

# Paper Review:

Eric Wan and Françoise Beaufays

## **Diagrammatic Derivation of Gradient Algorithms for Neural Networks**

as submitted to *Neural Computation*,  
March 1994 (published in 1996).

# The Main Idea

- Gradient descent algorithms can be derived for a wide variety of neural networks simply from the **signal flow graph** of the network.
- The classes of networks for which this works include time-based ones, such as Backpropagation Through Time and some recurrent networks.

# Cost Function

- The cost function (MSE) is defined as **summed over all time-steps**.

$$J = \sum_{k=1}^K L_k(\mathbf{d}(k), \mathbf{y}(k))$$

- Here  $L_k(d(k), y(k)) = e(k)e(k)^T$ , where  $e(k) = d(k) - y(k)$ , although this can be generalized to other  $L_k$ .
- $d(k)$  is the **desired** value at step  $k$ .
- $y(k)$  is the **actual** output at step  $k$ .

# Using Derivatives for Weight Changes

$$J = \sum_{k=1}^K L_k(\mathbf{d}(k), \mathbf{y}(k))$$

According to gradient descent, the contribution to the weight update at each time step is

$$\Delta W(k) = -\mu \frac{\partial J}{\partial W(k)}, \quad (2)$$

where  $\mu$  controls the learning rate. Note that we evaluate  $\partial J / \partial W(k)$  rather than the instantaneous gradient  $\partial(\mathbf{e}^T(k)\mathbf{e}(k)) / \partial W(k)$ . This is essential for the desired Network Reciprocity result.

---

(This method does not seem applicable to *on-line* training, as in one version of RTRL.)

# Signal Values and Sensitivities

- The values  $a_i(k)$  represent the value of a **signal** at time-step  $k$  at some point in the network, e.g.  $y(k)$  in

$$J = \sum_{k=1}^K L_k(\mathbf{d}(k), \mathbf{y}(k))$$

- The values  $\delta_i(k)$  represent the corresponding **sensitivities** (partial derivatives of the MSE  $J$  with respect to the signal  $a_i(k)$ ).

$$\delta_j(k) \triangleq \frac{\partial J}{\partial a_j(k)}$$

## Use of Sensitivities: Weight Updating

By definition,  $w_{ij}$  relates  $a_j$  to  $a_i$ :  $a_j = w_{ij}a_i$ .

Similar to the backpropagation derivation, employ the chain rule, in conjunction with the above:

$$\frac{\partial J}{\partial w_{ij}(k)} = \frac{\partial J}{\partial a_j(k)} \frac{\partial a_j(k)}{\partial w_{ij}(k)} = \frac{\partial J}{\partial a_j(k)} a_i(k),$$

$$\Delta w_{ij}(k) = -\mu \delta_j(k) a_i(k)$$

# Sensitivity of the Output Signal

- Sensitivity of the output signals  $y$  are

$$-2e(k)$$

= derivative of  $(d(k) - y(k))^2$

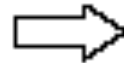
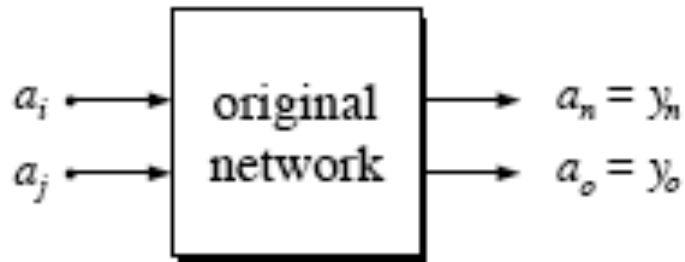
where  $e(k) = d(k) - y(k)$ .

$$J = \sum_{k=1}^K L_k(\mathbf{d}(k), \mathbf{y}(k))$$

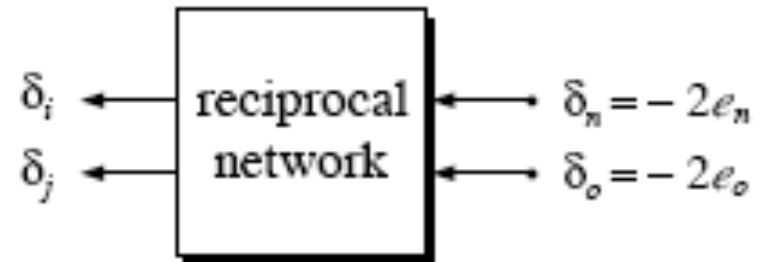
$$L_k(d(k), y(k)) = e(k)e(k)^T$$

# The General Derivation Setup

Given



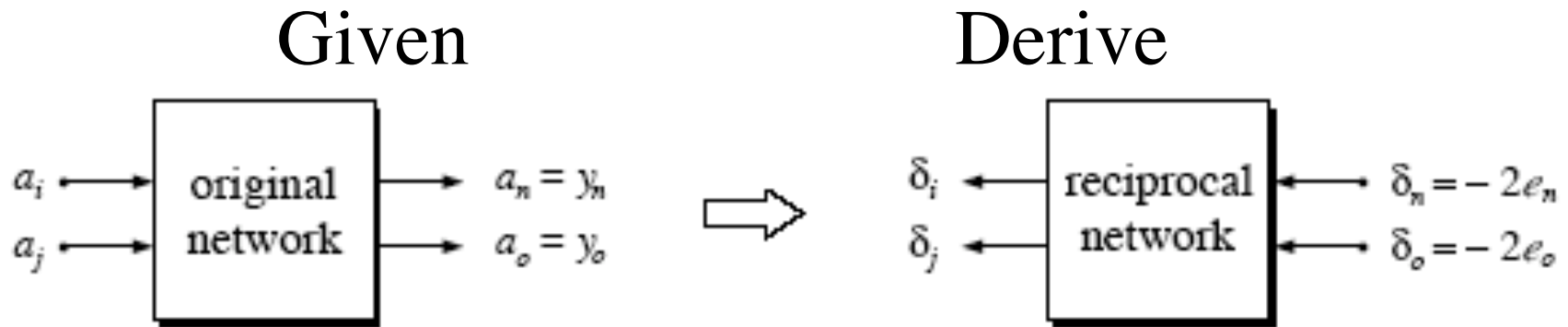
Derive



for purposes of  
computing  $\delta$ 's.

This network is  
called the “**reciprocal**”  
of the original.

