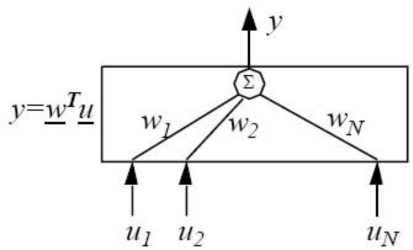
Adaptive Nonlinear Filters
Neural Networks
Overview

- Linear Perceptron Training $\equiv$ LMS algorithm
- Perceptron algorithm for Hard limiter Perceptrons
- Delta Rule training algorithm for Sigmoidal Perceptrons
- Generalized Delta Rule (Backpropagation) Algorithm for multilayer perceptrons
- Training static Multilayer Perceptron
- Temporal processing with NN
  - Neural Networks architectures for modelling dynamic signals and systems
  - Reduction of dynamic NN to static NN training through unfolding
  - Dynamic Backpropagation
  - Backpropagation through time

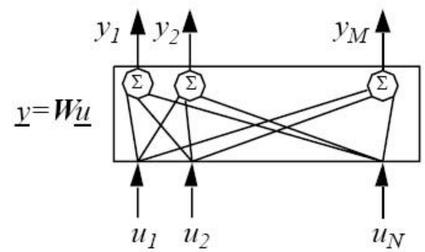
References:

Chapters 4, 5 and 6 from [Haykin, 1994] S. Haykin *Neural Networks – A comprehensive foundation* Macmillan College Publishing Company, 1994.

Linear Perceptron



LINEAR COMBINER (LINEAR PERCEPTRON)



MULTIPLE LINEAR COMBINER (ONE LAYER PERCEPTRON)

- A linear combiner has N inputs and one output.
- An FIR filter of order N is similar, but its N inputs are all shifted-in-time versions of the same signal.

Linear Perceptron Training $\equiv$ LMS algorithm

LMS algorithm for a linear FIR filter (REMINDER)

Given
- the (correlated) input signal samples $\{u(1), u(2), u(3), \ldots\}$, generated randomly;
- the desired signal samples $\{d(1), d(2), d(3), \ldots\}$ correlated with $\{u(1), u(2), u(3), \ldots\}$

1 Initialize the algorithm with an arbitrary parameter vector $\underline{w}(0)$, for example $\underline{w}(0) = 0$.

2 Iterate for $t = 0, 1, 2, 3, \ldots, n_{max}$

2.0 Read a new data pair, $(\underline{u}(t), d(t))$

2.1 (Compute the output) $y(t) = \underline{w}(t)^T \underline{u}(t) = \sum_{i=0}^{M-1} w_i(t) u(t-i)$

2.2 (Compute the error) $e(t) = d(t) - y(t)$

2.3 (Parameter adaptation) $\underline{w}(t+1) = \underline{w}(t) + \mu \underline{u}(t) e(t)$

or componentwise

$$\begin{bmatrix} w_0(t+1) \\ w_1(t+1) \\ . \\ . \\ . \\ w_{M-1}(t+1) \end{bmatrix} = \begin{bmatrix} w_0(t) \\ w_1(t) \\ . \\ . \\ . \\ w_{M-1}(t) \end{bmatrix} + \mu e(t) \begin{bmatrix} u(t) \\ u(t-1) \\ . \\ . \\ . \\ u(t-M+1) \end{bmatrix}$$

$\square$

LMS algorithm derivation for Linear Combiner (Review)

The Linear Combiner (linear neuron, or linear perceptron) has N inputs, which can be grouped into the vector

$$\underline{u}(t) = \left[\begin{array}{cccc} u_1(t) & u_2(t) & \ldots & u_N(t) \end{array}\right]^T$$

has also N parameters or weights (synaptic strength)

$$\underline{w} = \left[\begin{array}{cccc} w_1 & w_2 & \ldots & w_N \end{array}\right]^T$$

and computes its output as

$$y(t) \;=\; \underline{w}^T \underline{u}(t) = \sum_{i=1}^{N} w_i u_i(t)$$

which probably is not equal to the desired signal $d(t)$, the error being

$$e(t) \;=\; d(t) - y(t) = d(t) - \underline{w}^T \underline{u}(t)$$

The performance criterion is

$$J(\underline{w}) \;=\; E[e^2(t)] = E[(d(t) - y(t))^2] = E[(d(t) - \underline{w}^T \underline{u}(t))^2]$$

and must be minimized with respect to $\underline{w}$. At the minimum, the gradient vector must be zero

$$\begin{aligned} \nabla_{\underline{w}} J(\underline{w}) &= 0 \\ \nabla_{\underline{w}} J(\underline{w}) &= \nabla_{\underline{w}} E[e^2(t)] = 2Ee(t)\nabla_{\underline{w}}[(e(t)] = 2Ee(t)\nabla_{\underline{w}}[(d(t) - \underline{w}^T \underline{u}(t))] = -2Ee(t)\underline{u}(t) = 0 \end{aligned}$$

The gradient method for minimizing the criterion $E[e^2(t)]$ requires the modification at each time step of the parameter vector with a small step in the reversed direction of gradient vector:

$$\underline{w}(t+1) = \underline{w}(t) - \frac{1}{2}\mu \nabla_{\underline{w}(t)} J(\underline{w}(t)) = \underline{w}(t) + \mu[Ee(t)\underline{u}(t)]$$

In order to simplify the algorithm, instead the true gradient of the criterion

$$\nabla_{\underline{w}(t)} J(t) = -2E\underline{u}(t)e(t)$$

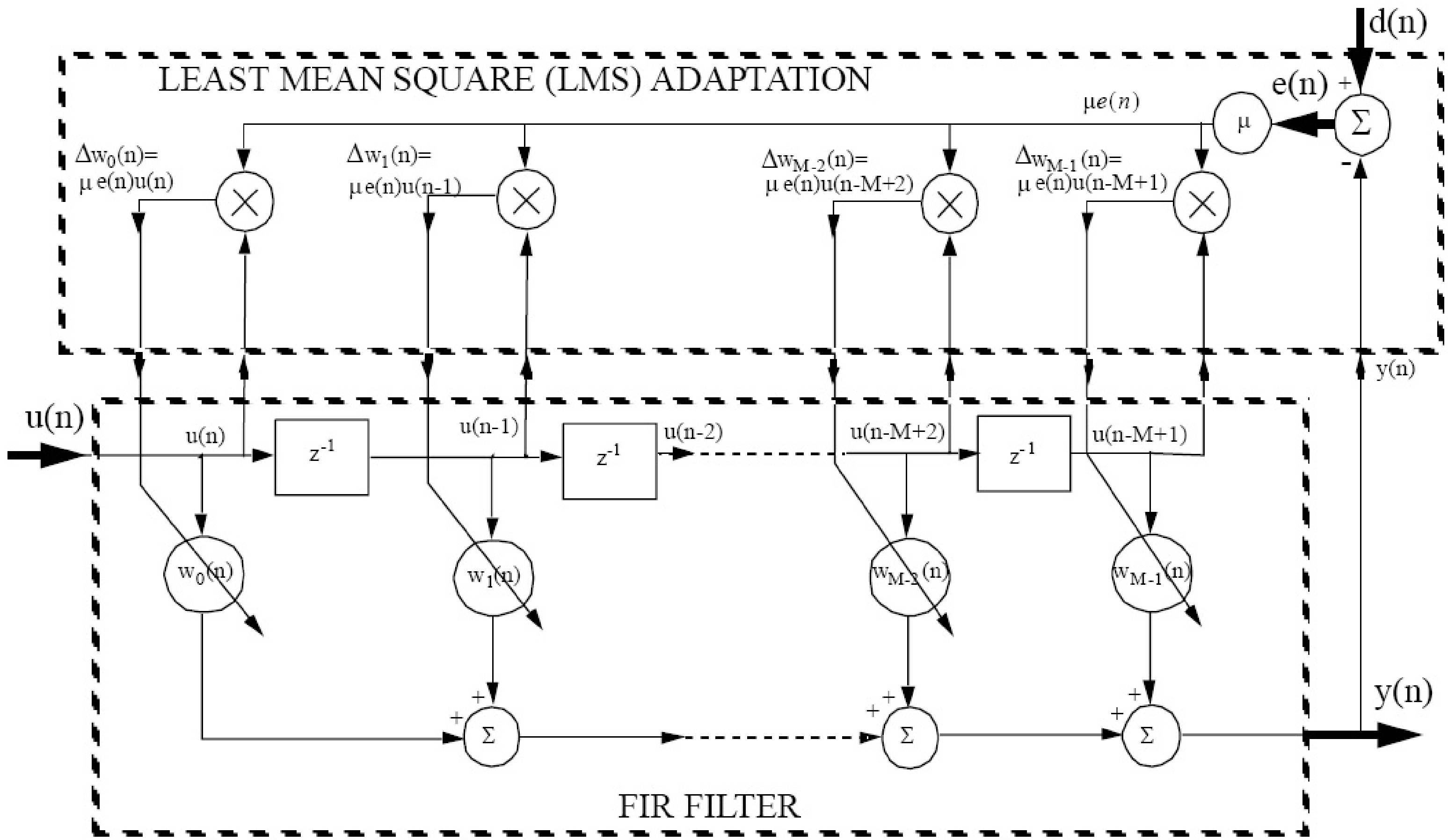
LMS algorithm will use an immediately available approximation

