Paper Review:

Eric Wan and Françoise Beaufays

Diagrammatic Derivation of Gradient Algorithms for Neural Networks

Neural Computation (8) 1994, 182-201

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 Back-Propagation Through Time and some (but not all) forms of 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 the squared error at the kth step is $L_k(d(k), y(k)) = e(k)e(k)^T$, where $e(k) = d(k)–y(k)$.
- $d(k)$ 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

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

where μ 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.

(Thus this method does not seem applicable to *on-line* training, as in one version of RTRL: Real-Time Recurrent Learning.)

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 $(d(k) - y(k))^2$

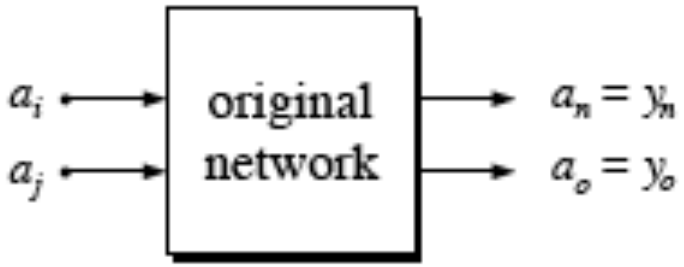
where $e(k) = d(k) - y(k)$.

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

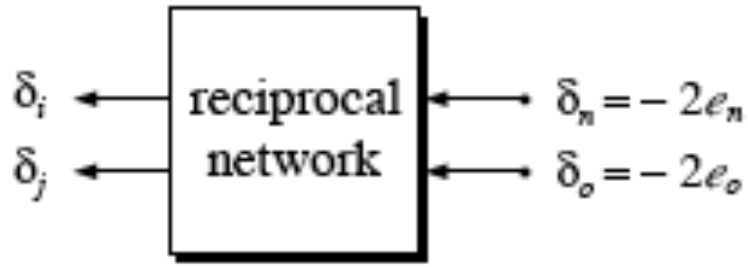
recall $L_k(d(k), y(k)) = e(k)e(k)^T$

The General Derivation Technique: Reciprocity

Given



Derive



(Similar to the
network reciprocity theorem
for electrical networks,
Tellegen, 1952.)

for purposes of
computing δ's.

This network is
called the "**reciprocal**"
of the original.

aside on Tellegen's Theorem (Bernard Tellegen, 1952)

[wikipedia]
Tellegen's theorem is one of the **most powerful theorems in network theory**. Most of the energy distribution theorems and extremum principles in network theory can be derived from it. Fundamentally, Tellegen's theorem gives a simple relation between magnitudes that satisfy the Kirchhoff's laws of electrical circuit theory.

The Tellegen theorem is applicable to a multitude of network systems. The basic assumptions for the systems are the conservation of flow of extensive quantities (Kirchhoff's current law, KCL) and the uniqueness of the potentials at the network nodes (Kirchhoff's voltage law, KVL). The Tellegen theorem provides a useful tool to analyze complex network systems among them electrical circuits, biological and metabolic networks, pipeline flow networks, and chemical process networks.

Consider an arbitrary lumped network whose graph G has b branches and n_t nodes. In an electrical network, the branches are two-terminal components and the nodes are points of interconnection. Suppose that to each branch of the graph we assign arbitrarily a branch potential difference W_k and a branch current F_k for $k = 1, 2, \ldots, b$, and suppose that they are measured with respect to arbitrarily picked *associated* reference directions. If the branch potential differences $W_1, W_2, \ldots, W_b$ satisfy all the constraints imposed by KVL and if the branch currents $F_1, F_2, \ldots, F_b$ satisfy all the constraints imposed by KCL, then

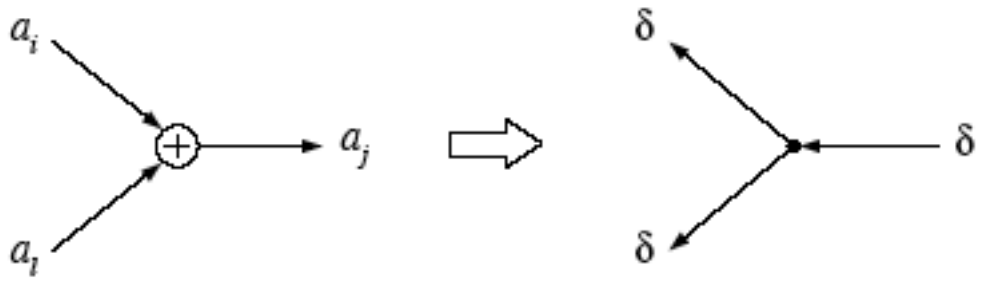
$$\sum_{k=1}^{b} W_k F_k = 0.$$

Tellegen's theorem is extremely general; it is valid for any lumped network that contains any elements, *linear or nonlinear*, *passive or active*, *time-varying or time-invariant*. The generality is extended when W_k and F_k are linear operations on the set of potential differences and on the set of branch currents (respectively) since linear operations don't affect KVL and KCL. For instance, the linear operation may be the average or the Laplace transform. Another extension is when the set of potential differences W_k is from one network and the set of currents F_k is from an entirely different network, so long as the two networks have the same topology (same incidence matrix). This extension of Tellegen's Theorem leads to many theorems relating to two-port networks.[1]

Strategy: Develop expressions for sensitivities inductively, based on network building blocks

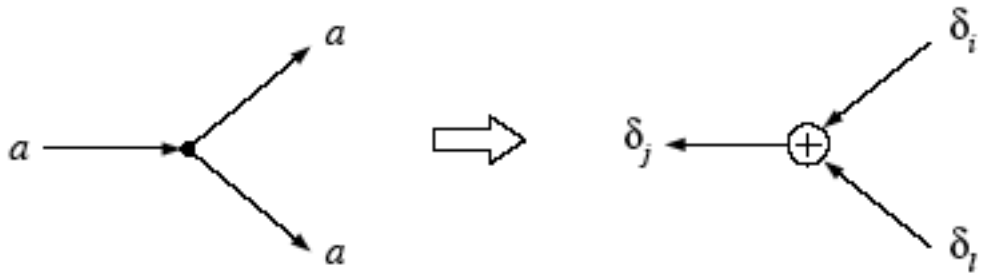
- summing junctions
- branch points
- unit time delays (only discrete-time systems are considered)
- functions (univariate and multivariate)