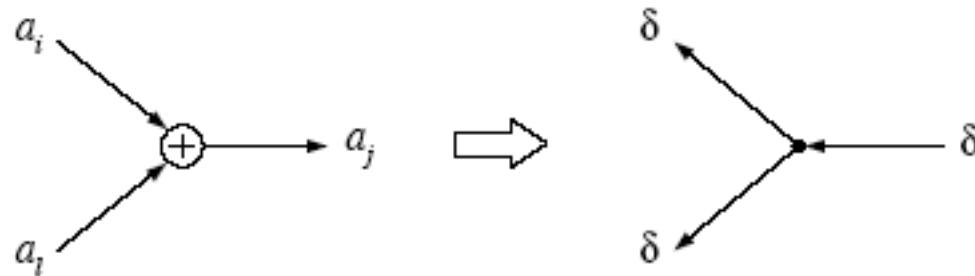
A form of *Induction* is implicit:



## Strategy: Develop expressions for sensitivity based on network building blocks

- summing junctions
- branch points
- unit time delays (only discrete-time systems are considered)
- functions (univariate and multivariate)

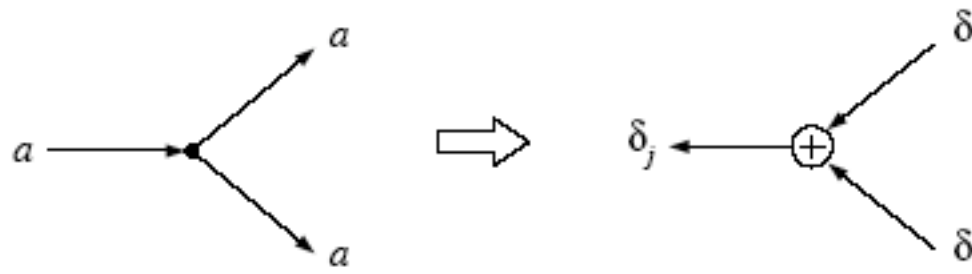
# Summing Junction Rule



The reciprocal of a summing junction is a branch-point.

(Justify by partial derivatives.)

# Branch-Point Rule



The reciprocal of a branch-point is a summing junction.

# Univariate Function Rule



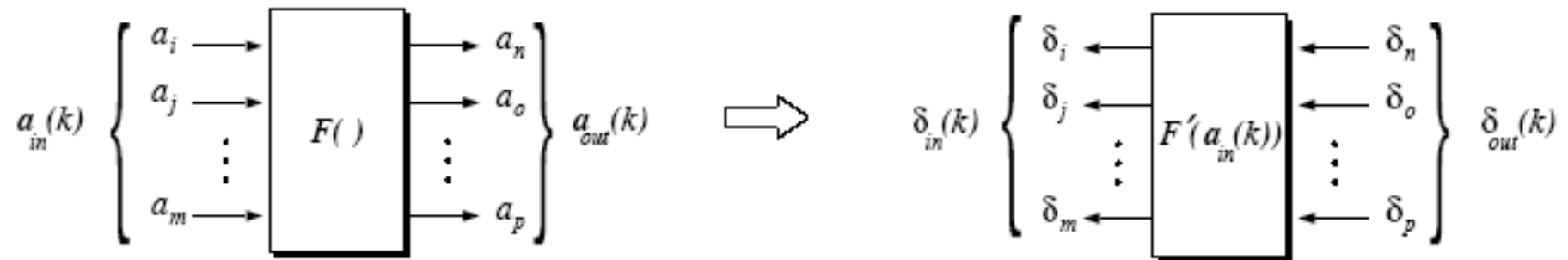
The reciprocal network for a univariate function is the function's derivative, evaluated at the original input.

Weights are a special case of a function (namely scalar-multiply)

$$a_i \xrightarrow{w_{ij}} a_j \quad \Rightarrow \quad \delta_i \xleftarrow{w_{ij}} \delta_j$$

$$a_i(k) \rightarrow \boxed{f(\cdot)} \rightarrow a_j(k) \quad \Rightarrow \quad \delta_i(k) \leftarrow \boxed{f'(a_i(k))} \leftarrow \delta_j(k)$$

# Multivariate Function Rule



The reciprocal of a multivariate function  $F$  is the function's Jacobian  $F'$ , evaluated at the original inputs.

[The Jacobian is the  $m \times p$  matrix of partial derivatives.]

(Note that summing junctions, branch points, and univariate functions are all special cases of multivariate functions.)

# Unit-Delay Rule

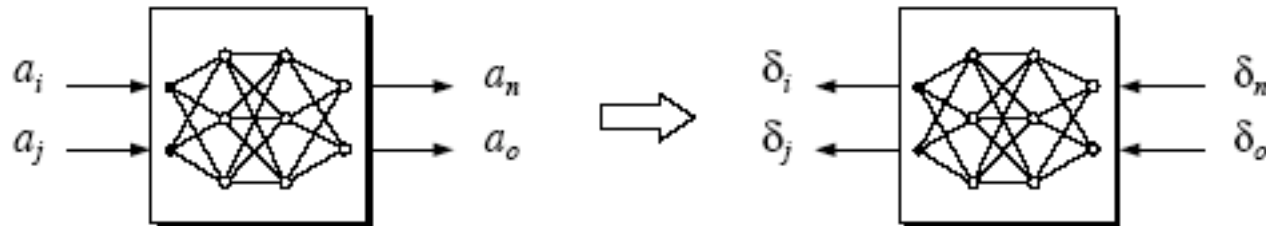
$$a_i(k) \longrightarrow \boxed{q^{-1}} \longrightarrow a_j(k) = a_i(k-1) \quad \Rightarrow \quad \delta_i(k) = \delta_j(k+1) \longleftarrow \boxed{q^{-1}} \longleftarrow \delta_j(k)$$

Note:  $q^{-1}$  is signal flowgraph notation for unit time delay.  
It is analogous to  $z^{-1}$  in z-transforms.

$q^{+1}$  is the “time-advance” operator.

So the *future* sensitivity is back-propagated through a unit delay.

# Sub-Networks, Layers



Sub-networks can be treated as a single functional unit by back-propagating sensitivities through the sub-network, i.e. by applying this entire process recursively.

