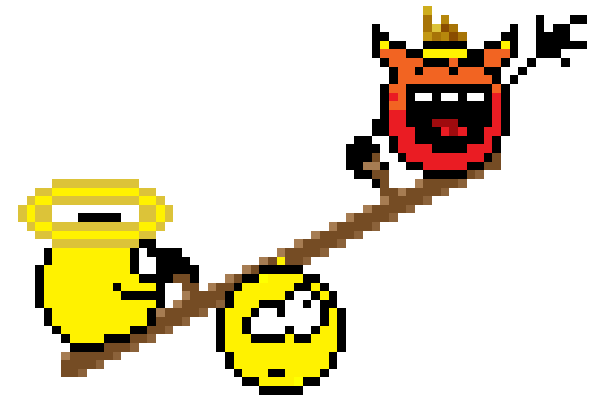
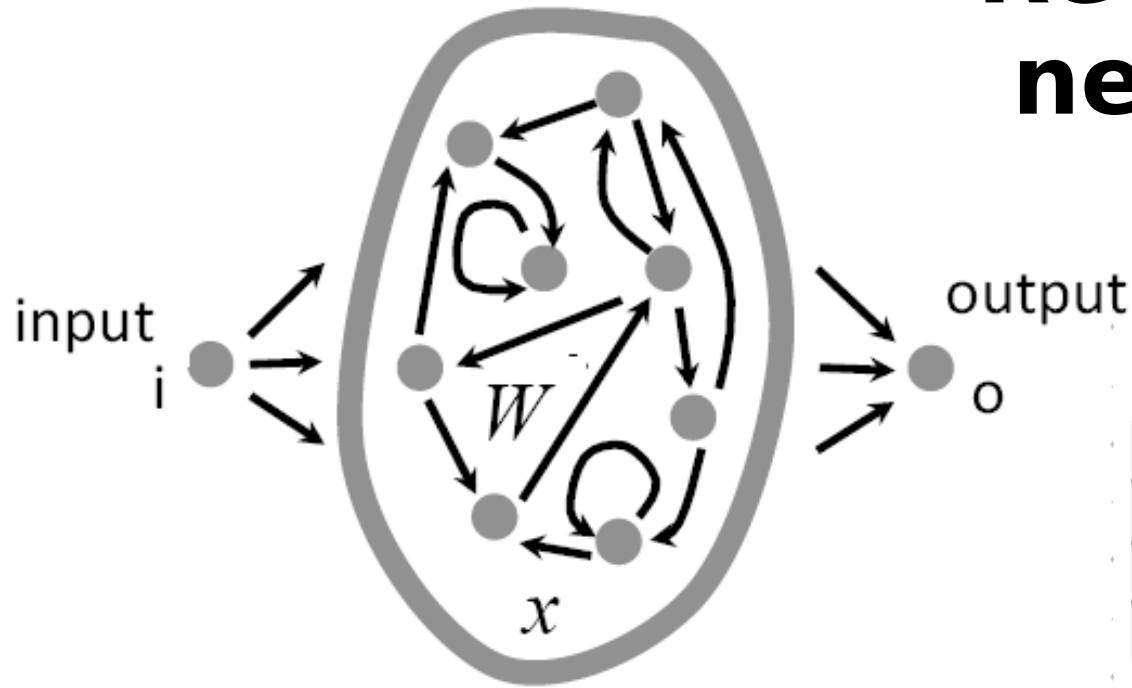