Summing Junction Rule



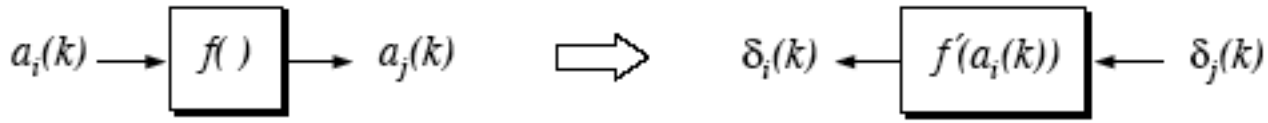
The reciprocal of a summing junction is a branch-point.
(Justify by partial derivatives.)

Branch-Point Rule



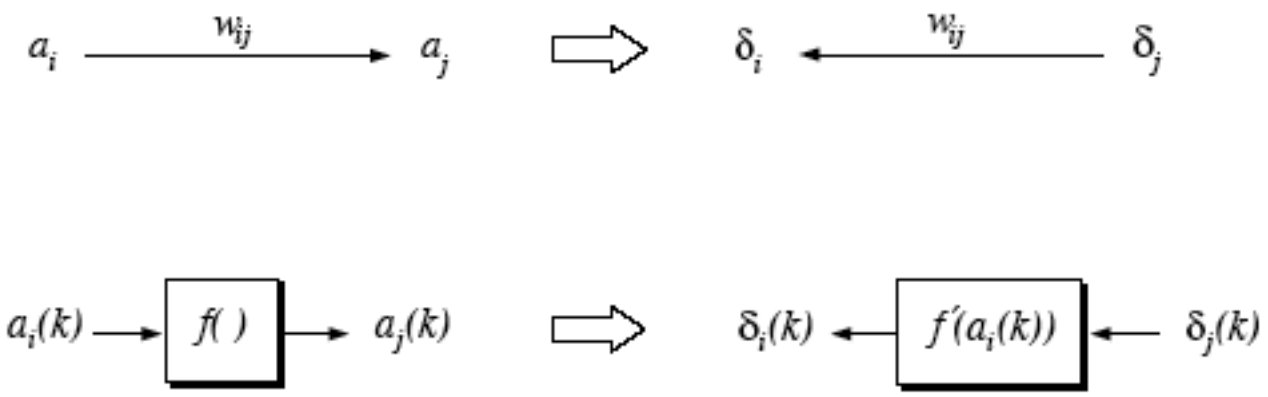
The reciprocal of a branch-point is a summing junction.

Univariate Function Rule

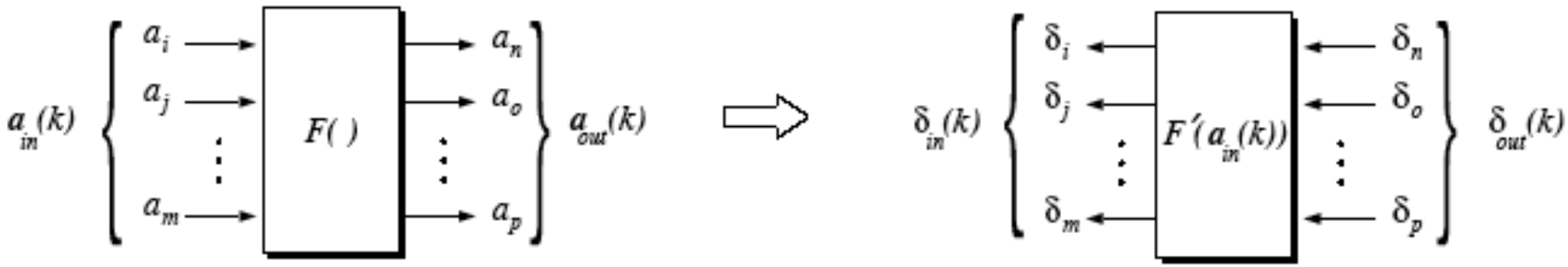


The reciprocal network for a univariate function is the function's derivative, evaluated at the original input.

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



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 x p matrix of partial derivatives.]

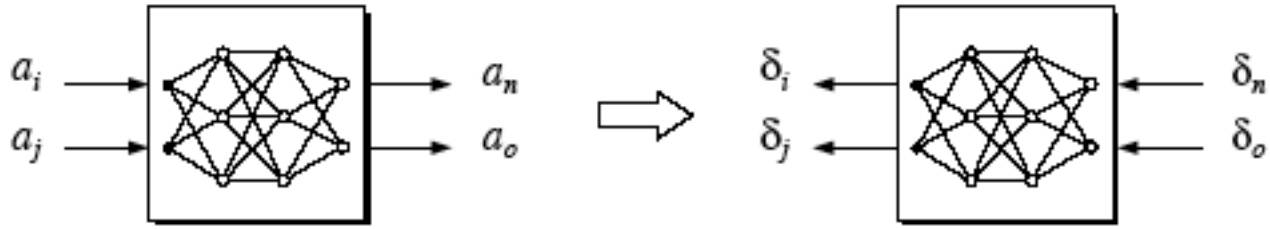
(Note that summing junctions, branch points, and univariate functions are all special cases of multivariate functions.)

Unit-Delay Rule

Note: q^{-1} is signal flowgraph notation for unit time delay.

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/inductively.