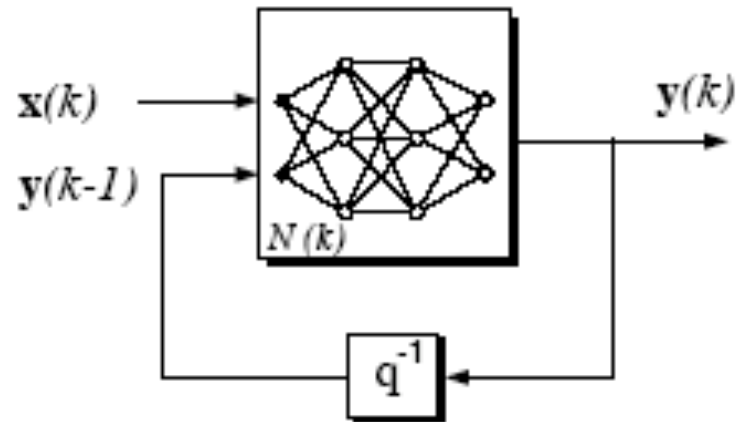
# Example: Standard Backpropagation

A graphic derivation yields the (time-insensitive) update formulae:

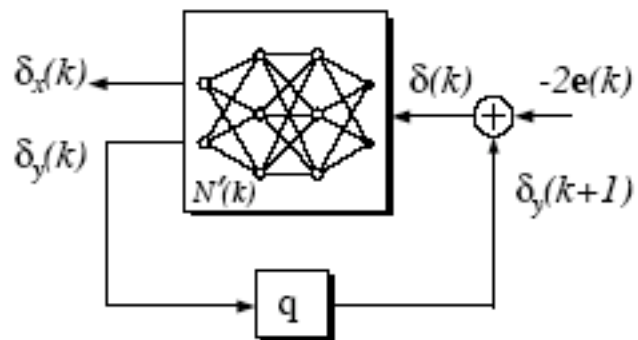
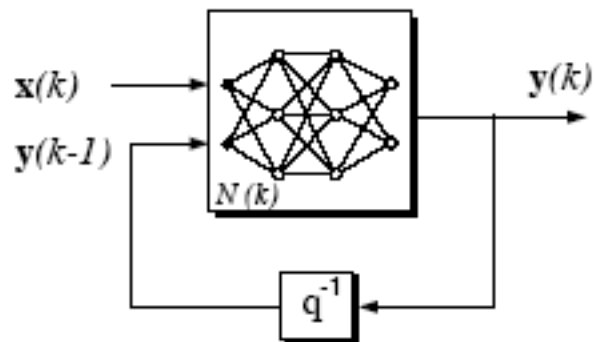
$$\delta_i^l = \begin{cases} -2e_i f'(s_i^L) & l = L \\ f'(s_i^l) \cdot \sum_j \delta_j^{l+1} \cdot w_{ij}^{l+1} & 0 \leq l \leq L - 1 \end{cases}$$

$$\Delta w_{pi}^l = -\mu \delta_i^l a_p^{l-1}$$

# Example: Backpropagation Through Time



# Construct the Reciprocal Network

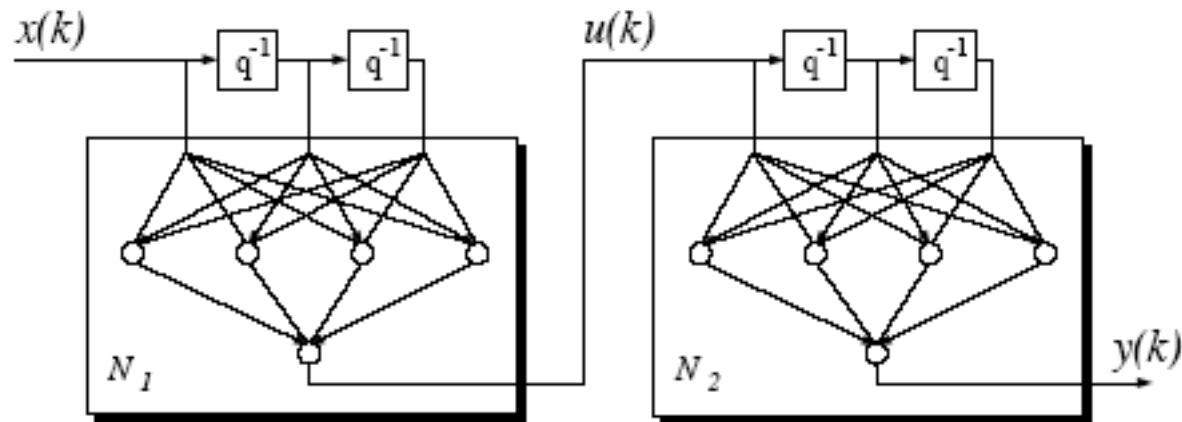


Derived recurrence formula for the *loop value*  $\delta(k)$ :

$$\begin{aligned}\delta(k) &= \delta_y(k+1) - 2e(k) \\ &= N'(k+1)\delta(k+1) - 2e(k)\end{aligned}$$