Recurrent neural network weight estimation though backward tuning

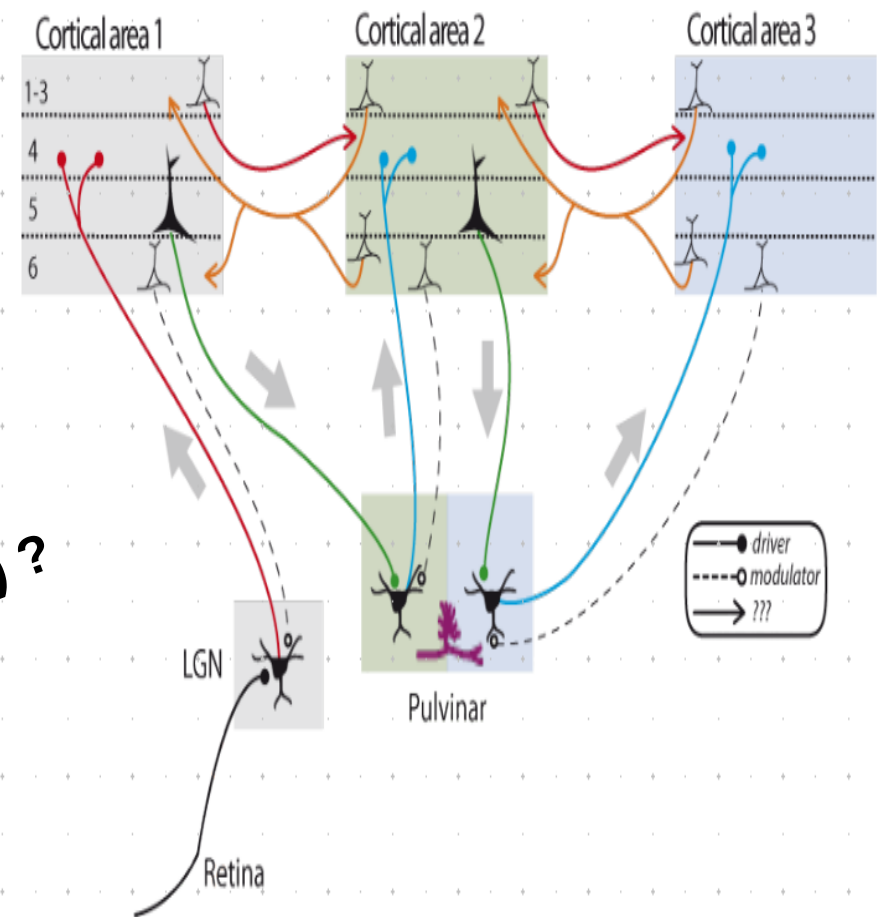


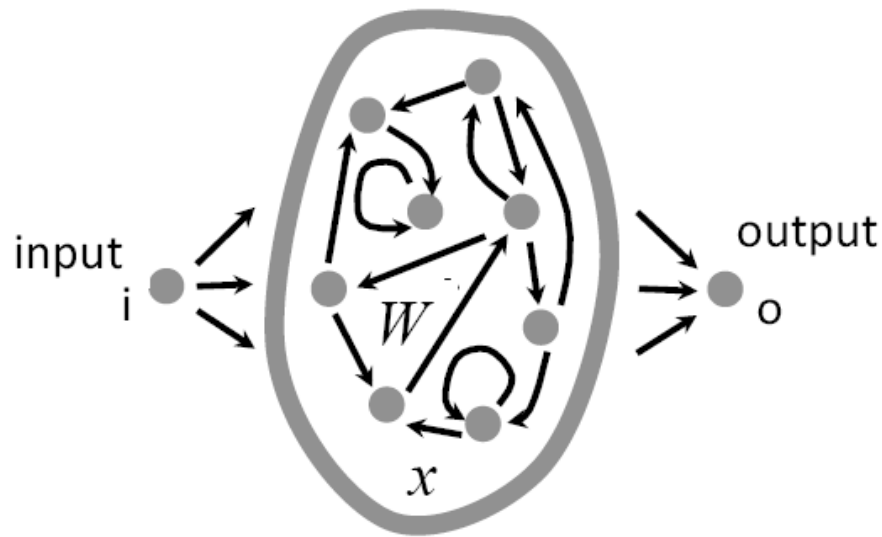
Recurrent neural network weight estimation



Still unsolved problem in the general case !

Feed-forward deep-learning ?
Only readout tuned reservoir computing ?
Predefined architecture (Elman, LSTM, RBM) ?