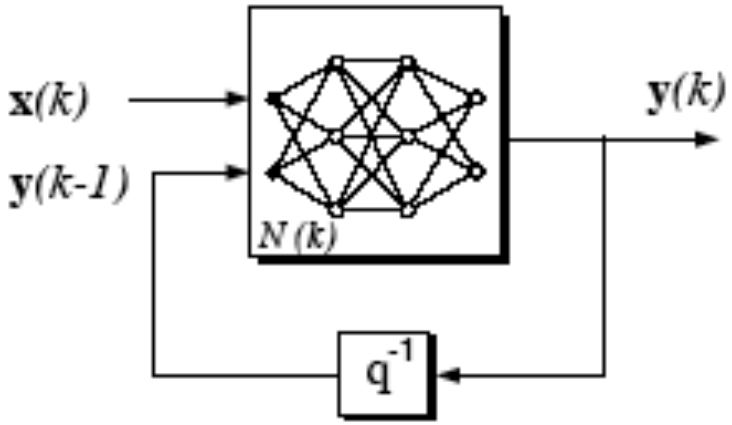
Example: Standard Backpropagation

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

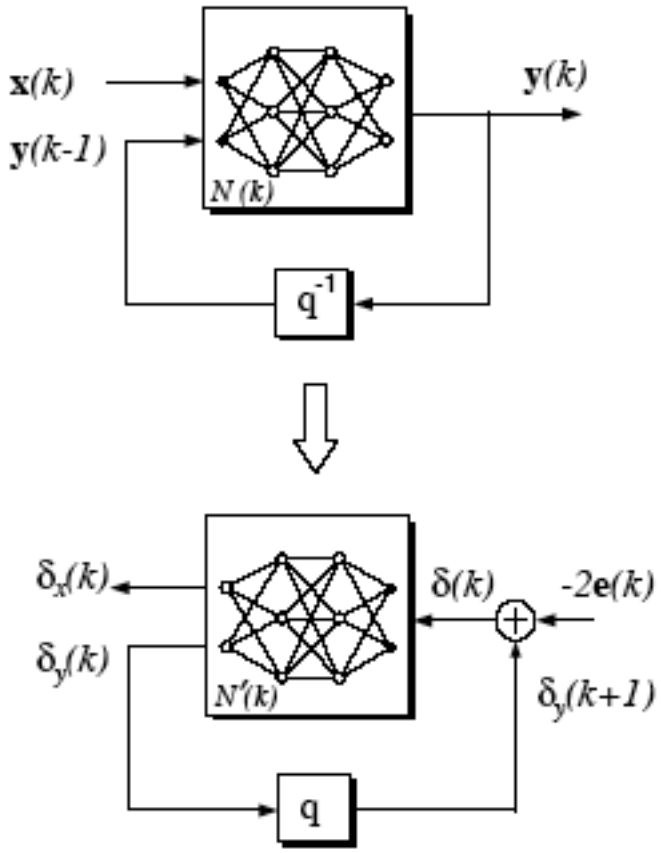
$$\delta_i^l = \begin{cases} -2e_i\, f'(s_i^L) & l = L \\ f'(s_i^l) \cdot \sum_j \delta_j^{l+1} \cdot w_{ij}^{l+1} & 0 \le l \le L-1 \end{cases}$$

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

Example: Backpropagation Through Time



BPTT: Construct the Reciprocal Network

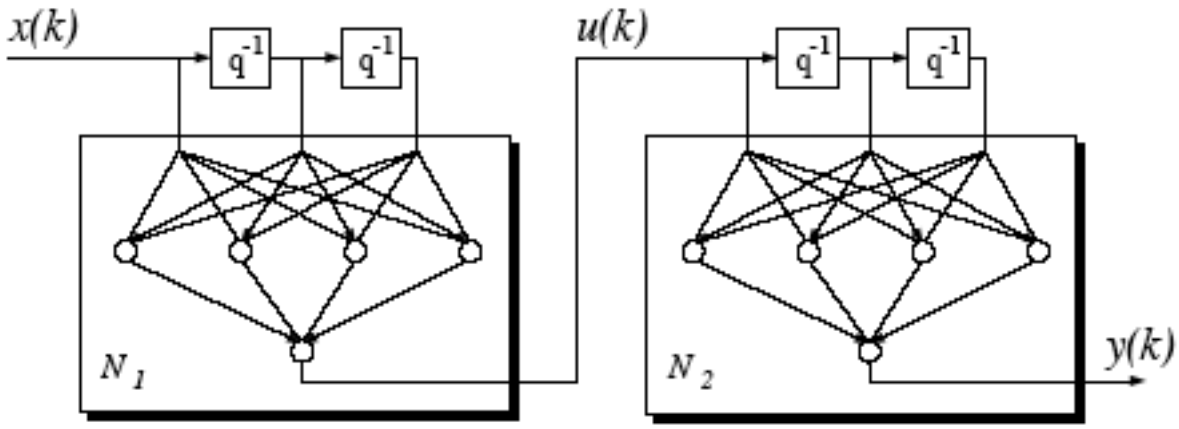


Derived recurrence formula for the *loop value* δ(k):

