Paper Review:

Eric Wan and Françoise Beaufays

Diagrammatic Derivation of Gradient Algorithms for Neural Networks

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.

Signal Values

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

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

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

- $L_k(d(k), y(k)) = e(k)e(k)^T$, 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 and y(k) the actual output.

Using Derivatives for Weight Changes

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.

(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.

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

- 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

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

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

Sensitivity of the Output Signal

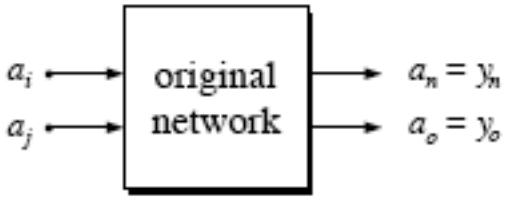
- Sensitivity of the output signals are, by definition,

  $-2e(k)$

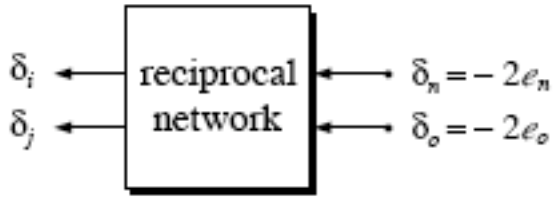
  $=$ derivative of $(d(k) - y(k))^2$
  where $e(k) = d(k) - y(k)$.

The General Derivation Setup

Given



Derive

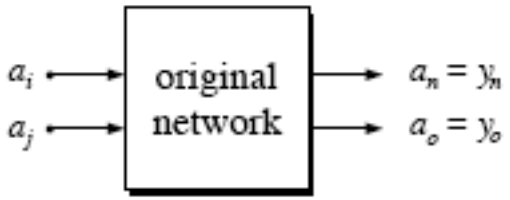


for purposes of computing δ's.

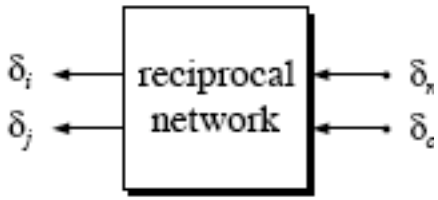
This network is called the "**reciprocal**" of the original.

A form of induction is implicit:

Given



Derive

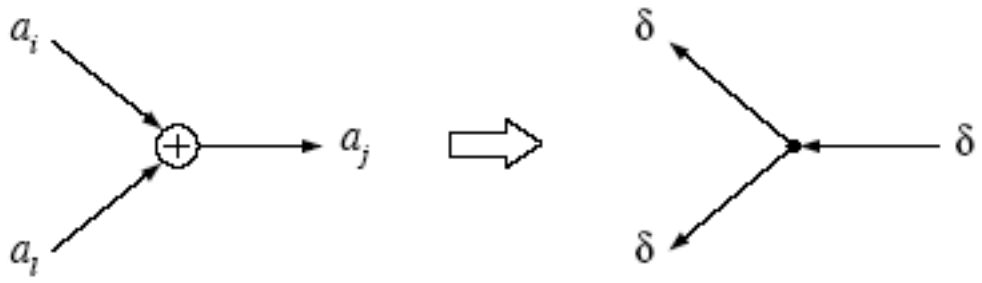


These δ's do not have to be at the overall output values.

Network Building Blocks

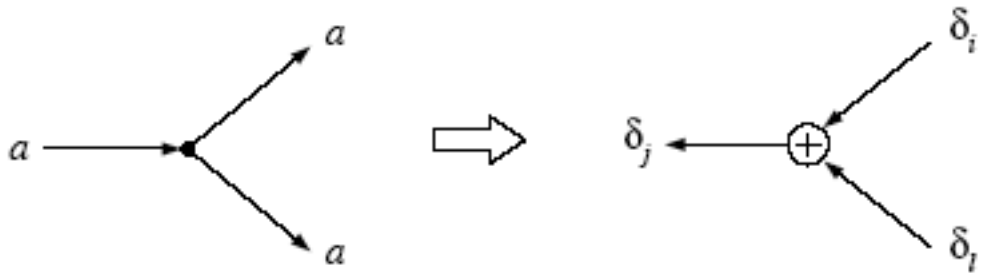
- summing junctions
- branch points
- unit time delays (only discrete-time systems are considered)
- functions (univariate and multivariate)

Summing Junction Rule



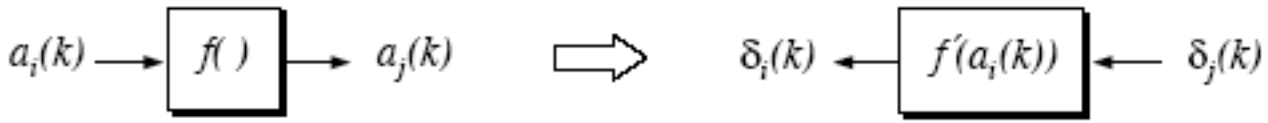
The reciprocal of a summing junction is a branch-point.