$$\hat{\nabla}_{\underline{w}(t)} J(t) = -2\underline{u}(t)e(t)$$

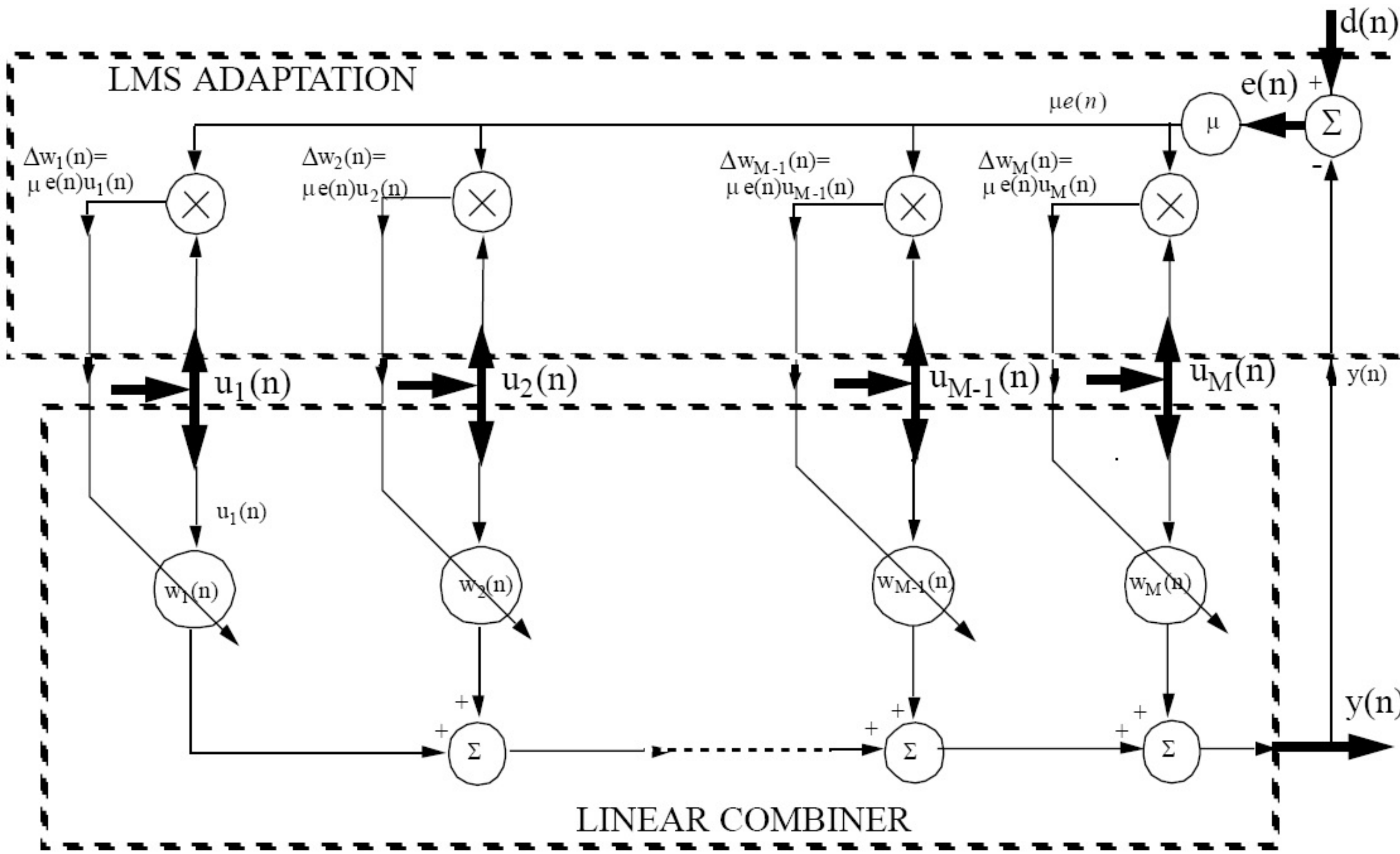
Using **the noisy gradient**, the adaptation will carry on the equation

$$\underline{w}(t+1) = \underline{w}(t) - \frac{1}{2}\mu\hat{\nabla}_{\underline{w}(t)} J(t) = \underline{w}(t) + \mu\underline{u}(t)e(t)$$

A Reminder of the circuit implementing the LMS algorithm



A similar circuit implementing the adaptation of the linear combiner (perceptron)



LMS algorithm for a Linear Combiner

Given
- the (correlated) input vector samples $\{\underline{u}(1), \underline{u}(2), \underline{u}(3), \ldots\}$, generated randomly;
- the desired signal samples $\{d(1), d(2), d(3), \ldots\}$

1 Initialize the algorithm with an arbitrary parameter vector $\underline{w}(0)$, for example $\underline{w}(0) = 0$.

2 Iterate for $t = 0, 1, 2, 3, \ldots, n_{max}$

2.0 Read a new data pair, $(\underline{u}(t), d(t))$

2.1 (Compute the output) $y(t) = \underline{w}(t)^T \underline{u}(t) = \sum_{i=1}^{N} w_i(t) u_i(t)$

2.2 (Compute the error) $e(t) = d(t) - y(t)$

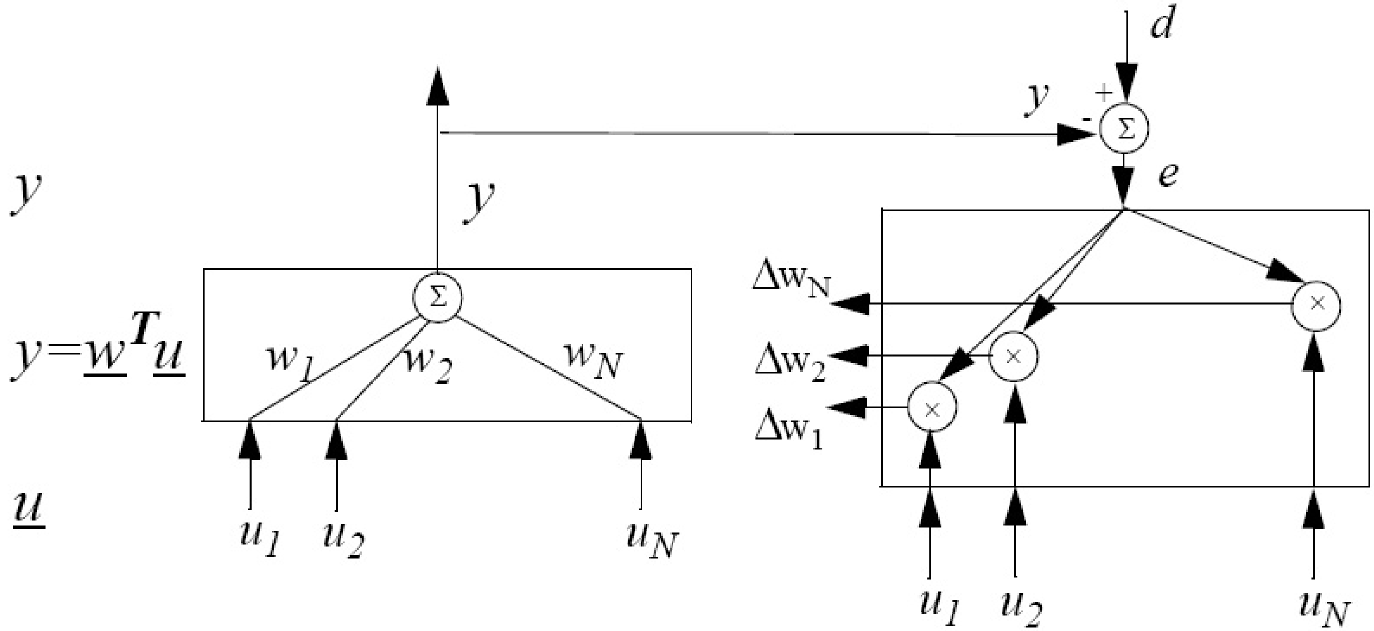
2.3 (Parameter adaptation) $\underline{w}(t+1) = \underline{w}(t) + \mu \underline{u}(t) e(t)$

or componentwise

$$\begin{bmatrix} w_1(t+1) \\ w_2(t+1) \\ . \\ . \\ . \\ w_N(t+1) \end{bmatrix} = \begin{bmatrix} w_1(t) \\ w_2(t) \\ . \\ . \\ . \\ w_N(t) \end{bmatrix} + \mu e(t) \begin{bmatrix} u_1(t) \\ u_2(t) \\ . \\ . \\ . \\ u_N(t) \end{bmatrix}$$