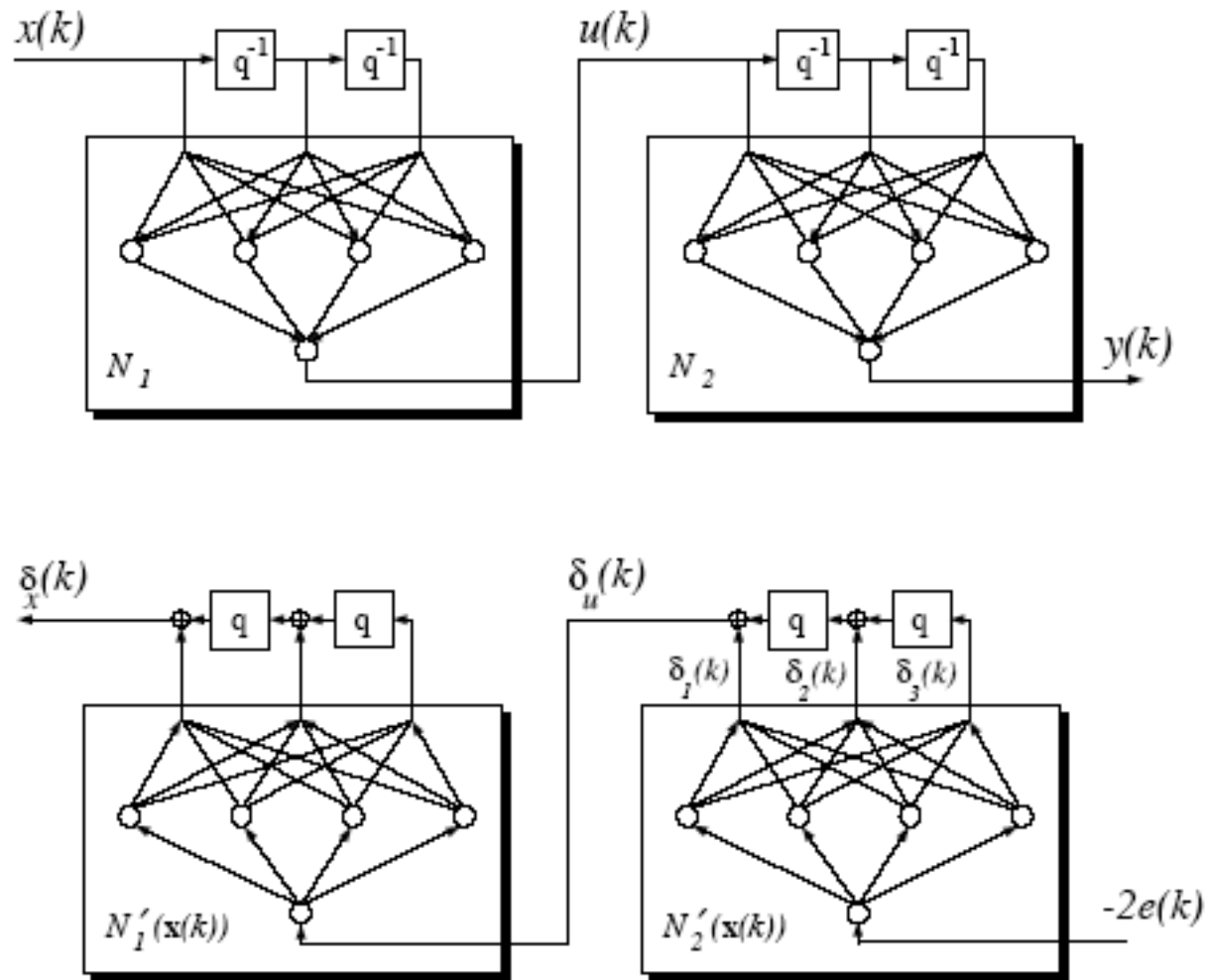
Because the above formula “looks ahead” in time, it can only be used when *the number of time steps is known in advance*.

## Example: Cascaded Networks with Delay Lines

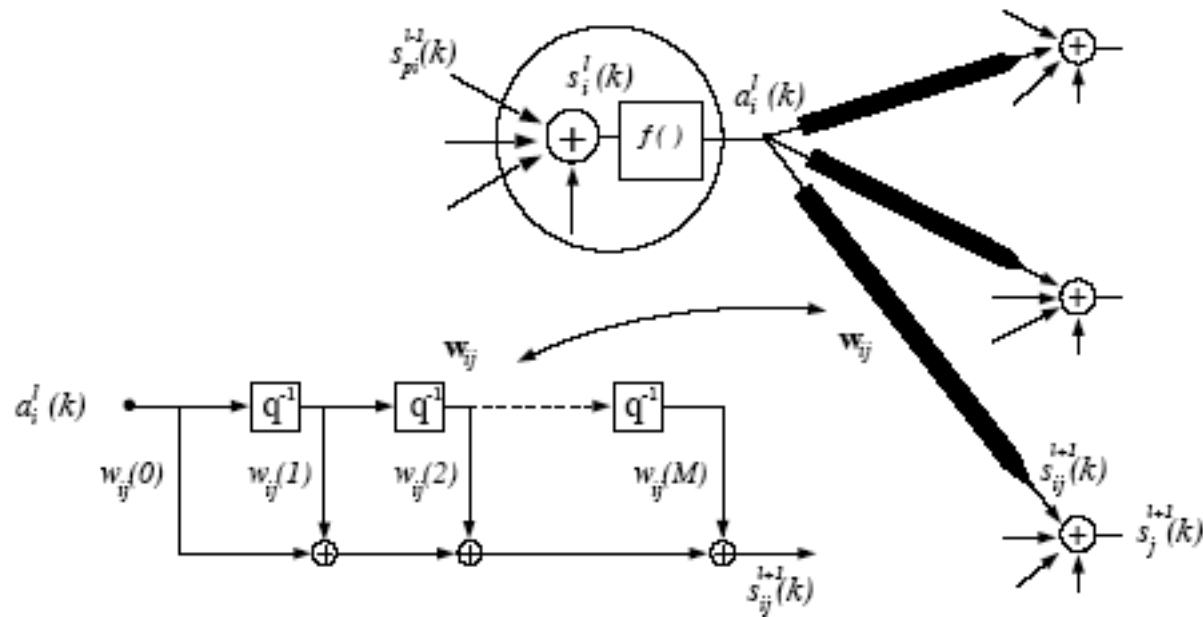


The  $\delta$  corresponding to  $u$  from  $N_2$  is used as if the output error for  $N_1$  in backpropagating.