Branch-Point Rule



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

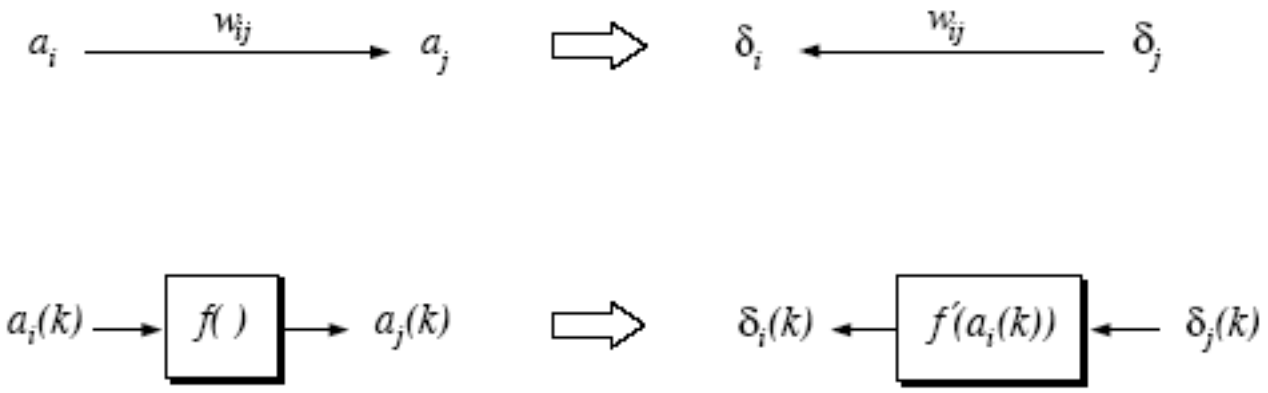
Univariate Function Rule



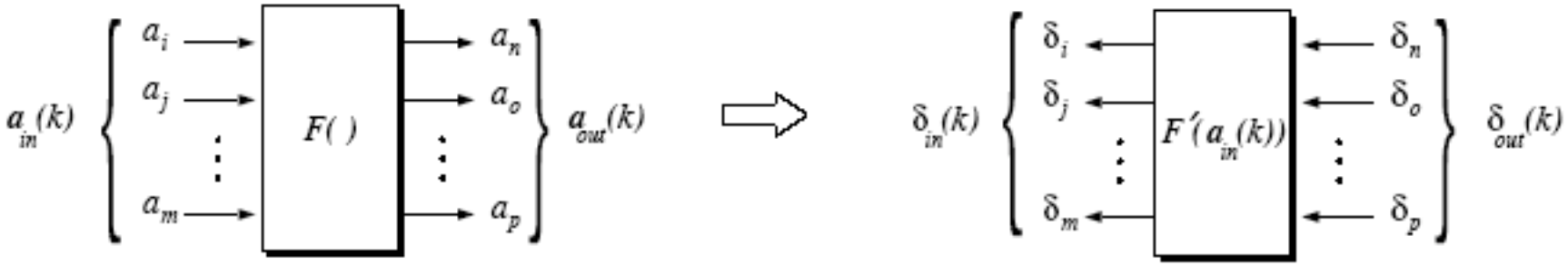
The reciprocal of a univarate function is the function's derivative, evaluated at the original input.

Note: The argument k is a time-step index.

Weights are a special case of function (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.]

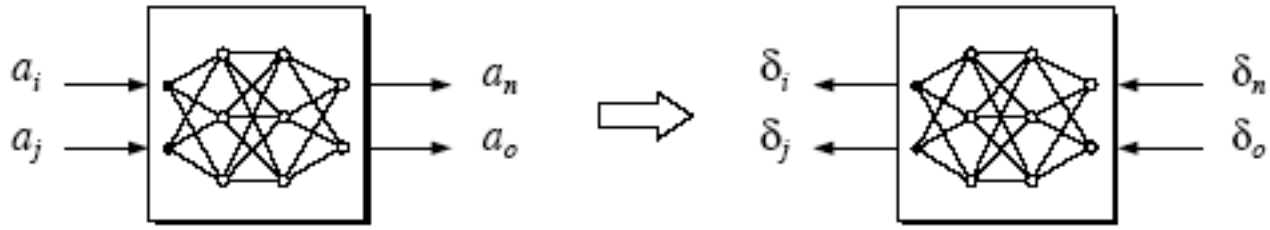
Note that summing junctions, branch points, and univariate functions are all special cases of multivariate functions.