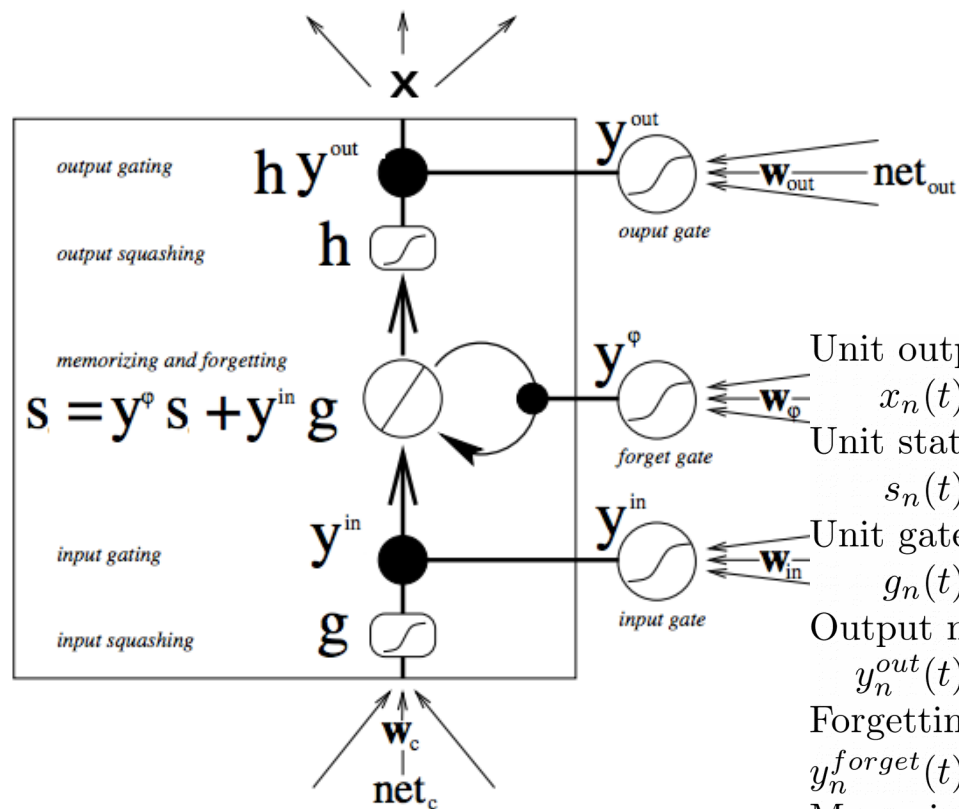




A new notation

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

- + 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:

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

Unit state:

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

Unit gate:

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

Output modulation:

$$y_n^{\text{out}}(t) = W_n^o s_n(t-1) + \sum_{n'} W_{nn'}^o y_{n'}^c(t-1) + \sum_m W_{nm}^o i_m(t-1)$$

Forgetting modulation:

$$y_n^{\text{forget}}(t) = W_n^f s_n(t-1) + \sum_{n'} W_{nn'}^f y_{n'}^c(t-1) + \sum_m W_{nm}^f i_m(t-1)$$

Memorizing modulation:

$$y_n^{\text{in}}(t) = W_n^i s_n(t-1) + \sum_{n'} W_{nn'}^i y_{n'}^c(t-1) + \sum_m W_{nm}^i i_m(t-1)$$

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

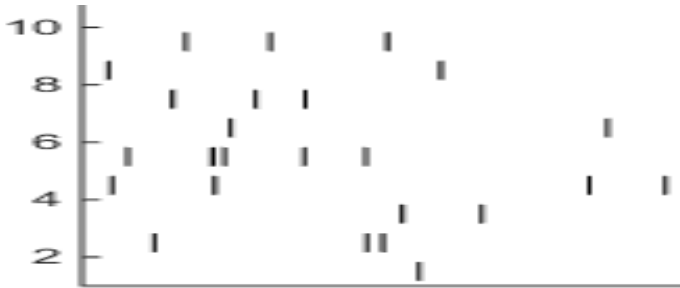
+ Includes adjustable leak, variable architecture, ...

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

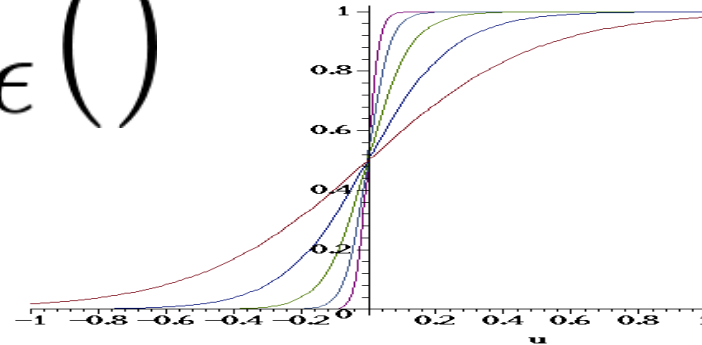
+ Includes event based units

A general notation

$$x_n(t) = \gamma_n ((1 - H_\epsilon(x_n(t-1) - 1)) x_n(t-1) + \sum_{n'=0}^N W_{nn'} H_\epsilon(x_{n'}(t-1) - 1) + \sum_{m=0}^N W_{nm} i_m(t-1)),$$



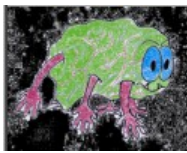
$H_\epsilon()$



+ 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

virtual unsigned int **getKernelDimension** (unsigned int n) const
Returns the recurrent kernel dimension. [More...](#)