# Corresponding Reciprocal Network

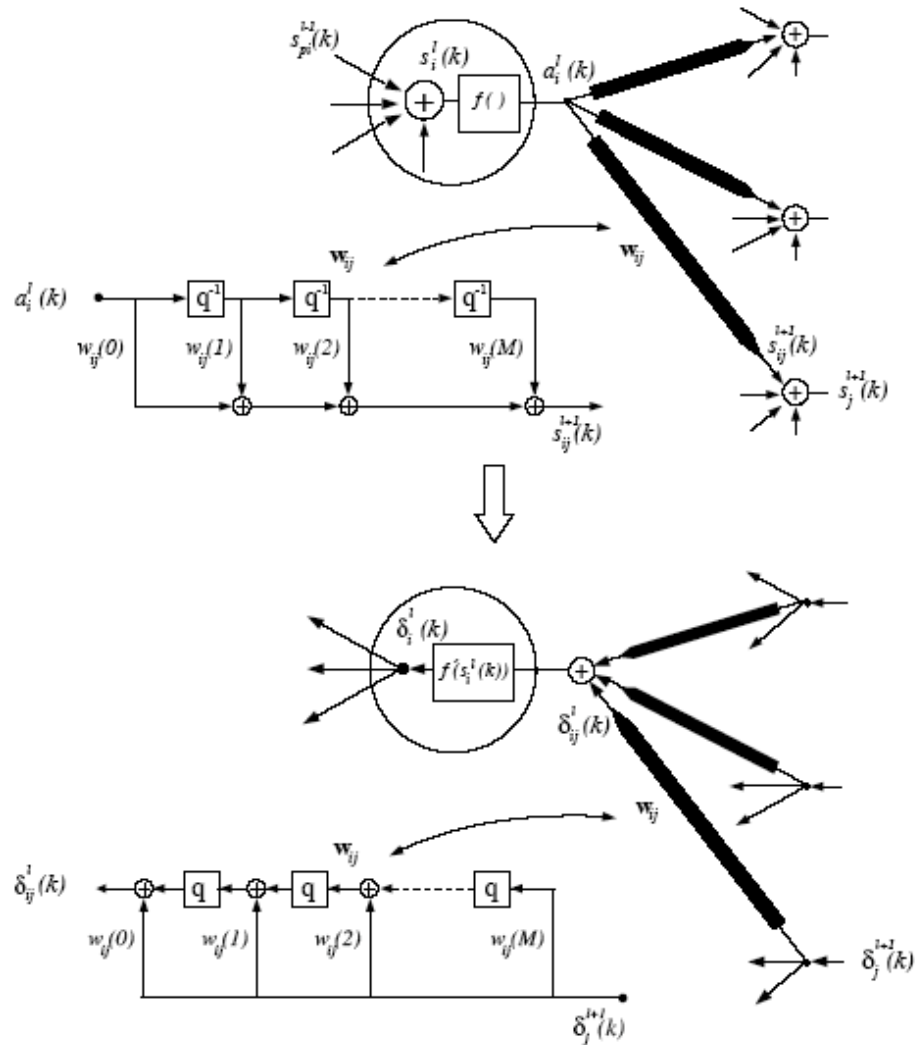


# Example: FIR Network (Wan, 1993)



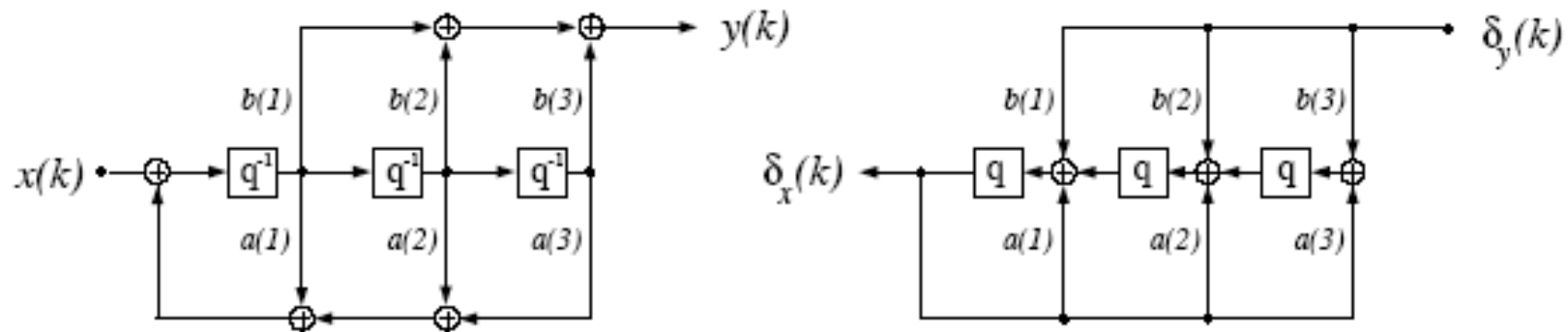
Weights in a standard MLP are replaced with FIR (Finite Impulse Response) filters.

# Reciprocal Network



# Example: IIR Network

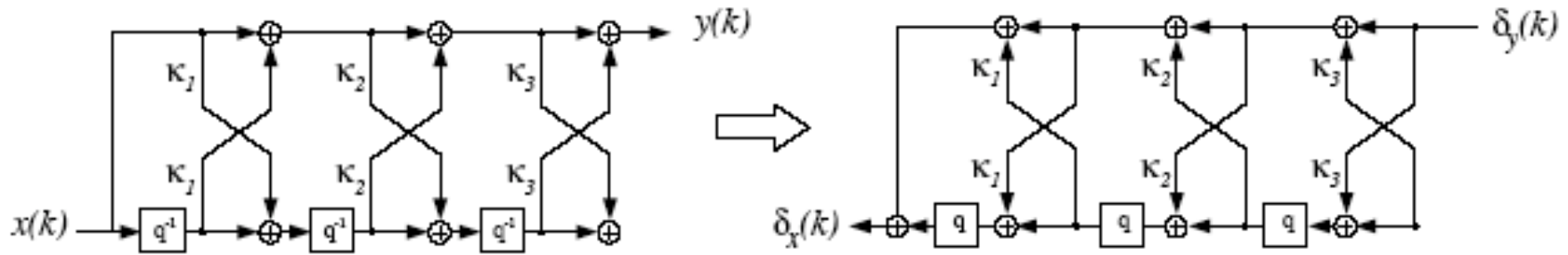
As in FIR case, but weights are replaced with IIR (Infinite Impulse Response) filters.



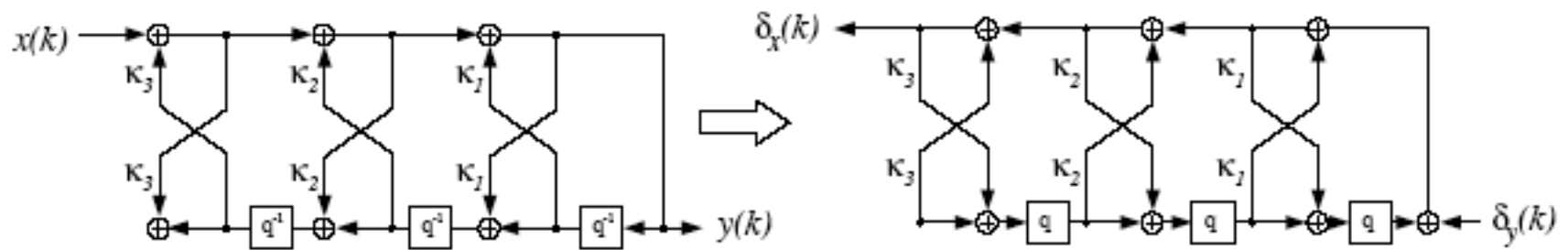
given: 
$$y(k) = \sum_{m=1}^M a(m)y(k-m) + \sum_{m=0}^M b(m)x(k-m) = \frac{\sum_{m=0}^M b(m)q^{-m}}{1 - \sum_{m=1}^M a(m)q^{-m}} x(k)$$

derived: 
$$\delta_x(k) = \sum_{m=1}^M a(m)\delta_x(k+m) + \sum_{m=0}^M b(m)\delta_y(k+m) = \frac{\sum_{m=0}^M b(m)q^{+m}}{1 - \sum_{m=1}^M a(m)q^{+m}} \delta_y(k)$$