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

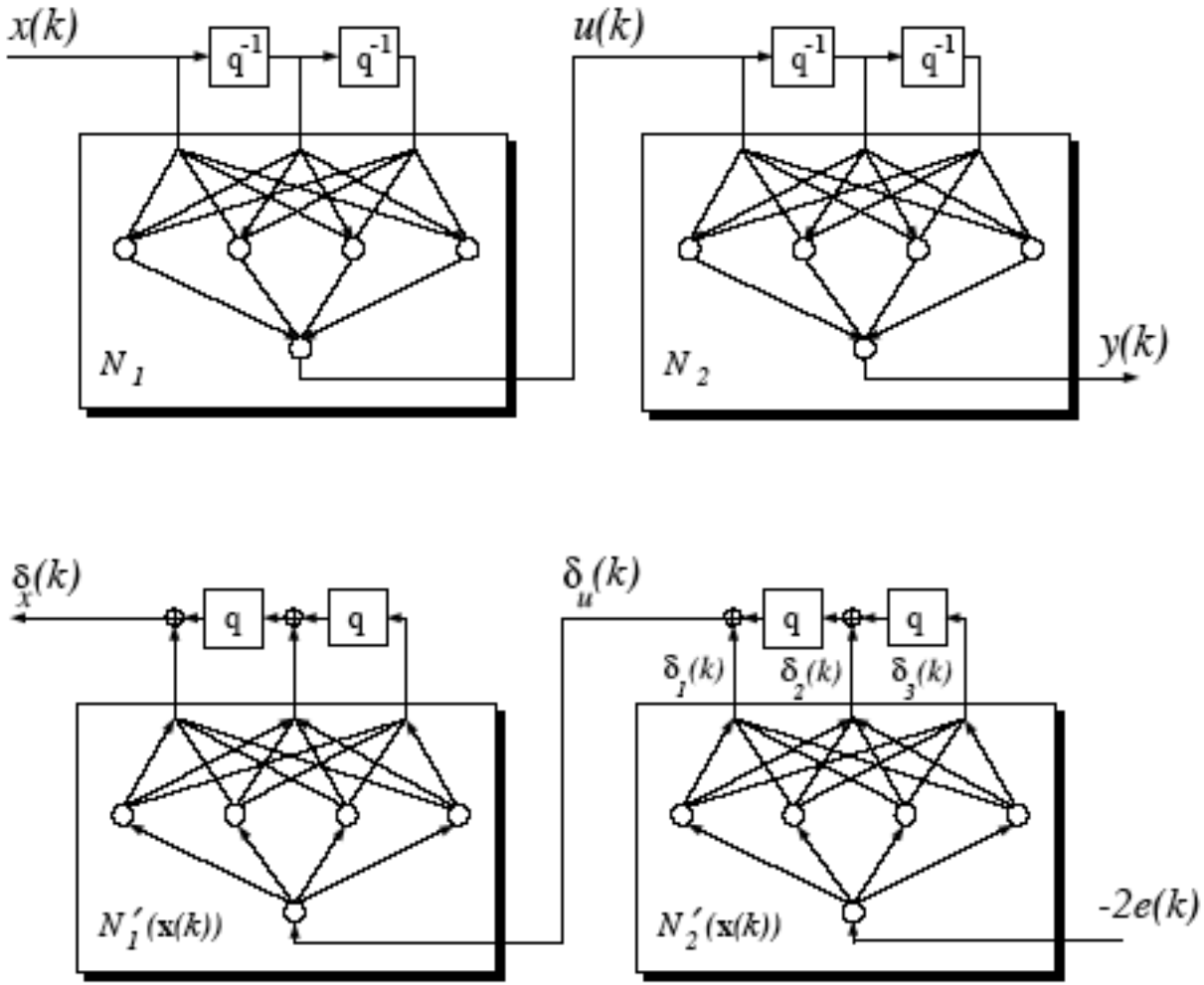
Because the above formula "looks ahead" in time, **it can only be used when *the number of time steps is pre-determined*.**

Example: Cascaded Networks with Delay Lines

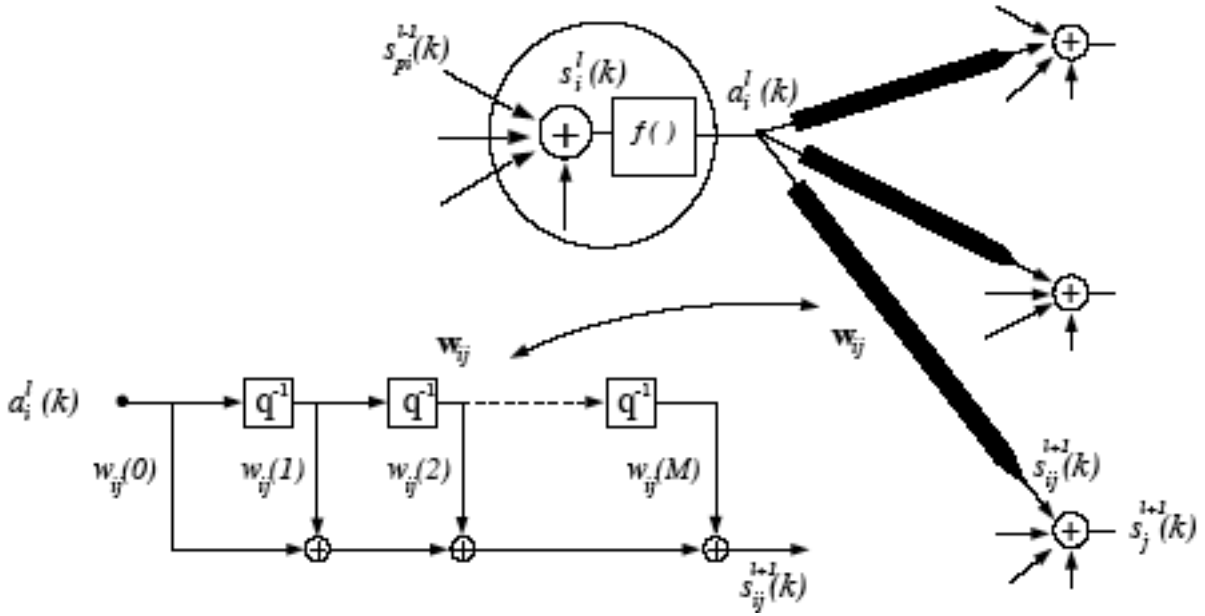


The δ corresponding to u from N_2 is used
as if the output error for N_1 in back-propagating.