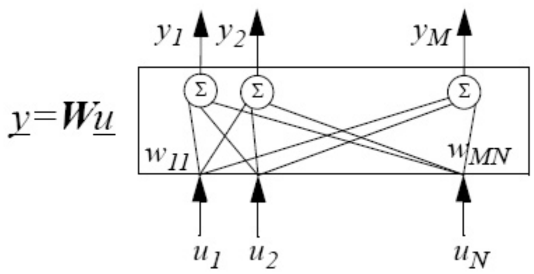
□

A Redrawing of the circuit implementing the adaptation of the linear combiner (perceptron)

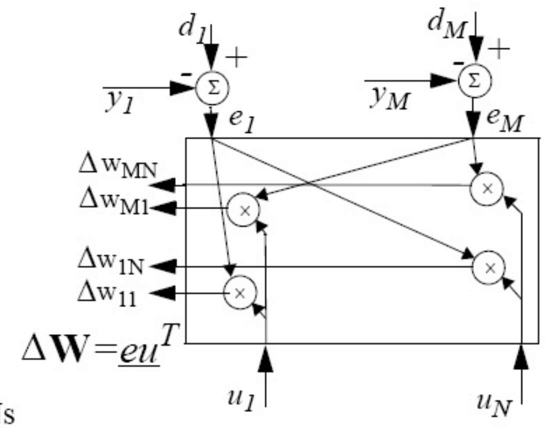


TRAINING A LINEAR PERCEPTRON

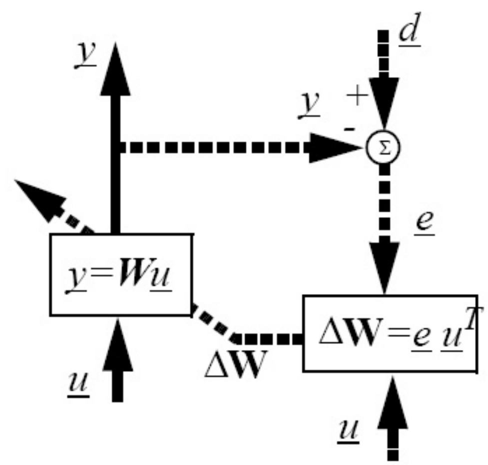
Generalizing the linear combiner to a layer of M linear combiners



ONE LAYER OF LINEAR PERCEPTRONs



ADAPTATION CIRCUIT



TRAINING ONE LAYER LINEAR PERCEPTRON

Perceptrons with Hard Nonlinearities

A perceptron with hard nonlinearity (the proper perceptron) is a linear combiner followed by one "hard" nonlinearity $h = h^{[0,1]}$ or $h = h^{[-1,1]}$

$$y(t) \;=\; h[\underline{w}^T \underline{u}(t)] = h^{0,1}[\sum_{i=1}^{N} w_i u_i(t)]$$

where

$$h^{[0,1]}(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

$$h^{[-1,1]}(x) = \mathrm{sign}(x) = \begin{cases} 1, & x \geq 0 \\ -1, & x < 0 \end{cases}$$

Defining as before the error

$$e(t) \;=\; d(t) - y(t)$$

the performance criterion can be written as either MSE (mean square error) or MAE(mean absolute error)

$$J(\underline{w}) \;=\; E[e^2(t)] = E[(d(t) - y(t))^2] = E[|e(t)|]$$

(for binary values of $d(t)$ and $y(t)$ MSE and MAE are identical), and must be minimized with respect to $\underline{w}$. The gradient is

$$\nabla_{\underline{w}} J(\underline{w}) \;=\; \nabla_{\underline{w}} E[e^2(t)] = 2Ee(t)\nabla_{\underline{w}}[(e(t)] = 2Ee(t)\nabla_{\underline{w}}(d(t) - h[\underline{w}^T \underline{u}(t)]) = -2Ee(t)\underline{u}(t)h'[\underline{w}^T \underline{u}(t)]$$

The problem with this expression is that almost everywhere $h' = 0$ so the gradient based updating will be zero almost all the time, and when it is not zero, the derivative is not defined.

However, using an algorithm with the updating

$$\underline{w}(t+1) = \underline{w}(t) + \underline{u}(t)e(t)$$

will result in a powerful result:

If there is a vector $\underline{w}_0$ which makes the criterion zero, the algorithm will find in a finite number of iterations a parameter vector making the criterion zero (probably not $\underline{w}_0$, since in general there are many different parameter vectors for which the criterion is zero).

Perceptron algorithm

Given
- the (correlated) input vector samples $\{\underline{u}(1), \underline{u}(2), \underline{u}(3), \ldots\}$, generated randomly;
- the desired signal samples $\{d(1), d(2), d(3), \ldots\}$

1 Initialize the algorithm with an arbitrary parameter vector $\underline{w}(0)$, for example $\underline{w}(0) = 0$.

2 Iterate for $t = 0, 1, 2, 3, \ldots, n_{max}$

2.0 Read a new data pair, $(\underline{u}(t), d(t))$

2.1 (Compute the output) $y(t) = \text{sign}[\underline{w}(t)^T \underline{u}(t)] = \text{sign}[\sum_{i=1}^{N} w_i(t) u_i(t)]$

2.2 (Compute the error) $e(t) = d(t) - y(t)$

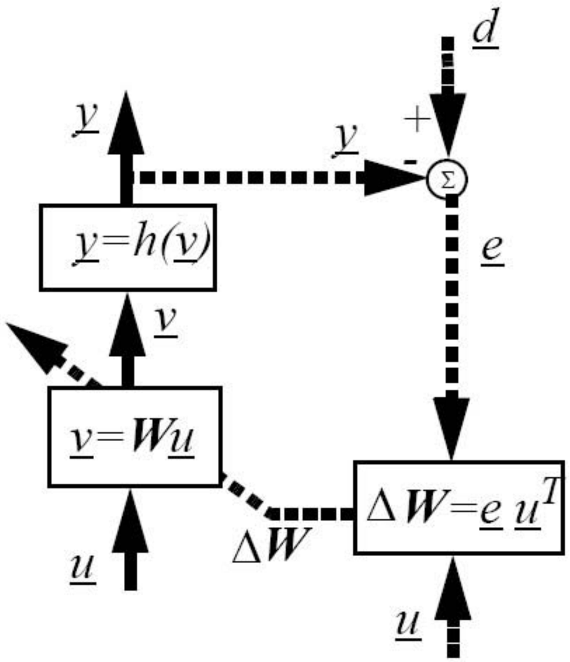
2.3 (Parameter adaptation) $\underline{w}(t+1) = \underline{w}(t) + \underline{u}(t) e(t)$

or componentwise

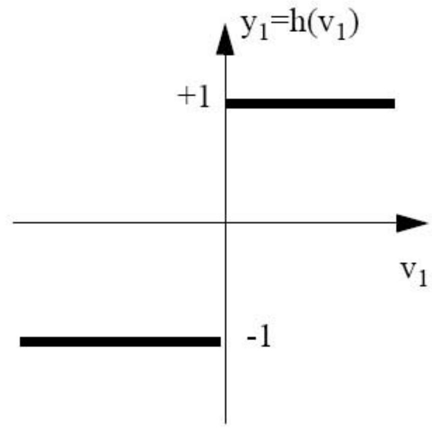
$$\begin{bmatrix} w_1(t+1) \\ w_2(t+1) \\ . \\ . \\ . \\ w_N(t+1) \end{bmatrix} = \begin{bmatrix} w_1(t) \\ w_2(t) \\ . \\ . \\ . \\ w_N(t) \end{bmatrix} + e(t) \begin{bmatrix} u_1(t) \\ u_2(t) \\ . \\ . \\ . \\ u_N(t) \end{bmatrix}$$