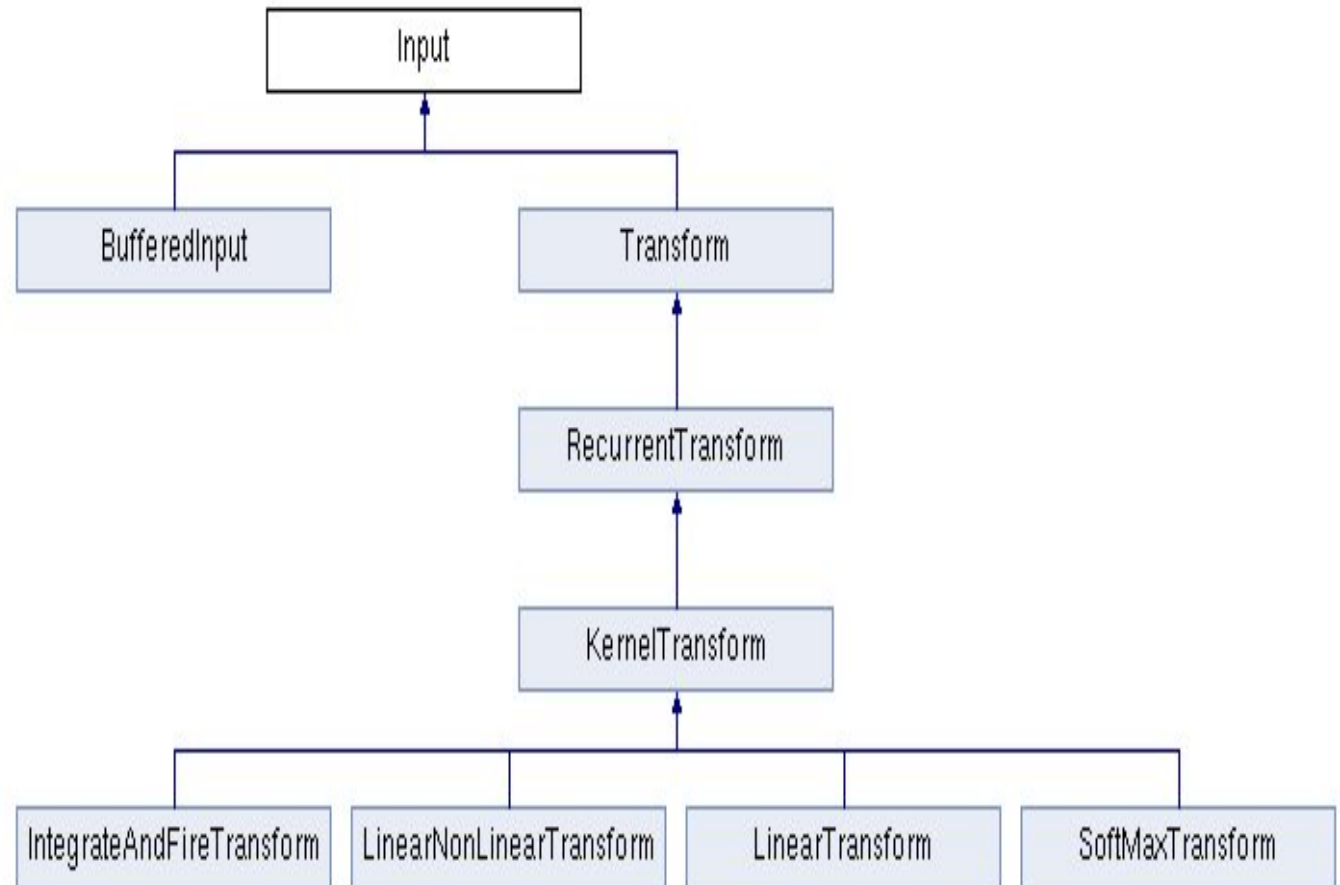
virtual double **getKernelValue** (unsigned int n, unsigned int d, double t) const
Returns the recurrent kernel value. [More...](#)

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

<https://vthierry.github.io/mnemonas>



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;
}
```

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