Corresponding Reciprocal Network

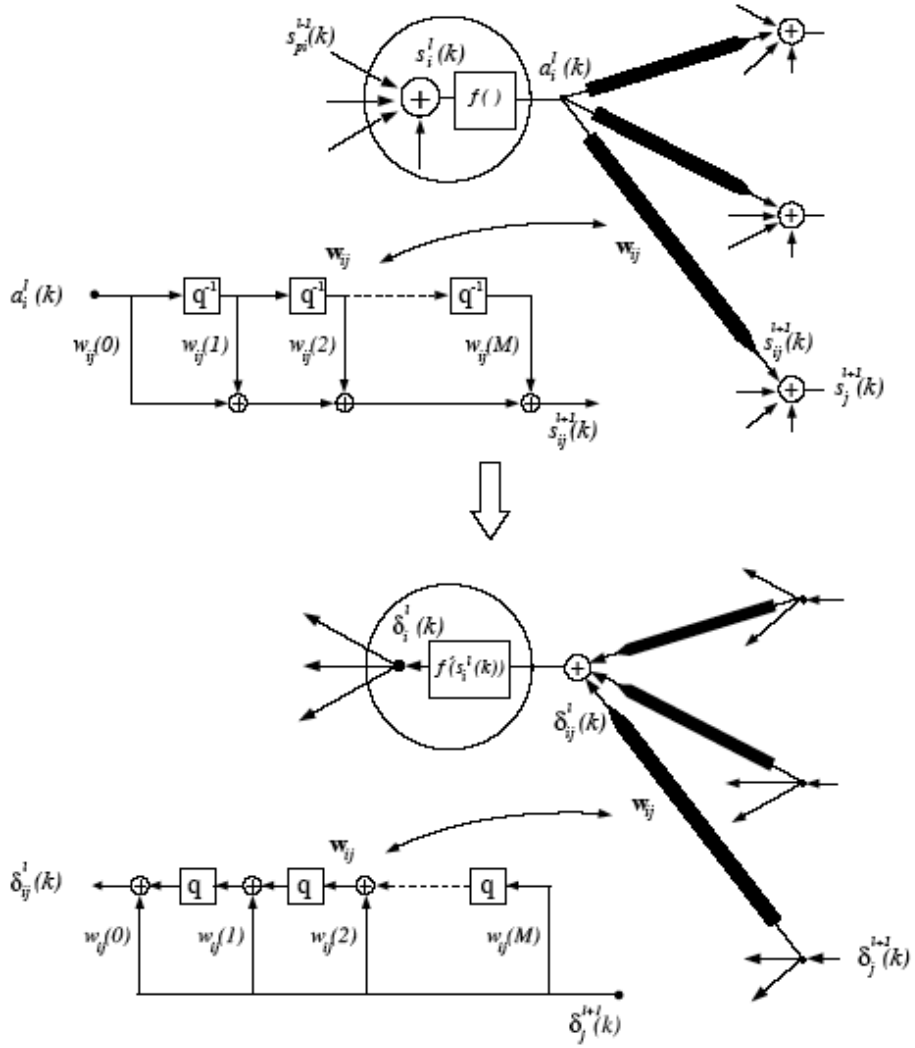


Example: FIR Network (Wan, 1993)



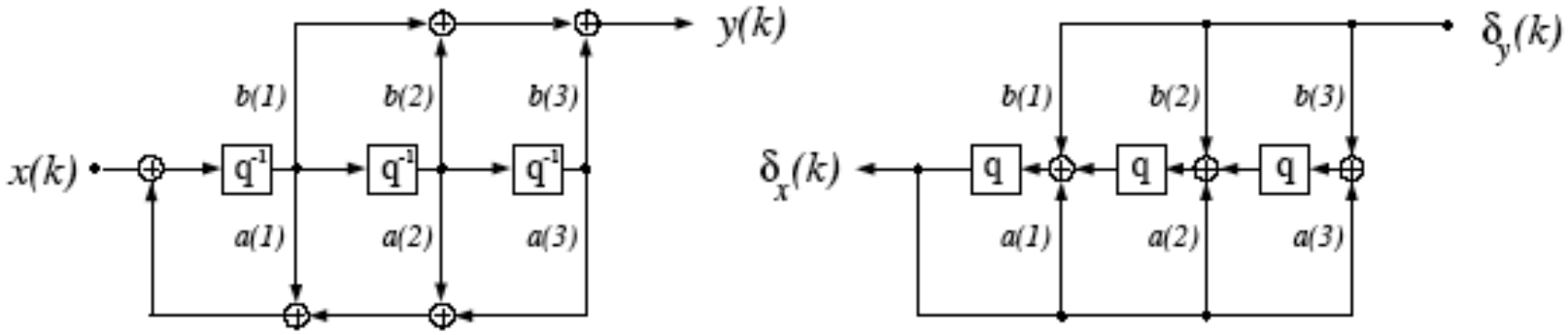
Weights in a standard MLP are replaced with FIR (Finite Impulse Response) filters.

Reciprocal Network



Example: IIR Network

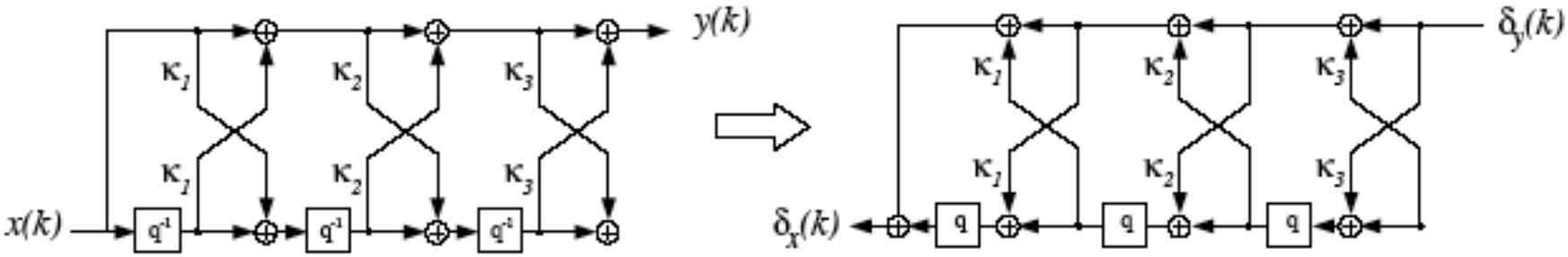
As in FIR case, but weights are replaced with IIR (Infinite Impulse Response) filters.