Unit-Delay Rule

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

q^{+1} is the "time-advance" operator.
So the *future* sensitivity is back-propagated.

Sub-Networks, Layers



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

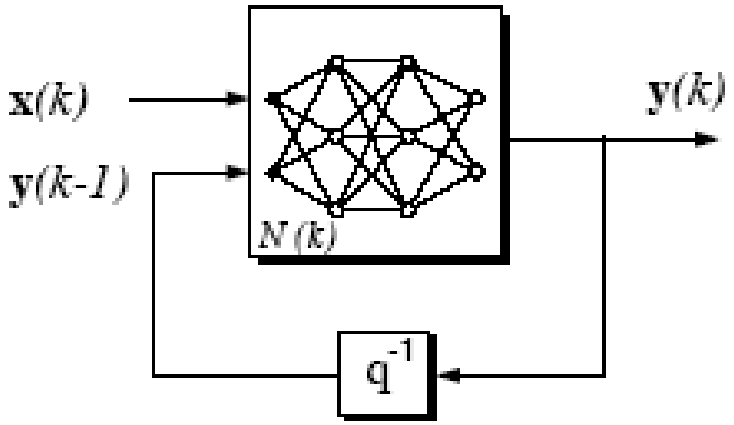
Example: Standard Backpropagation

The graphic derivation gives rise to the rules:

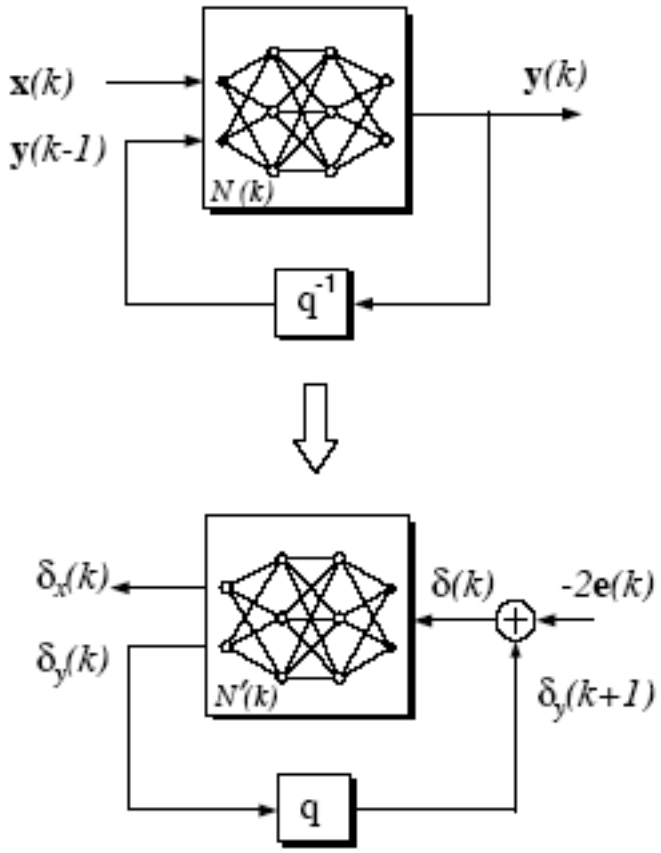
$$\delta_i^l = \begin{cases} -2e_i\, f'(s_i^L) & l = L \\ f'(s_i^l) \cdot \sum_j \delta_j^{l+1} \cdot w_{ij}^{l+1} & 0 \le l \le L-1 \end{cases}$$

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

Example: Backpropagation Through Time



Construct the Reciprocal Network

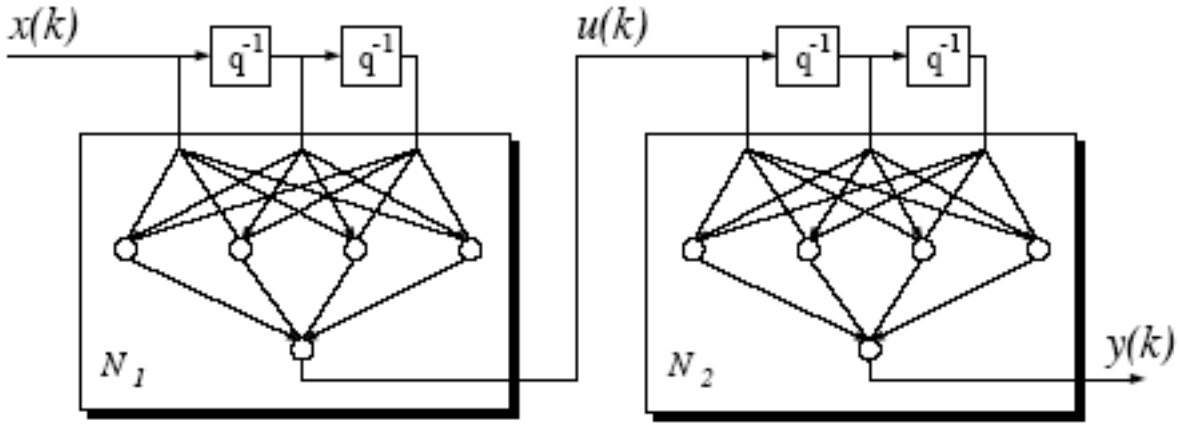


Derived recurrence formula for the *loop value* δ(k):