```
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;
}
```

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

A clean estimation

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

$$\begin{aligned} & \sum_{nt} \rho_{nt}(x_n(t)) \\ - & \sum_{nt} \varepsilon_{nt} (\hat{x}_n(t) - x_n(t)) \\ + & \mathcal{R}(\mathbf{W}) \end{aligned}$$

desired values

network dynamic constraint

regularization

e.g. :

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

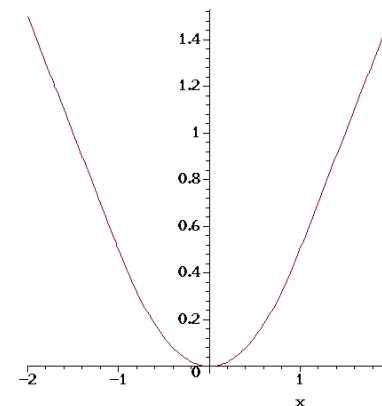
Unsupervised criterion

+ sparseness

+ orthogonality

+ .../...

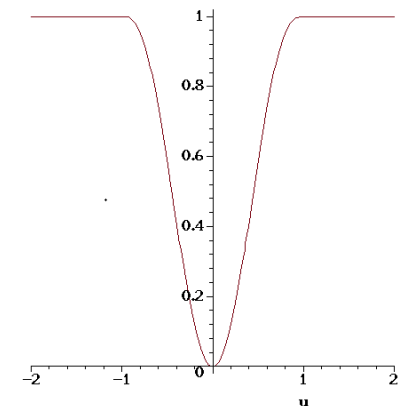
Robust profile



Sparse weights

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

$\rho_{nt}()$



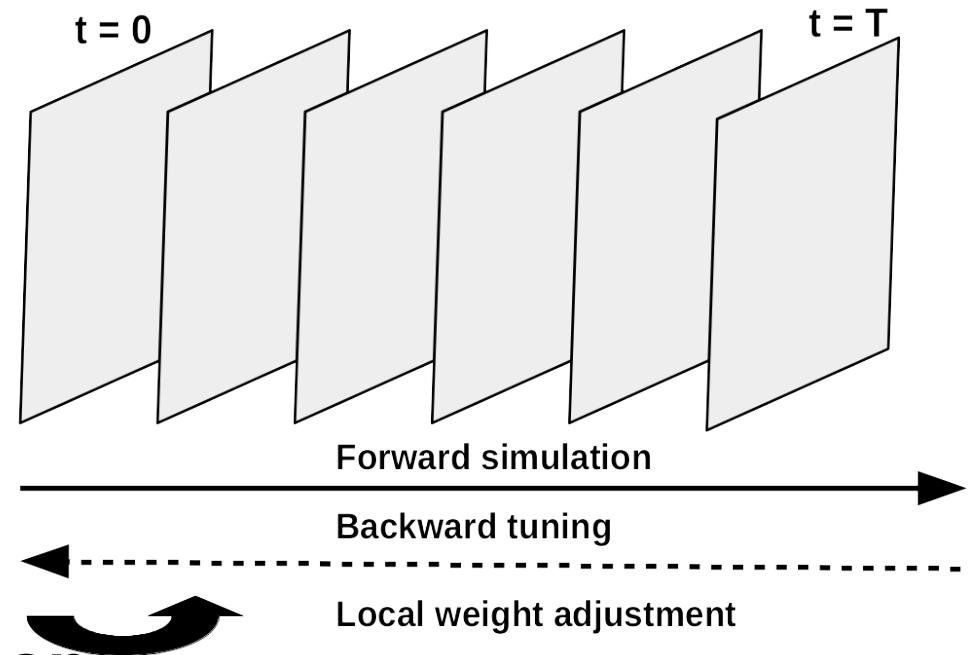
The three steps

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

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

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

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

A clean estimation

Backward tuning

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