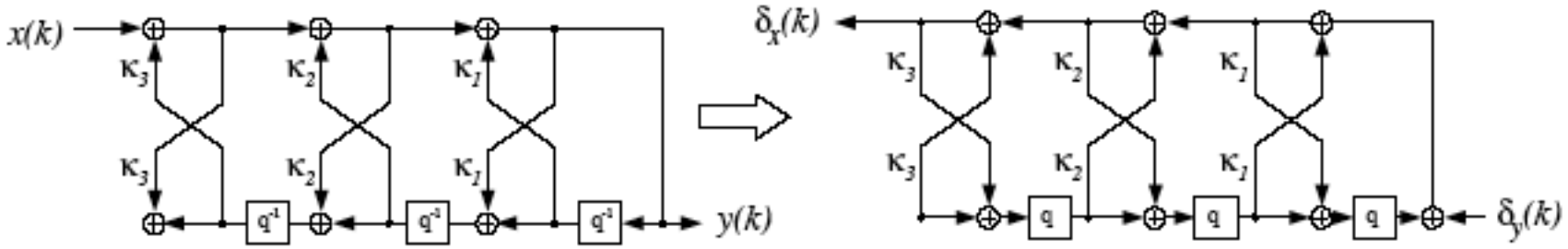
given: $$y(k) = \sum_{m=1}^{M} a(m)y(k-m) + \sum_{m=0}^{M} b(m)x(k-m) = \frac{\sum_{m=0}^{M} b(m)q^{-m}}{1 - \sum_{m=1}^{M} a(m)q^{-m}} x(k)$$

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

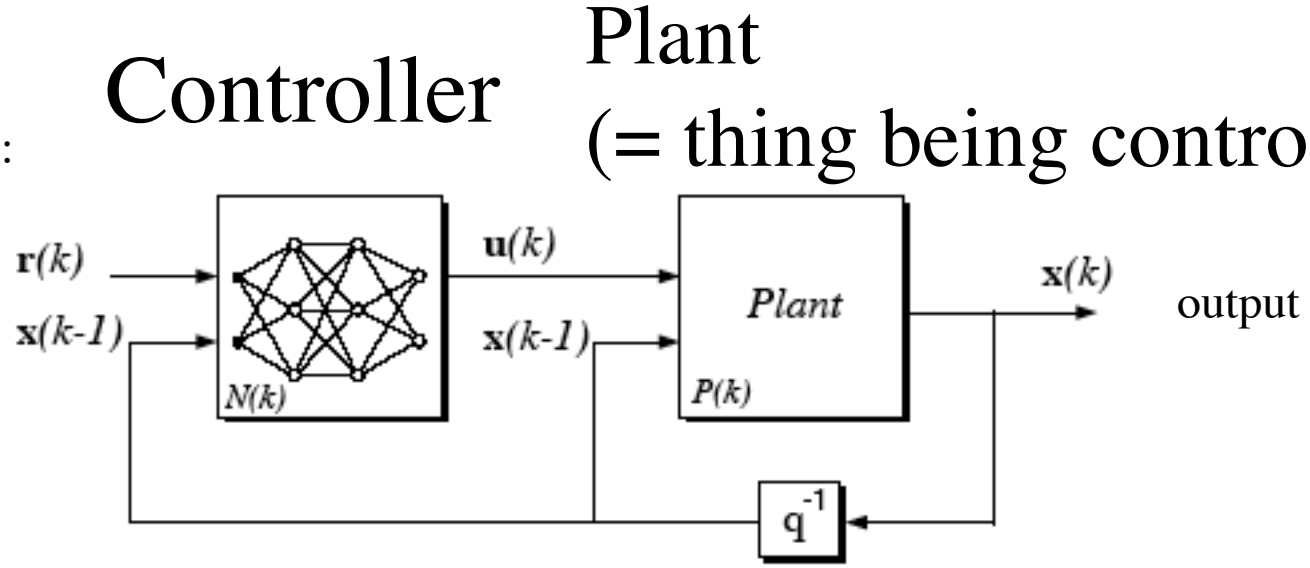


Other Options: Lattice IIR



Application: Neural Controllers

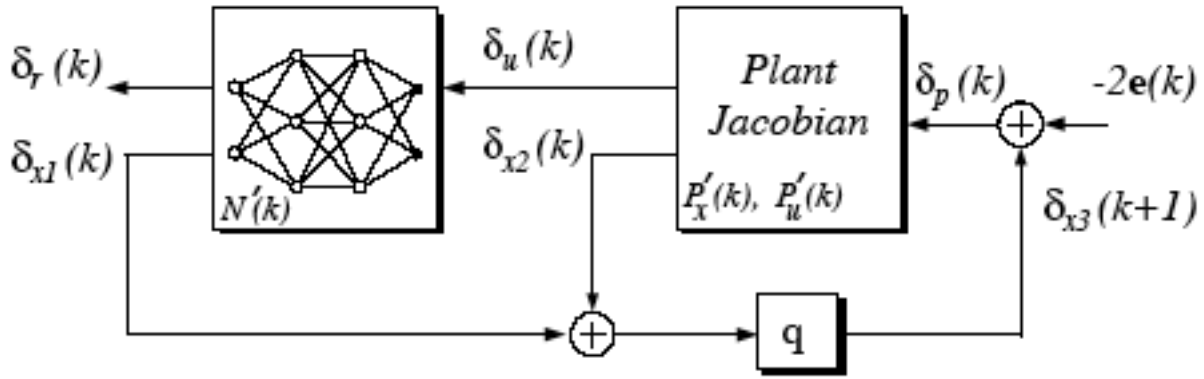
reference signal:
controls how
plant is to behave



The approach is again like BPTT.