□

A circuit implementing the adaptation of the perceptron with hard nonlinearity



TRAINING ONE LAYER HARD LIMITER PERCEPTRON



Perceptrons with Sigmoidal Nonlinearities

A perceptron with a sigmoidal nonlinearity (the proper perceptron) is a linear combiner followed by a monotonically increasing nonlinearity h

$$y(t) \;=\; h[\underline{w}^T \underline{u}(t)]$$

where, e.g. h can be the nonsymmetrical function

$$h(x) = \frac{1}{1+e^{-\beta x}}$$

or the symmetrical function

$$h(x) = \frac{1-e^{-\beta x}}{1+e^{-\beta x}}$$

Now the derivation we tried for the hardlimiter nonlinearity will be meaningful: The gradient

$$\nabla_{\underline{w}} J(\underline{w}) \;=\; \nabla_{\underline{w}} E[e^2(t)] = 2Ee(t)\nabla_{\underline{w}}[(e(t)] = 2Ee(t)\nabla_{\underline{w}}(d(t) - h[\underline{w}^T \underline{u}(t)]) = -2Ee(t)\underline{u}(t)h'[\underline{w}^T \underline{u}(t)]$$

can be approximated by

$$\nabla_{\underline{w}} J(\underline{w}) \;\approx\; -2e(t)\underline{u}(t)h'[\underline{w}^T \underline{u}(t)]$$

and then use the gradient based update

$$\underline{w}(t+1) = \underline{w}(t) + \mu\underline{u}(t)e(t)h'[\underline{w}^T(t)\underline{u}(t)]$$

The computation of the derivative can be done using (for the nonsymmetrical sigmoid)

$$h'[x] = h(x)(1-h(x))$$

Delta Rule Algorithm

Given
- the (correlated) input vector samples $\{\underline{u}(1), \underline{u}(2), \underline{u}(3), \ldots\}$, generated randomely;
- the desired signal samples $\{d(1), d(2), d(3), \ldots\}$

1 Initialize the algorithm with an arbitrary parameter vector $\underline{w}(0)$, for example $\underline{w}(0) = 0$.

2 Iterate for $t = 0, 1, 2, 3, \ldots, n_{max}$

2.0 Read a new data pair, $(\underline{u}(t), d(t))$

2.1 (Compute the output) $y(t) = h[\underline{w}(t)^T \underline{u}(t)] = h[\sum_{i=1}^{N} w_i(t) u_i(t)]$

2.2 (Compute the error) $e(t) = d(t) - y(t)$

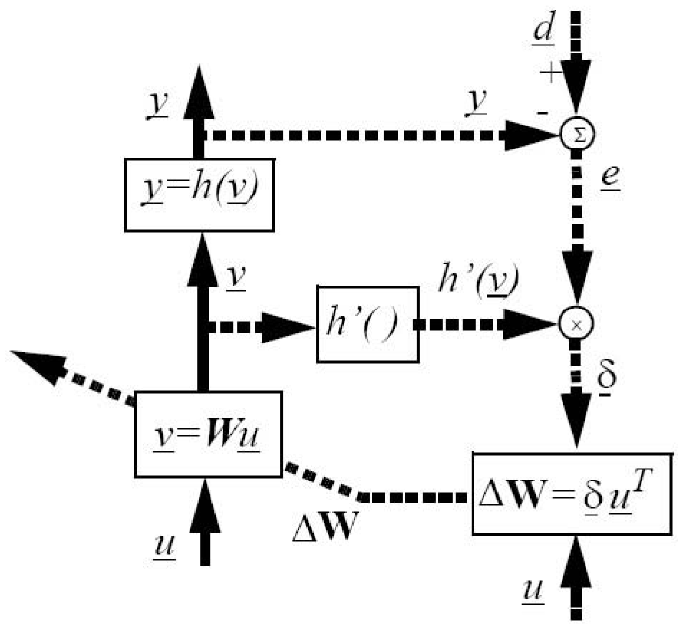
2.3 (Parameter adaptation) $\underline{w}(t+1) = \underline{w}(t) + \mu \underline{u}(t) e(t) h'[\underline{w}^T(t) \underline{u}(t)]$

or componentwise

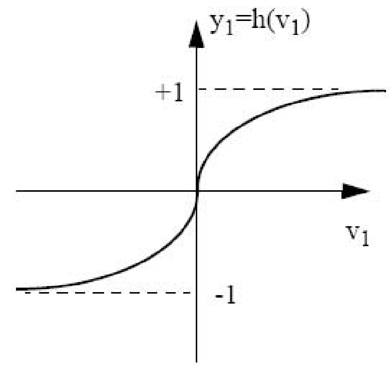
$$\begin{bmatrix} w_1(t+1) \\ w_2(t+1) \\ \cdot \\ \cdot \\ \cdot \\ w_N(t+1) \end{bmatrix} = \begin{bmatrix} w_1(t) \\ w_2(t) \\ \cdot \\ \cdot \\ \cdot \\ w_N(t) \end{bmatrix} + \mu e(t) h'[\underline{w}^T(t) \underline{u}(t)] \begin{bmatrix} u_1(t) \\ u_2(t) \\ \cdot \\ \cdot \\ \cdot \\ u_N(t) \end{bmatrix}$$

□

A circuit implementing the adaptation of the perceptron with sigmoidal (soft) nonlinearity



TRAINING ONE LAYER SIGMOIDAL PERCEPTRON



SIGMOIDAL NONLINEARITY

Multilayer Perceptrons

1. Multilayer Perceptron: Consider a three layers perceptron (having two hidden layers). The first perceptron layer has
   - n_1 inputs, $\underline{u}^{[1]} = \left[\begin{array}{ccc} u_1^{[1]} & \ldots & u_{n_1}^{[1]} \end{array}\right]^T$ and n_2 neurons;
   - The weight matrix $\mathbf{W}^{[1]}$ with dimensions $n_2 \times n_1$;
   - the activations $\underline{v}^{[1]} = \left[\begin{array}{ccc} v_1^{[1]} & \ldots & v_{n_2}^{[1]} \end{array}\right]^T$ of the neurons are computed as $\underline{v}^{[1]} = \mathbf{W}^{[1]}\underline{u}^{[1]}$
   - the outputs $\underline{u}^{[2]} = \left[\begin{array}{ccc} u_1^{[2]} & \ldots & u_{n_2}^{[2]} \end{array}\right]^T$ of the neurons are computed as $\underline{u}^{[2]} = h(\underline{v}^{[1]})$
2. The second perceptron layer has
   - n_2 inputs, $\underline{u}^{[2]} = \left[\begin{array}{ccc} u_1^{[2]} & \ldots & u_{n_2}^{[2]} \end{array}\right]^T$ and n_3 neurons;
   - The weight matrix $\mathbf{W}^{[2]}$ with dimensions $n_3 \times n_2$;
   - the activations $\underline{v}^{[2]} = \left[\begin{array}{ccc} v_1^{[2]} & \ldots & v_{n_3}^{[2]} \end{array}\right]^T$ of the neurons are computed as $\underline{v}^{[2]} = \mathbf{W}^{[2]}\underline{u}^{[2]}$
   - the outputs $\underline{u}^{[3]} = \left[\begin{array}{ccc} u_1^{[3]} & \ldots & u_{n_3}^{[3]} \end{array}\right]^T$ of the neurons are computed as $\underline{u}^{[3]} = h(\underline{v}^{[2]})$
3. The third perceptron layer has
   - n_3 inputs, $\underline{u}^{[3]} = \left[\begin{array}{ccc} u_1^{[3]} & \ldots & u_{n_3}^{[3]} \end{array}\right]^T$ and n_4 neurons;
   - The weight matrix $\mathbf{W}^{[3]}$ with dimensions $n_4 \times n_3$;

- the activations $\underline{v}^{[3]} = \begin{bmatrix} v_1^{[3]} & \dots & v_{n_2}^{[3]} \end{bmatrix}^T$ of the neurons are computed as $\underline{v}^{[3]} = \mathbf{W}^{[3]}\underline{u}^{[3]}$
- the outputs $\underline{y} = \underline{u}^{[4]} = \begin{bmatrix} u_1^{[3]} & \dots & u_{n_4}^{[3]} \end{bmatrix}^T$ of the neurons are computed as $\underline{y} = \underline{u}^{[4]} = h(\underline{v}^{[3]})$