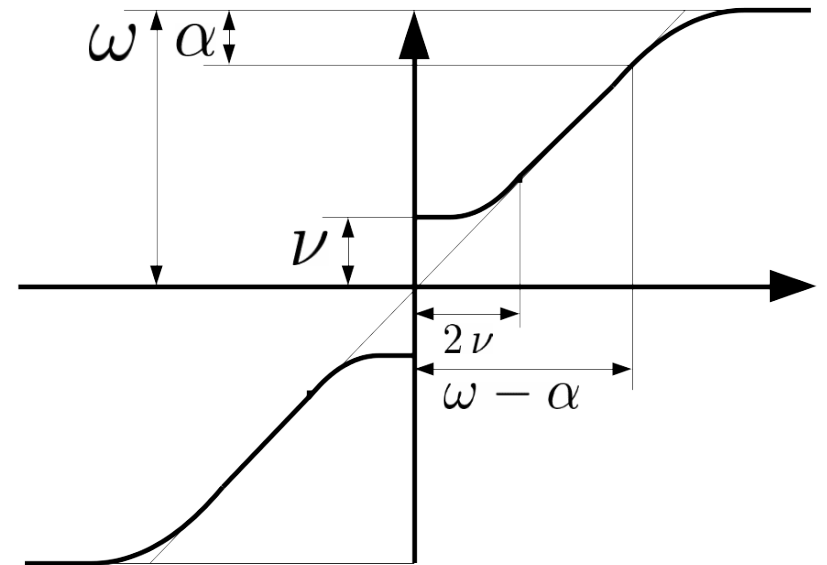
A clean estimation

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

tiny error bias + huge error saturation :



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



2nd order weight adjustment

A clean estimation

$$\nabla_{W_{nd}} \mathcal{L} = 0 \Rightarrow \sum_{n''} b_{n'', d} = \sum_{n''} \sum_{d'=1}^{D_n} A_{n'', d d'} W_{nd'}$$
$$\left\{ \begin{array}{l} b_{n, d} = \sum_t \phi_{ndt} (\varepsilon_{nt} + \kappa_{nt} (x_n(t) - \phi_{n0t})) + \nabla_{W_{nd}} \mathcal{R}(\tilde{\mathbf{W}}), \\ A_{n, d d'} = \sum_t \kappa_{nt} \phi_{ndt} \phi_{nd't}. \end{array} \right. \quad \kappa_{nt} = 1$$

+ 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 :

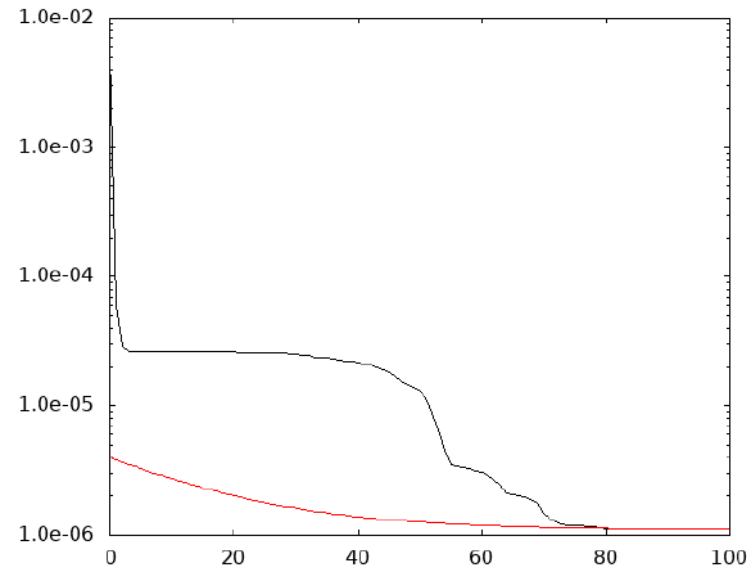
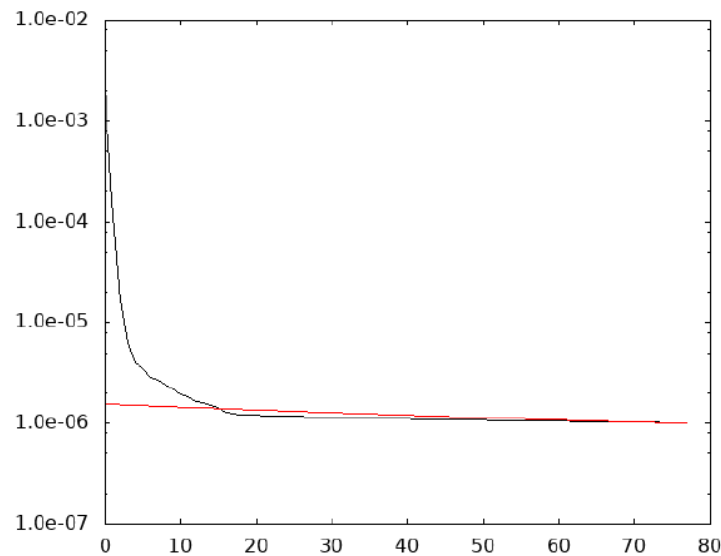
[**https://vthierry.github.io/mnemonas**](https://vthierry.github.io/mnemonas)

(*) 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 with**

different kind of nodes

several robots criteria

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

→ **it also allows global statistical estimation**



It works



That's all folks