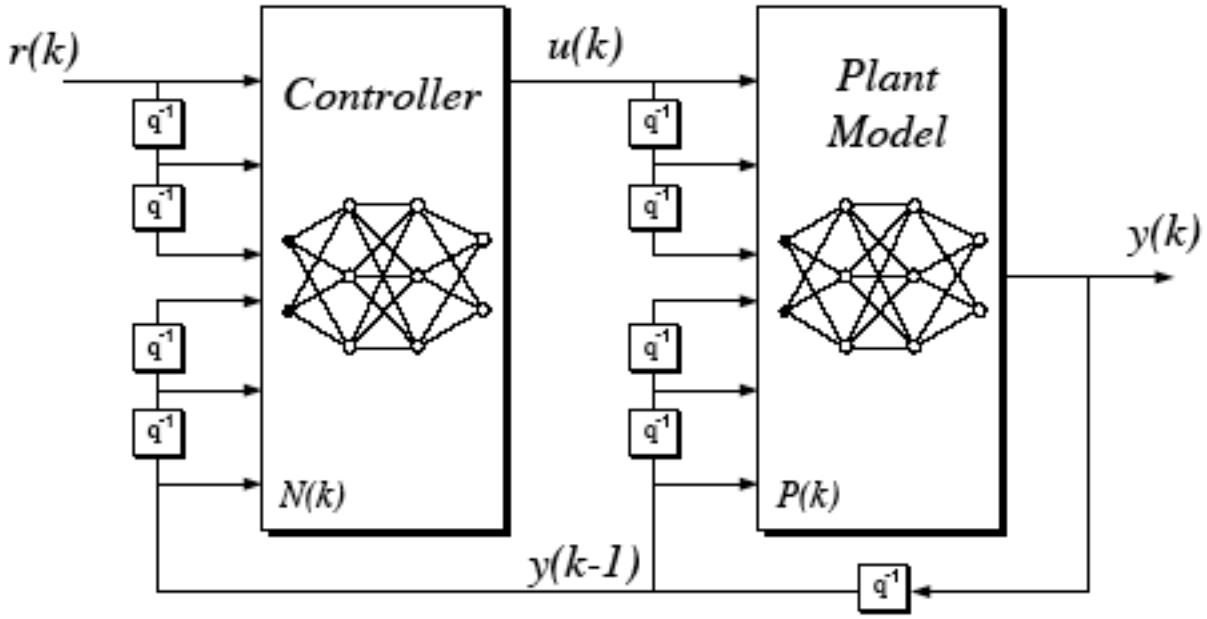
Training the Controller

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

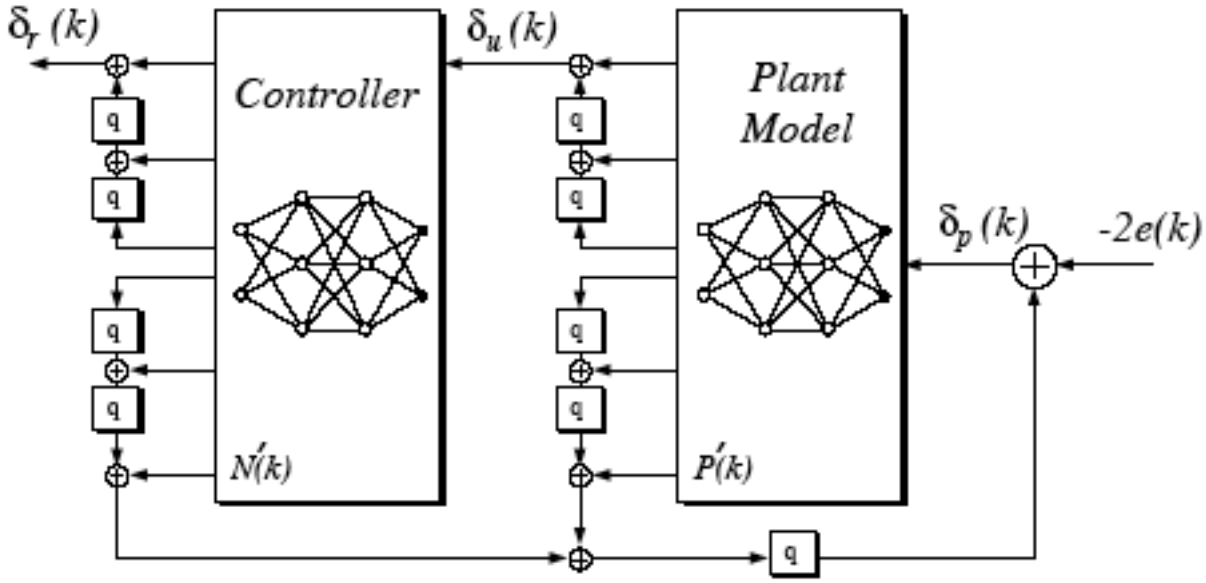


A *neural model* could be substituted for the plant, as in the truck-backer problem.

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

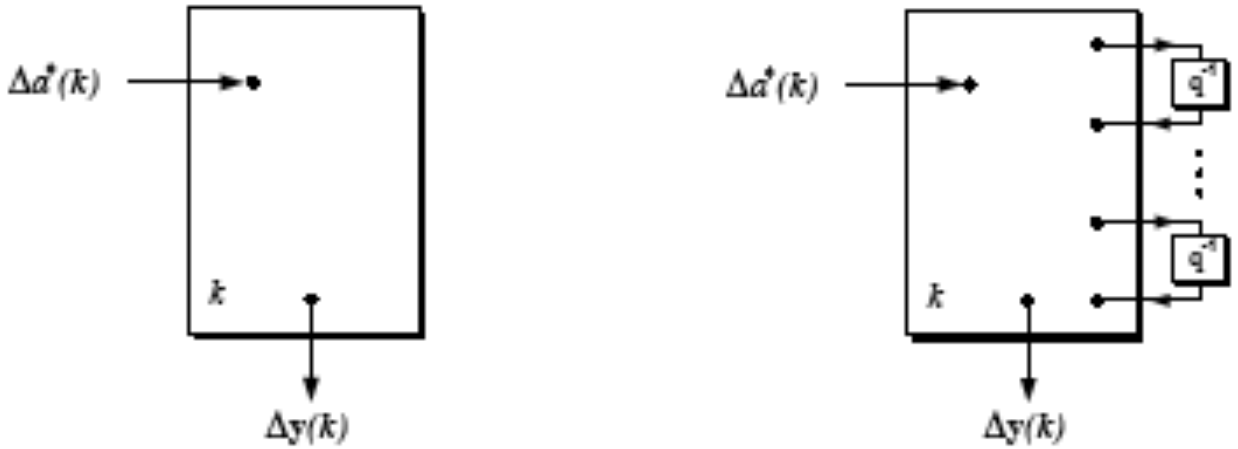
- The signals in the network are inter-related by sets of **equations** (e.g. at a summing junction, at a branch point, at a delay, etc.).