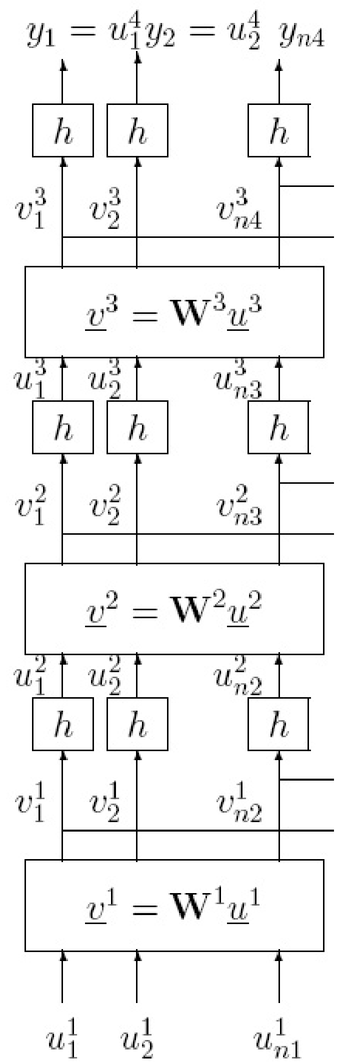
The computation of the output $\underline{y}$ with the multilayer perceptron is realized using

$$\begin{aligned} \underline{v}^{[1]} &= \mathbf{W}^{[1]}\underline{u}^{[1]}; & \underline{u}^{[2]} &= h(\underline{v}^{[1]}) \\ \underline{v}^{[2]} &= \mathbf{W}^{[2]}\underline{u}^{[2]}; & \underline{u}^{[3]} &= h(\underline{v}^{[2]}) \\ \underline{v}^{[3]} &= \mathbf{W}^{[3]}\underline{u}^{[3]}; & \underline{y} = \underline{u}^{[4]} &= h(\underline{v}^{[3]}) \end{aligned}$$

or written componentwise

$$\begin{aligned} v_i^{[1]} &= \sum_{j=1}^{n_1} w_{ij}^{[1]} u_j^{[1]} & u_i^{[2]} &= h(v_i^{[1]}) & i &= 1, \dots, n_2 \\ v_i^{[2]} &= \sum_{j=1}^{n_2} w_{ij}^{[2]} u_j^{[2]} & u_i^{[3]} &= h(v_i^{[2]}) & i &= 1, \dots, n_3 \\ v_i^{[3]} &= \sum_{j=1}^{n_3} w_{ij}^{[3]} u_j^{[3]} & y_i = u_i^{[4]} &= h(v_i^{[3]}) & i &= 1, \dots, n_4 \end{aligned}$$

A circuit implementing the multilayer perceptron



The optimality criterion can be selected to take into account the norm of errors at time t:

$$J_t = \frac{1}{2}\sum_{i=1}^{n_4}(d_i(t) - u_i^{[4]}(t))^2 = \frac{1}{2}\sum_{i=1}^{n_4} e^2(t) = \frac{1}{2}e^T e \tag{1}$$

Another choice is to evaluate with the criterion the overall training set ($t = 1, \ldots, N_{set}$) performance of the neural network

$$J = \frac{1}{2}\sum_{t=1}^{Nset} J_t \tag{2}$$

In order to minimize (1) and (2) we introduce some notations, and derive the BP algorithm:

$$\frac{\partial J_t}{\partial v_i^{[3]}} = \frac{\partial J_t}{\partial u_i^{[4]}}\frac{\partial u_i^{[4]}}{\partial v_i^{[3]}} = -(d_i - u_i^{[4]})h'(v^{[3]}) \triangleq -\delta_i^{[3]} \tag{3}$$

For the output layer the gradient computation is straightforward

$$\frac{\partial J_t}{\partial w_{ij}^{[3]}} = \frac{\partial J_t}{\partial v_i^{[3]}}\frac{\partial v_i^{[3]}}{\partial w_{ij}^{[3]}} = -\delta_i^{[3]} u_j^{[3]} \tag{4}$$

For any hidden layer, we recognize that

$$\frac{\partial v_i^{[k+1]}}{\partial v_j^{[k]}} = \frac{\partial v_i^{[k+1]}}{\partial u_j^{[k]}}\frac{\partial u_j^{[k]}}{\partial v_j^{[k]}} = w_{ij}^{[k]} h'(v_j^{[k]}) \tag{5}$$

and

$$\frac{\partial v_n^{[k+1]}}{\partial w_{ij}^{[k]}} = \begin{cases} u_j^{[k]}, & n = i \\ 0, & n \neq i \end{cases} \tag{6}$$

Now we can apply the rule for derivative of a compose function

$$\frac{\partial J_t}{\partial v_j^{[k]}} = \sum_{i=1}^{n_{k+2}} \frac{\partial J_t}{\partial v_i^{[k+1]}} \frac{\partial v_i^{[k+1]}}{\partial v_j^{[k]}} = -\sum_{i=1}^{n_{k+2}} \delta_i^{[k]} w_{ij}^{[k]} h'(v_j^{[k]}) \triangleq -\delta_j^{[k-1]} \tag{7}$$

$$\frac{\partial J_t}{\partial w_{ij}^{[k]}} = \sum_{p=1}^{n_{k+1}} \frac{\partial J_t}{\partial v_p^{[k+1]}} \frac{\partial v_p^{[k+1]}}{\partial w_{ij}^{[k]}} = -\delta_i^{[k]} u_j^{[k]} \tag{8}$$

Backpropagation Algorithm

1. Initialization of weight matrices, $\mathbf{W}^{[1]}(1)$, $\mathbf{W}^{[2]}(1)$, $\mathbf{W}^{[3]}(1)$ at time instant $t = 1$ with small random numbers.

2. For $it = 1, 2, 3, \ldots, N_{it}$

    2.0 Initialize $\left(\frac{\partial J(it)}{\partial w_{ij}^{[k]}}\right) \leftarrow 0, \quad k = \overline{1,3}, \quad j = \overline{1, n_k}, \quad i = \overline{1, n_{k+1}}, \quad J(it) \leftarrow 0$

    2.1 For $n = 1, 2, \ldots, N_{set}$ (we omit writing the time argument (t))

    2.1.0 Read a new element $(\underline{u}^{[1]}(t), d(t)))$

    2.1.1 Forward computations "**FORWARD PATH**"

$$\underline{v}^{[1]} = \mathbf{W}^{[1]}\underline{u}^{[1]}, \qquad \underline{u}^{[2]} = h(\underline{v}^{[1]})$$
$$\underline{v}^{[2]} = \mathbf{W}^{[2]}\underline{u}^{[2]}, \qquad \underline{u}^{[3]} = h(\underline{v}^{[2]})$$
$$\underline{v}^{[3]} = \mathbf{W}^{[3]}\underline{u}^{[3]}, \qquad \underline{u}^{[4]} = h(\underline{v}^{[3]})$$
$$\underline{e} = \underline{d}(t) - \underline{u}^{[4]}, \qquad J_t = \underline{e}^T\underline{e} = \sum_{j=1}^{n_4} e_j^2$$
$$J(it) \leftarrow J(it) + J_t$$

2.1.2 Backward computation "**BACKWARD PATH**"

Compute the generalized errors for all neurons, starting with last layer

$\delta_i^{[3]} = (d_i(t) - u_i^{[4]})h'(v_i^{[3]}) \quad i = \overline{1, n4}$

$\delta_i^{[2]} = h'(v_i^{[2]}) \sum_{j=1}^{n_4} w_{ji}^{[3]} \delta_i^{[3]} \quad i = \overline{1, n3}$

$\delta_i^{[1]} = h'(v_i^{[1]}) \sum_{j=1}^{n_3} w_{ji}^{[2]} \delta_i^{[2]} \quad i = \overline{1, n2}$

Compute the elements of the gradients, $\frac{\partial J_t}{\partial w_{ij}^{[1]}}$

$$\left(\frac{\partial J_t}{\partial w_{ij}^{[1]}}\right) = -u_j^{[1]} \delta_i^{[1]} \quad i = \overline{1, n_2}, \quad j = \overline{1, n_1}$$

$$\left(\frac{\partial J_t}{\partial w_{ij}^{[2]}}\right) = -u_j^{[2]} \delta_i^{[2]} \quad i = \overline{1, n_3}, \quad j = \overline{1, n_2}$$

$$\left(\frac{\partial J_t}{\partial w_{ij}^{[3]}}\right) = -u_j^{[3]} \delta_i^{[3]} \quad i = \overline{1, n_4}, \quad j = \overline{1, n_3}$$

2.1.3 Accumulate the gradient elements $J(it)$

$$\left(\frac{\partial J(it)}{\partial w_{ij}^{[k]}}\right) \leftarrow \left(\frac{\partial J(it)}{\partial w_{ij}^{[k]}}\right) + \left(\frac{\partial J_t}{\partial w_{ij}^{[k]}}\right), \quad k = \overline{1, 3}, \quad j = \overline{1, n_k}, \quad i = \overline{1, n_{k+1}}$$