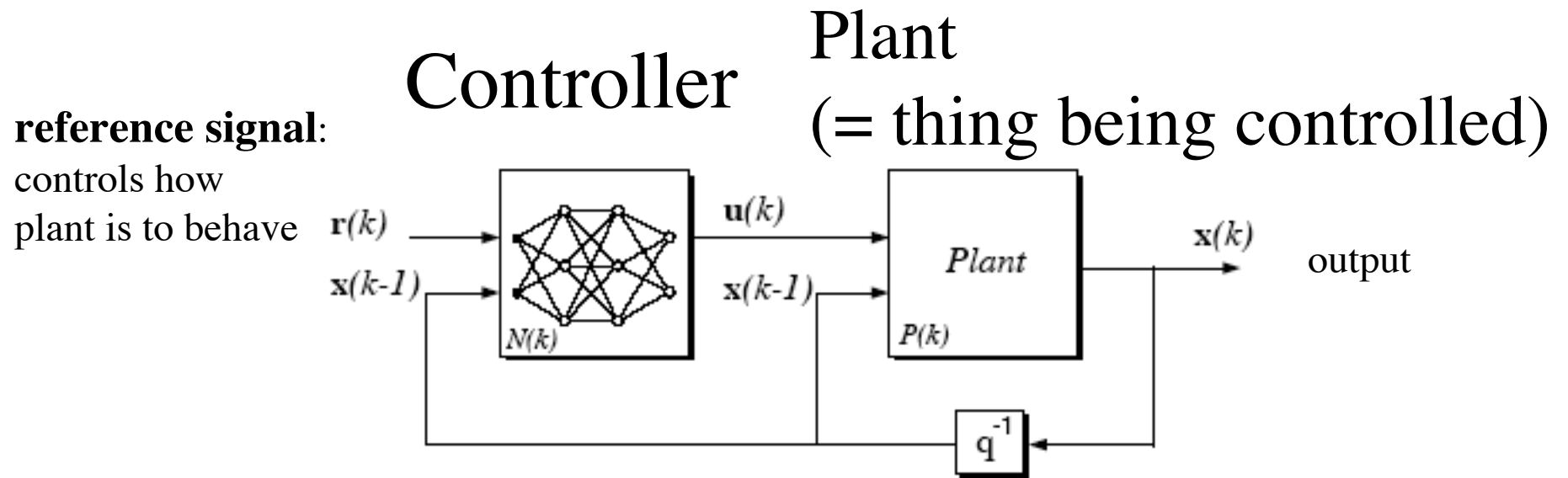
# Other Options: Lattice FIR



# Other Options: Lattice IIR



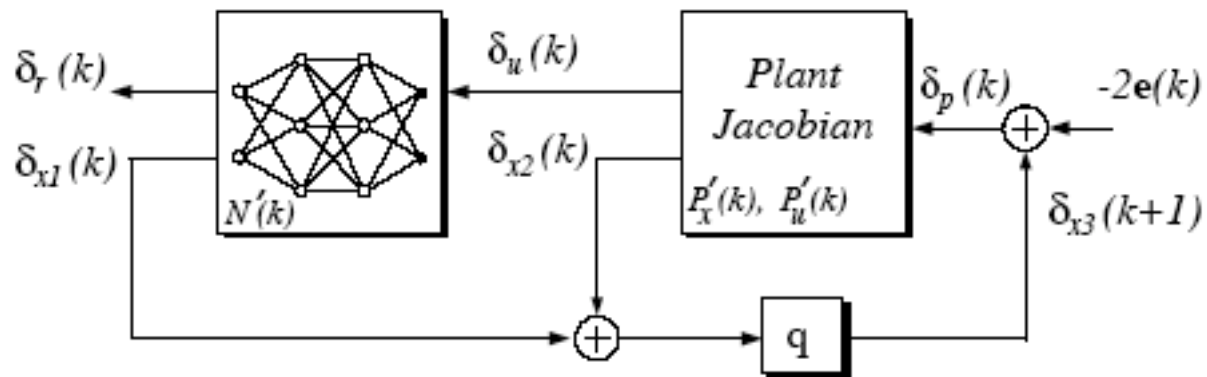
# Application: Neural Controllers



The approach is again like BPTT.

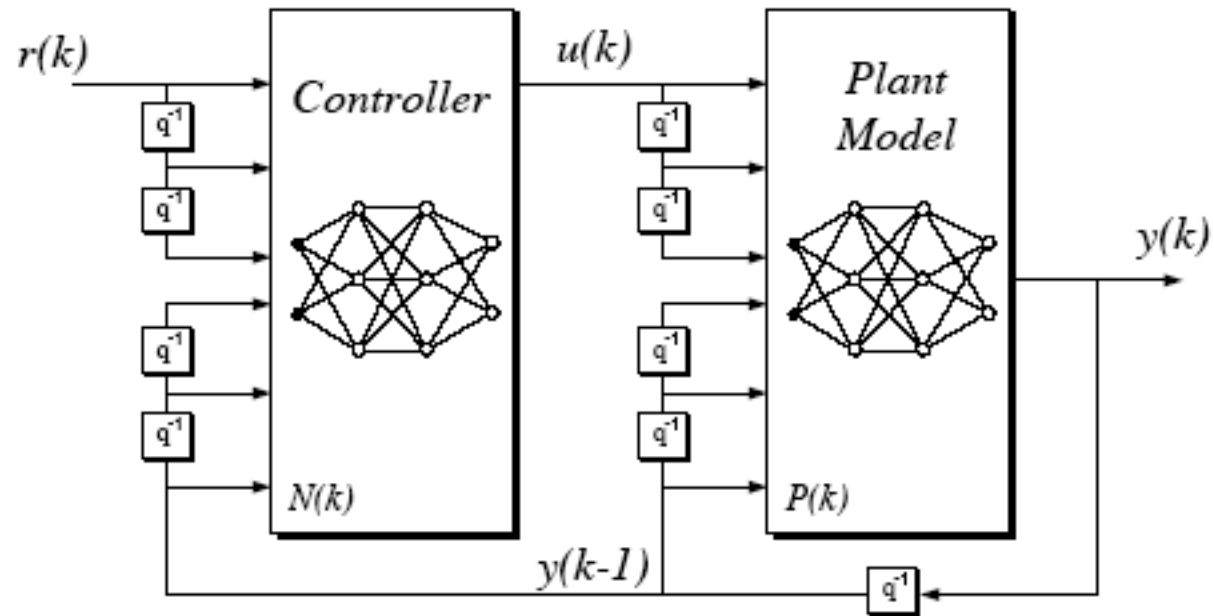
# Training the Controller

If the Jacobian of the plant can be computed, the controller can be trained thus:

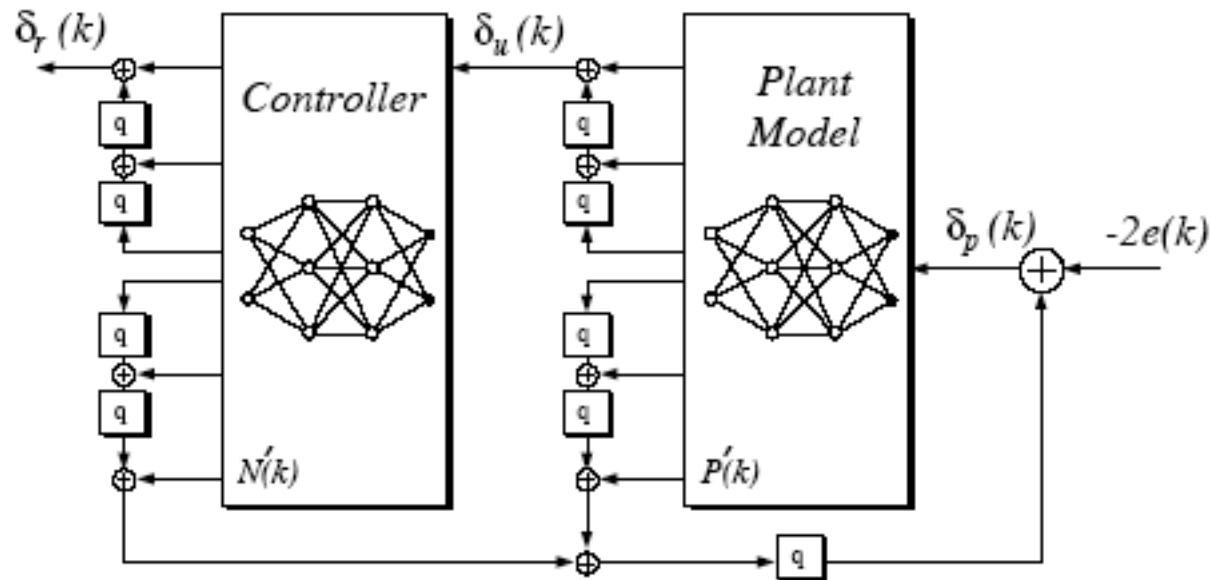


*A neural model* could be substituted for the plant, as in the truck-backer problem.

# Example: Controlling with a NARMA (nonlinear, autoregressive, moving average) filter



# Reciprocal network for the NARMA case



# Summary

- Clearly an unlimited set of network configurations of a wide variety can be trained by this approach.
- In some cases (e.g. cascaded networks with time delays), the reciprocal approach is computationally more efficient than previously-presented methods.
- The authors offer a proof that their method is correct.

# Proof Idea

- The main issue seems to be what to do about time delay operators. If there were none, it would be simple structural induction (on the structure of the network).
- If a perturbation  $\Delta a^*(k)$  were applied to some node  $a^*(k)$  in the network, then we want to determine the effect on  $J$  in the form  $\partial J / \partial a^*(k)$ , which is  $\delta^*(k)$ .
- We want to find these values  $\delta^*(k)$  for *all* nodes in the network.

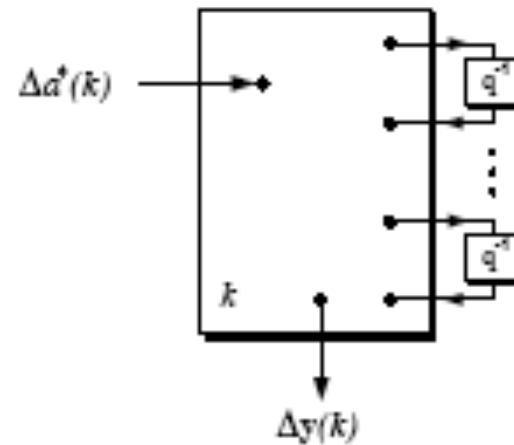
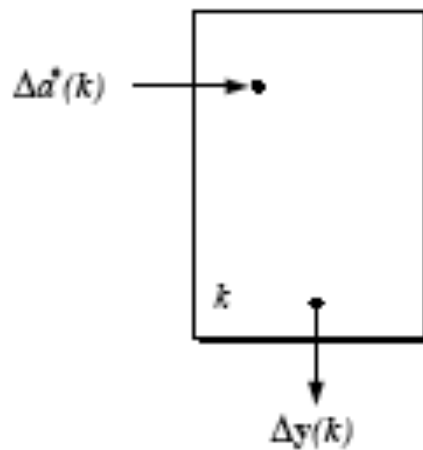
# Proof Idea

- The signals in the network are inter-related by sets of **equations** (e.g. at a summing junction, at a branch point, at a delay, etc.).
- For each such equation, we can propagate the perturbation  $\Delta$ . For example, through a delay with  $a_i$  as input and  $a_j$  as output, we will have