- For each such equation, we can propagate the perturbation Δ. For example, through a delay with a_i as input and a_j as output, we will have

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

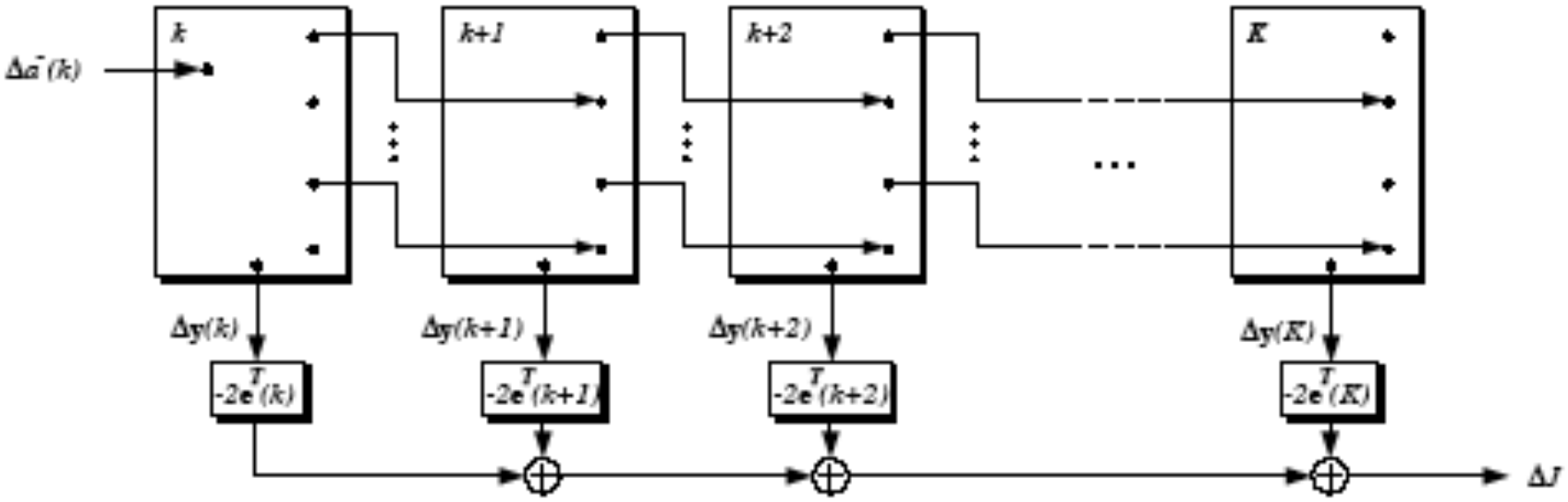
Proof Idea

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



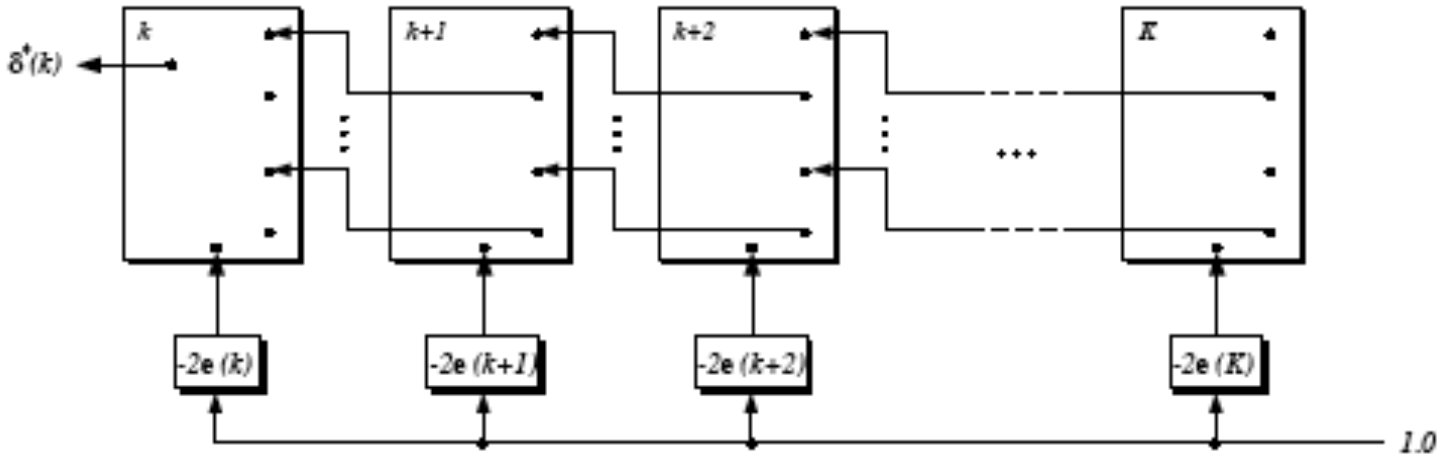
Proof Idea

- 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.



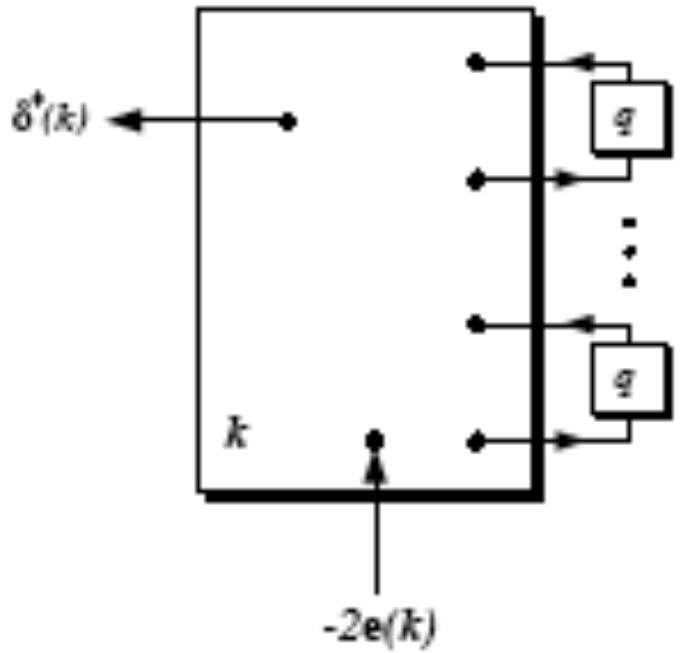
Proof Idea

- Then derive $\delta^*(k)$ at the input to the unfolded network, using the obvious reciprocal relationships.



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 ‘06 did such a project in Spring 2005 (using Python).