2.2 If $J(it) < \varepsilon$ **STOP**

2.3 Modify the wheight values in the direction opposed to gradient vector

$$w_{ij}^{[1]}(it+1) = w_{ij}^{[1]}(it) - \lambda \left(\frac{\partial J}{\partial w_{ij}^{[1]}}\right)_{it}$$

$$w_{ij}^{[2]}(it+1) = w_{ij}^{[2]}(it) - \lambda \left(\frac{\partial J}{\partial w_{ij}^{[2]}}\right)_{it}$$

$$w_{ij}^{[3]}(it+1) = w_{ij}^{[3]}(it) - \lambda \left(\frac{\partial J}{\partial w_{ij}^{[3]}}\right)_{it}$$

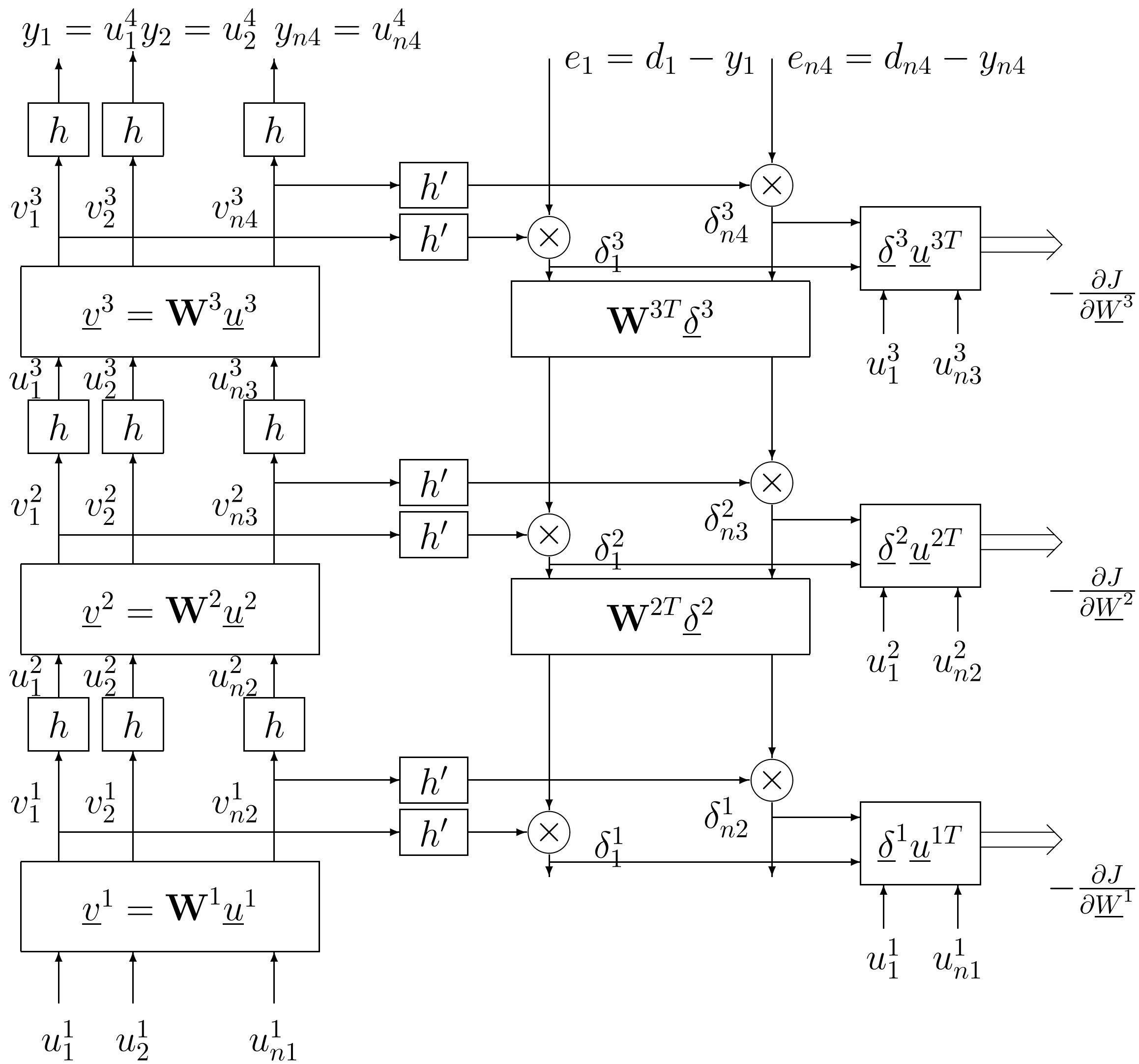


Figure 1: Circuit implementing Backpropagation algorithm

Temporal processing with Neural Networks

Neural Networks architectures for modelling dynamic signals and systems

- NN with linear dinamic synapses: FIR multilayer perceptron and Time Delay Neural Networks
- Mixed NN – Transfer function model
- Standard nonlinear state space models

Generalization of static NN models to dynamic NN models

- *Definition 1*: A static NN is characterized by :
  - has memoryless transmitance of the synapse, w_{ij}, between the output of neuron j and the input of neuron i;
  - there are no feedback loops (once there is a connection from neuron i to neuron j, there is no connection path from neuron j back to neuron i.
- *Definition 2*: A dynamic NN satisfies one of the following conditions:
  - either has a nontrivial transfer function at some synapses (a linear filter) $w_{ij}(q^{-1})$, between the output of neuron j and the input of neuron i (e.g. FIR multilayer perceptrons).
  - or it contains feedback loops: dynamic feedback loops (not algebraic loops) (e.g. reccurent neural networks).
  - or it is formed by composing Linear Filters structures with Neural Network structures (not necessarily at synapse level).

NN with linear dinamic synapses: FIR multilayer perceptrons

The basic model of a neuron can be generalized to include memory elements, (like delay elements).

Consider the classical sigmoidal perceptron, which processes the inputs $u_1, u_2, \ldots, u_N$ as

$$\begin{aligned} v &= \sum_{i=1}^{N} w_i u_i = \underline{w}^T \underline{u} \\ y &= h(v) \end{aligned}$$

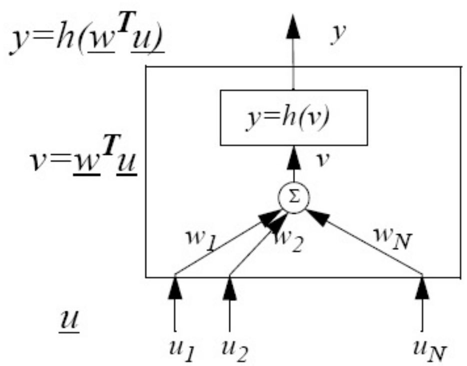
to obtain the output y. We don't specify the time moment t since all variables have the same time argument. The nonlinear function $h(\cdot)$ can be either symmetric sigmoid or the asymmetrical sigmoid.

A dynamic neuron can be defined as

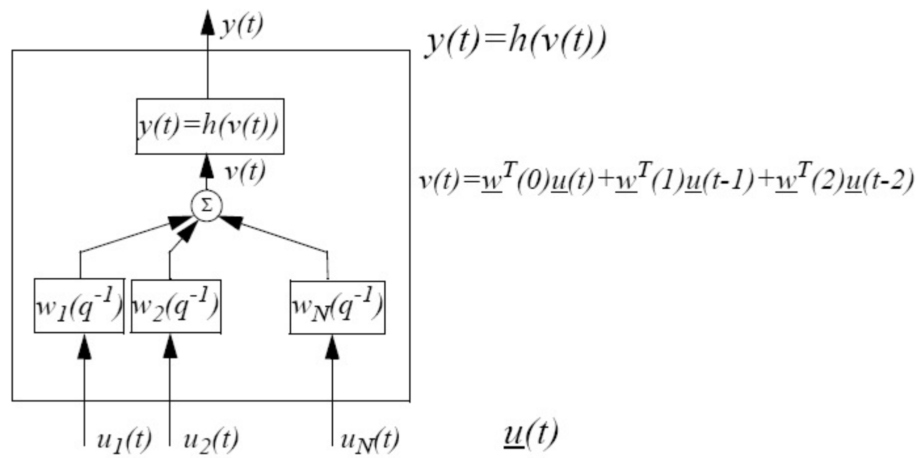
$$\begin{aligned} v(t) &= \sum_{i=1}^{N} w_i(q^{-1}) u_i(t) \\ y(t) &= h(v(t)) \end{aligned}$$

where each $w_i(q^{-1})$ is a FIR filter.