$$\Delta a_j(k) = \Delta a_i(k-1).$$

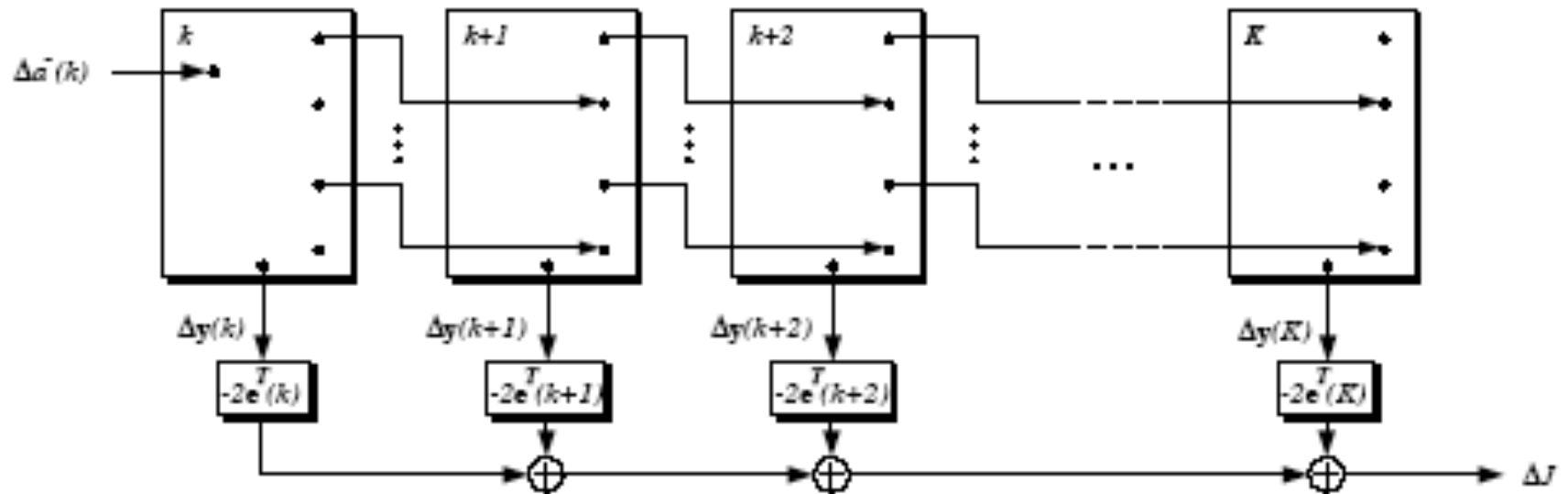
# Proof Idea

- Drag all delays to the *outside* of the network.



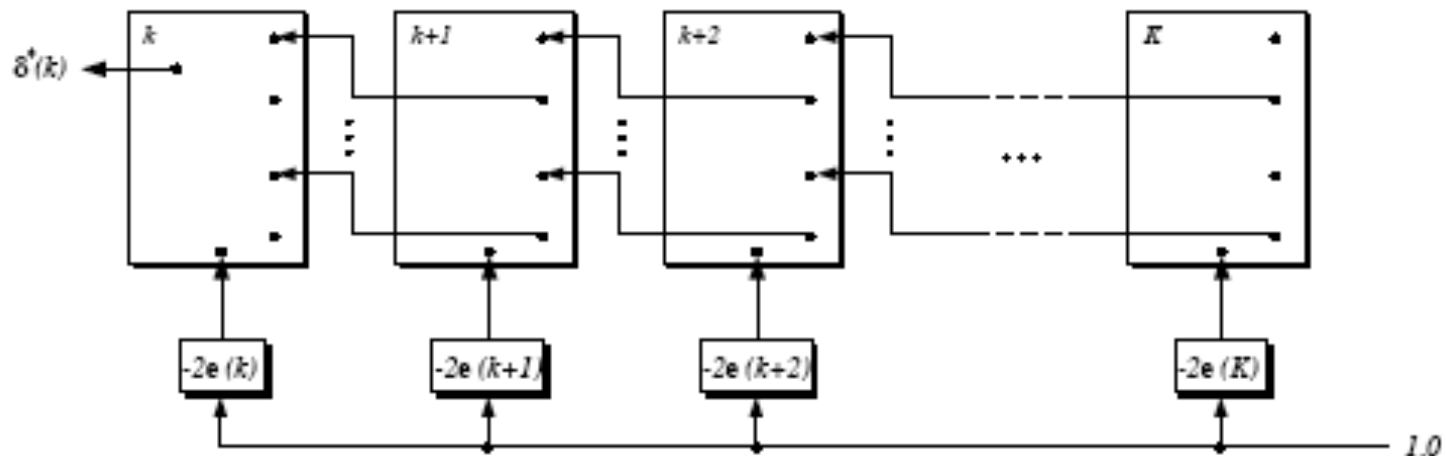
# Proof Idea

- Then **unfold** the resulting network, to get rid of delays entirely, using the error values at different time steps, up to the total number of time steps  $K$ .



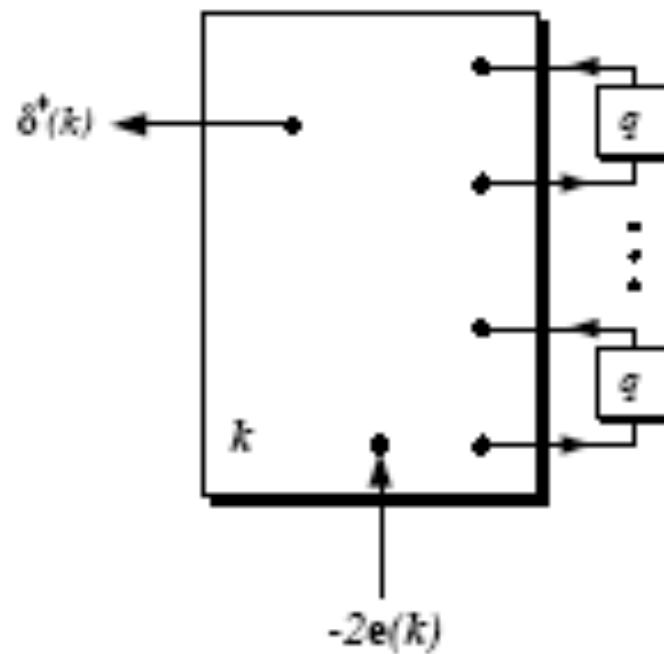
# Proof Idea

- Then derive  $\delta^*(k)$  at the input to the unfolded network, using the obvious reciprocal relationships. (Similar to the network reciprocity theorem of Tellegen, 1952.)



# Proof Idea

- Then **re-fold** to get back to the original network.



# Key Assumption

- (Network can be cyclic).
- We assume that the number of time steps is finite, so we only have to unfold that many copies of the network, at which point we can assume that loop variables are at their initial values.

# Other Work

- A number of papers appear to extend or improve upon this one, including:
  - Another paper by these authors in 1998, introducing “gradient flow graphs”.
  - Papers by Campolucci, et al., including RTRL.
  - Paper by Atiya and Parlos, 2000.
  - PhD Thesis by Sona, 2002.
  - Graph Transformation Work

# Project Idea

- Develop a *modular* software implementation for network learning based on this method.
- (Jonathan Beall did that in his project in Spring 2005.)