Static and dynamic neurons



A static neuron



A dynamic neuron

A **FIR multilayer perceptron** is obtained replacing all neurons in a multilayer perceptron by dynamic neurons.

Example

For notation simplicity, consider dynamic neurons FIR filters of order 3.

Consider a three layers perceptron (having two hidden layers).

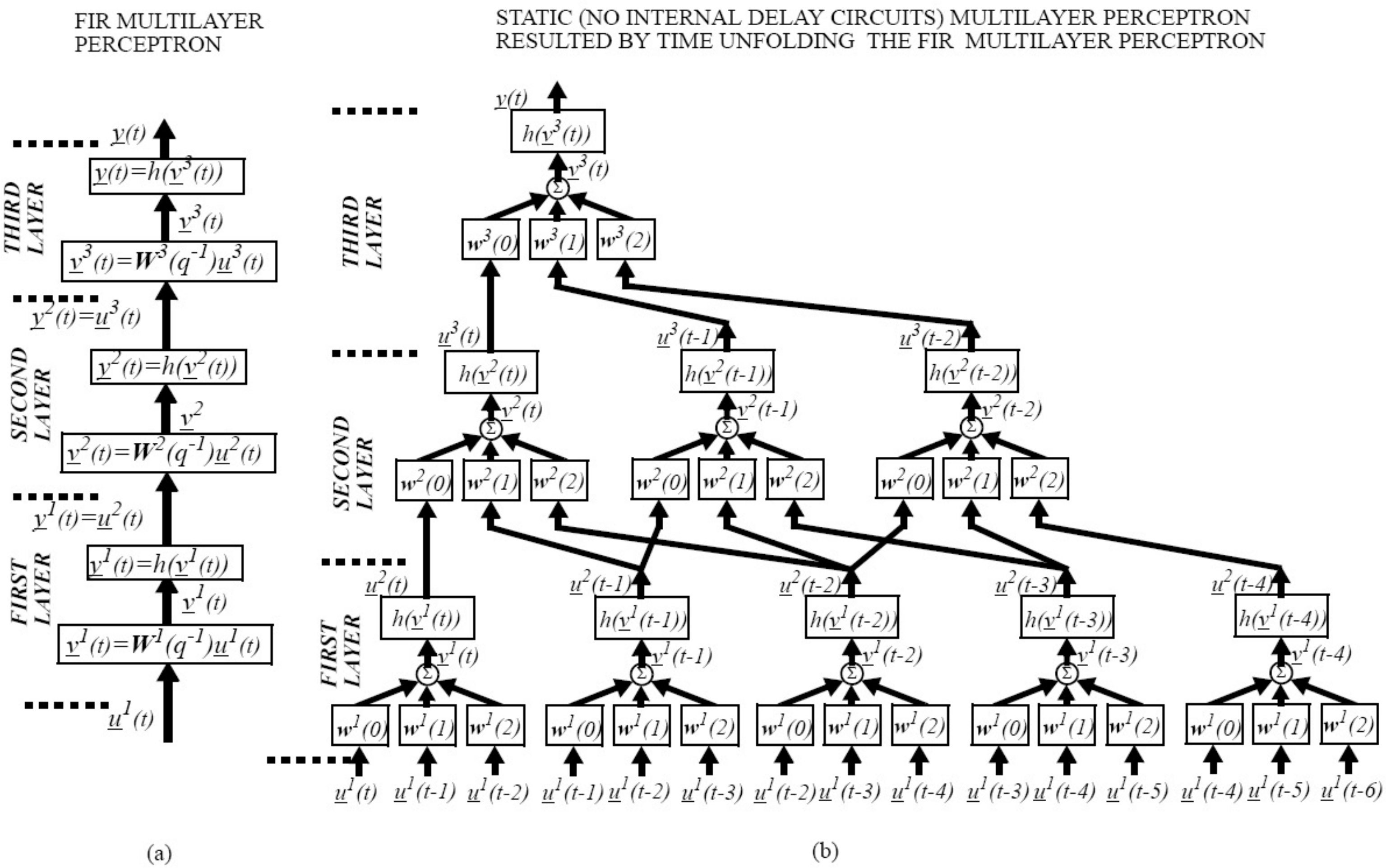
We define for the first perceptron layer (for other layers notations are straightforward extensions of static multilayer perceptron)

- n_1 inputs, $\underline{u}^{[1]} = \left[\begin{array}{ccc} u_1^{[1]} & \dots & u_{n_1}^{[1]} \end{array} \right]^T$ and n_2 neurons;
- The matrix of transfer functions $\mathbf{W}^{[1]}(q^{-1})$ with $n_2 \times n_1$ elements,
$$\mathbf{W}^{[1]}(q^{-1}) = \mathbf{w}_0^{[1]} + \mathbf{w}_1^{[1]} q^{-1} + \mathbf{w}_2^{[1]} q^{-2}$$
where the matrices $\mathbf{w}_0^{[1]}$, $\mathbf{w}_1^{[1]}$, and $\mathbf{w}_2^{[1]}$ have dimensions $n_2 \times n_1$;
- the activations $\underline{v}^{[1]} = \left[\begin{array}{ccc} v_1^{[1]} & \dots & v_{n_2}^{[1]} \end{array} \right]^T$ of the neurons are computed as
$$\underline{v}^{[1]}(t) = \mathbf{W}^{[1]}(q^{-1})\underline{u}^{[1]}(t) = \mathbf{w}_0^{[1]}\underline{u}^{[1]}(t) + \mathbf{w}_1^{[1]}\underline{u}^{[1]}(t-1) + \mathbf{w}_2^{[1]}\underline{u}^{[1]}(t-2)$$
or elementwise
$$v_i^{[1]}(t) = \sum_{j=1}^{n_1} \mathbf{W}_{ij}^{[1]}(q^{-1}) u_j^{[1]}(t) = \sum_{j=1}^{n_1} (\mathbf{w}_{0ij}^{[1]} u_j^{[1]}(t) + \mathbf{w}_{1ij}^{[1]} u_j^{[1]}(t-1) + \mathbf{w}_{2ij}^{[1]} u_j^{[1]}(t-2))$$

- the outputs $\underline{u}^{[2]} = \left[\begin{array}{ccc} u_1^{[2]} & \dots & u_{n_2}^{[2]} \end{array} \right]^T$ of the neurons are computed as $\underline{u}^{[2]}(t) = h(\underline{v}^{[1]}(t))$

The resulting FIR multilayer perceptron is shown in Figure (a).

A circuit implementing the adaptation of the FIR multilayer perceptron



Reduction of dynamic NN training to static NN training through unfolding

The optimality criterion can be selected to take into account the norm of errors at time t:

$$J_t = \frac{1}{2}\sum_{i=1}^{n_4}(d_i(t) - y_i(t))^2 = \frac{1}{2}(\underline{d}(t) - \underline{y}(t))^T(\underline{d}(t) - \underline{y}(t)) = \frac{1}{2}\sum_{i=1}^{n_4} e^2(t) = \frac{1}{2}\underline{e}(t)^T\underline{e}(t) \tag{9}$$

Another choice is to evaluate with the criterion the overall training set ($t = 1, \ldots, N_{set}$) performance of the neural network

$$J = \frac{1}{2}\sum_{t=1}^{Nset} J_t \tag{10}$$

The derivative of criterion J_t with respect to an elemental weight, say, $\mathbf{w}_{0ij}^{[1]}$, of the FIRMP can be computed applying the derivative chain rule to the system of equations

$$\begin{aligned}
v_i^{[1]}(t) &= \sum_{j=1}^{n_1}(\mathbf{w}_{0ij}^{[1]}u_j^{[1]}(t) + \mathbf{w}_{1ij}^{[1]}u_j^{[1]}(t-1) + \mathbf{w}_{2ij}^{[1]}u_j^{[1]}(t-2)), \quad i = 1, \ldots, n_2, \quad j = 1, \ldots, n_1 \\
u_i^{[2]}(t) &= h(v_i^{[1]}(t)), \quad i = 1, \ldots, n_2, \\
v_i^{[2]}(t) &= \sum_{j=1}^{n_2}(\mathbf{w}_{0ij}^{[2]}u_j^{[2]}(t) + \mathbf{w}_{1ij}^{[2]}u_j^{[2]}(t-1) + \mathbf{w}_{2ij}^{[2]}u_j^{[2]}(t-2)), \quad i = 1, \ldots, n_3, \quad j = 1, \ldots, n_2 \\
u_i^{[3]}(t) &= h(v_i^{[2]}(t)), \quad i = 1, \ldots, n_3, \\
v_i^{[3]}(t) &= \sum_{j=1}^{n_3}(\mathbf{w}_{0ij}^{[3]}u_j^{[3]}(t) + \mathbf{w}_{1ij}^{[3]}u_j^{[3]}(t-1) + \mathbf{w}_{2ij}^{[3]}u_j^{[3]}(t-2)), \quad i = 1, \ldots, n_4, \quad j = 1, \ldots, n_3 \\
y_i(t) &= u_i^{[4]}(t) = h(v_i^{[3]}(t)), \quad i = 1, \ldots, n_4, \\
J_t &= \sum_{i=1}^{n_4}(d_i(t) - y_i(t))^2
\end{aligned}$$

Once the gradients

$$\frac{dJ_t}{d\mathbf{w}_{lij}^{[k]}}$$

for all defined k, l, i, j are computed, the updating of the weights will take place as:

$$\mathbf{w}_{lij}^{[k]}(t+1) = \mathbf{w}_{lij}^{[k]}(t) - \mu \frac{dJ_t}{d\mathbf{w}_{lij}^{[k]}}$$

Unfolding the FIR MP

One equivalent way to perform the training of FIR MP is to use the standard Backpropagation algorithm for an unfolded structure, as in Figure (b).

- Establish the structure of the static multilayer perceptron,(number of neurons in each layer)
  In the first layer: $(15 \times n_1 \times n_2)$ nonzero weights, there are $n_1^{ex} = 15n_1$ inputs and $n_2^{ex} = 5n_2$ outputs;
  In the second layer: $(9 \times n_2 \times n_3)$ nonzero weights, there are $n_2^{ex} = 5n_2$ inputs and $n_3^{ex} = 3n_3$ outputs;
  In the third layer: $(3 \times n_3 \times n_4)$ nonzero weights, there are $n_3^{ex} = 3n_3$ inputs and $n_4^{ex} = n_4$ outputs.
- In the connection matrices $\mathbf{W}^{[ex1]}$, $\mathbf{W}^{[ex2]}$, $\mathbf{W}^{[ex3]}$, many weights are constraint to zero;
- Some other weights must obey equality constraints (e.g. in the first layer connections matrix $\mathbf{W}^{[ex1]}$, the block $\mathbf{W}_0^{[1]}$ appears five times).
- Apply one step of BP algorithm to the MP $\mathbf{W}^{[ex1]}$, $\mathbf{W}^{[ex2]}$, $\mathbf{W}^{[ex3]}$, finding the gradients with respects to all nonzero weights in the matrices $\mathbf{W}^{[ex1]}$, $\mathbf{W}^{[ex2]}$, $\mathbf{W}^{[ex3]}$.

- Find the constraint gradients, with respect to original FIR MP weights, by adding all corresponding gradients in MP.
- change the weights using the constraint gradients.
- iterate until convergence.

Temporal Back -Propagation Algorithm (Wan, 1990)

The change in a weight on a hidden layer, $\Delta\mathbf{w}_{lij}^{[k]}$, will modify the value of $v_i^{[k]}(t)$, which will have effect not only in the computation of J_t, but also in the computation of J_t for some other future time instants.

Then, an alternative updating of the weights may be defined as

$$\mathbf{w}_{lij}^{[k]}(t+1) = \mathbf{w}_{lij}^{[k]}(t) - \mu \frac{dJ}{dv_i^{[k]}(t)} \frac{dv_i^{[k]}(t)}{d\mathbf{w}_{lij}^{[k]}}$$

The gradients

$$\frac{dJ}{dv_i^{[k]}(t)} = -\delta_i^{[k]}(t)$$

for all defined k, i, are now key internal variables (generalized errors) for the updating algorithm. We immediately evaluate the derivative

$$\frac{dv_i^{[k]}(t)}{d\mathbf{w}_{lij}^{[k]}} = u_j^{[k]}(t-l)$$

and the only remaining task is to evaluate $\delta_i^{[k]}(t)$. But we have

$$\begin{aligned}
\frac{dJ}{dv_i^{[k]}(t)} &= \sum_{j=1}^{n_{k+2}} \sum_{n=0}^{2} \frac{dJ}{dv_j^{[k+1]}(t+n)} \frac{dv_j^{[k+1]}(t+n)}{dv_i^{[k]}(t)} = \sum_{j=1}^{n_{k+2}} \sum_{n=0}^{2} \frac{dJ}{dv_j^{[k+1]}(t+n)} \frac{dv_j^{[k+1]}(t+n)}{dy_i^{[k]}(t)} \frac{dy_i^{[k]}(t)}{dv_i^{[k]}(t)} = \\
&= -\sum_{j=1}^{n_{k+2}} \sum_{n=0}^{2} \delta_j^{[k+1]}(t+n) \frac{dv_j^{[k+1]}(t+n)}{dy_i^{[k]}(t)} h'(v_i^{[k]}(t)) = -h'(v_i^{[k]}(t)) \sum_{j=1}^{n_{k+2}} \sum_{n=0}^{2} \delta_j^{[k+1]}(t+n)\mathbf{w}_{nji}^{[k+1]}
\end{aligned}$$

and thus

$$\delta_i^{[k]}(t) \;=\; -\frac{dJ}{dv_i^{[k]}(t)} = h'(v_i^{[k]}(t)) \sum_{j=1}^{n_{k+2}} \sum_{n=0}^{2} \delta_j^{[k+1]}(t+n)\mathbf{w}_{nji}^{[k+1]}$$

leading to the backward generalized error computation

$$\delta_i^{[k]}(t) \;=\; h'(v_i^{[k]}(t)) \sum_{j=1}^{n_{k+2}} \sum_{n=0}^{2} \delta_j^{[k+1]}(t+n)\mathbf{w}_{nji}^{[k+1]} \tag{11}$$

which can be iterated from the output level $k = 3$, where trivially we have

$$\delta_i^{[3]}(t) \;=\; -\frac{dJ}{dv_i^{[3]}(t)} = -\frac{dJ_t}{dv_i^{[3]}(t)} = h'(v_i^{[3]}(t))e_i(t)$$

and then backward we can find all generalized errors, up to the first layer, $k = 1$. The weight update can be summarized as

$$\begin{aligned}
\mathbf{w}_{lij}^{[k]}(t+1) &= \mathbf{w}_{lij}^{[k]}(t) + \mu\frac{dJ}{dv_i^{[k]}(t)}\frac{dv_i^{[k]}(t)}{d\mathbf{w}_{lij}^{[k]}} \\
&= \mathbf{w}_{lij}^{[k]}(t) + \mu\delta_i^{[k]}(t)u_j^{[k]}(t-l)
\end{aligned}$$

Unfortunately, this procedure is not causal, since we need in (11) the future values $\delta_j^{[k+1]}(t+n)$.

Causal Temporal Back -Propagation Algorithm

Iterate for all time instants t, until convergence:

- Compute the forward path, using the weight values at time t, $\mathbf{w}_{lij}^{[k]}(t)$.
- Output layer updating

$$\begin{aligned}
\delta_i^{[3]}(t) &= h'(v_i^{[3]}(t))e_i(t) \\
\mathbf{w}_{lij}^{[3]}(t+1) &= \mathbf{w}_{lij}^{[3]}(t) + \mu\delta_i^{[3]}(t)u_j^{[3]}(t-l)
\end{aligned}$$

- For the uppermost hidden layer $k = 2$

$$\begin{aligned}
\delta_i^{[k]}(t-3) &= h'(v_i^{[k]}(t-3)) \sum_{j=1}^{n_{k+2}} \sum_{n=0}^{2} \delta_j^{[k+1]}(t+n-3)\mathbf{w}_{nji}^{[k+1]}(t) \\
\mathbf{w}_{lij}^{[k]}(t+1) &= \mathbf{w}_{lij}^{[k]}(t) + \mu\delta_i^{[k]}(t-3)u_j^{[k]}(t-l-3)
\end{aligned}$$

- For lower hidden layer $k = 1$

$$\begin{aligned}
\delta_i^{[k]}(t-6) &= h'(v_i^{[k]}(t-6)) \sum_{j=1}^{n_{k+2}} \sum_{n=0}^{2} \delta_j^{[k+1]}(t+n-6)\mathbf{w}_{nji}^{[k+1]}(t) \\
\mathbf{w}_{lij}^{[k]}(t+1) &= \mathbf{w}_{lij}^{[k]}(t) + \mu\delta_i^{[k]}(t-6)u_j^{[k]}(t-l-6)
\end{aligned}$$