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

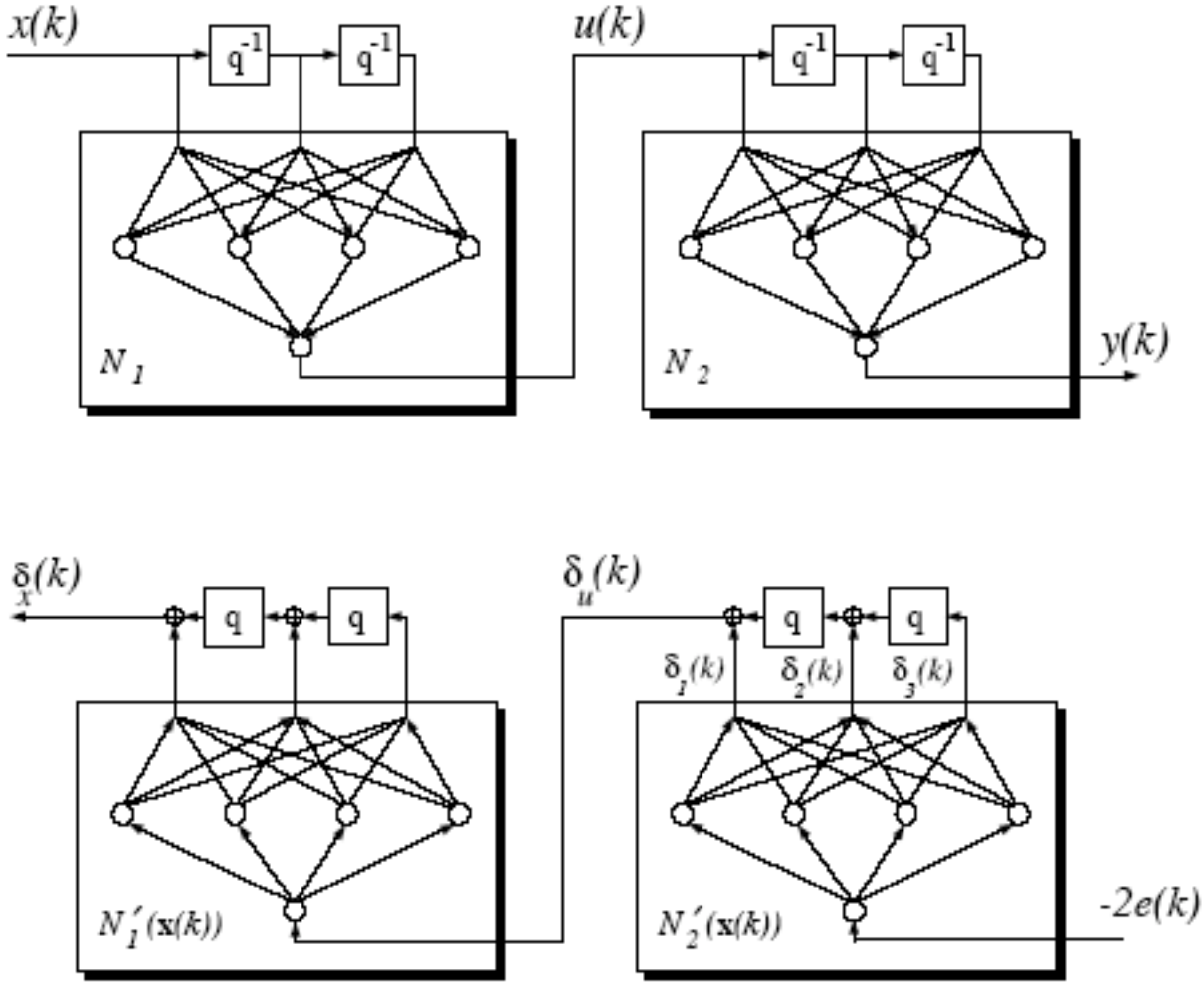
Because the above formula "looks ahead" in time, it can only be used when the number of steps is known in advance.

Example: Cascaded Networks with Delay Lines

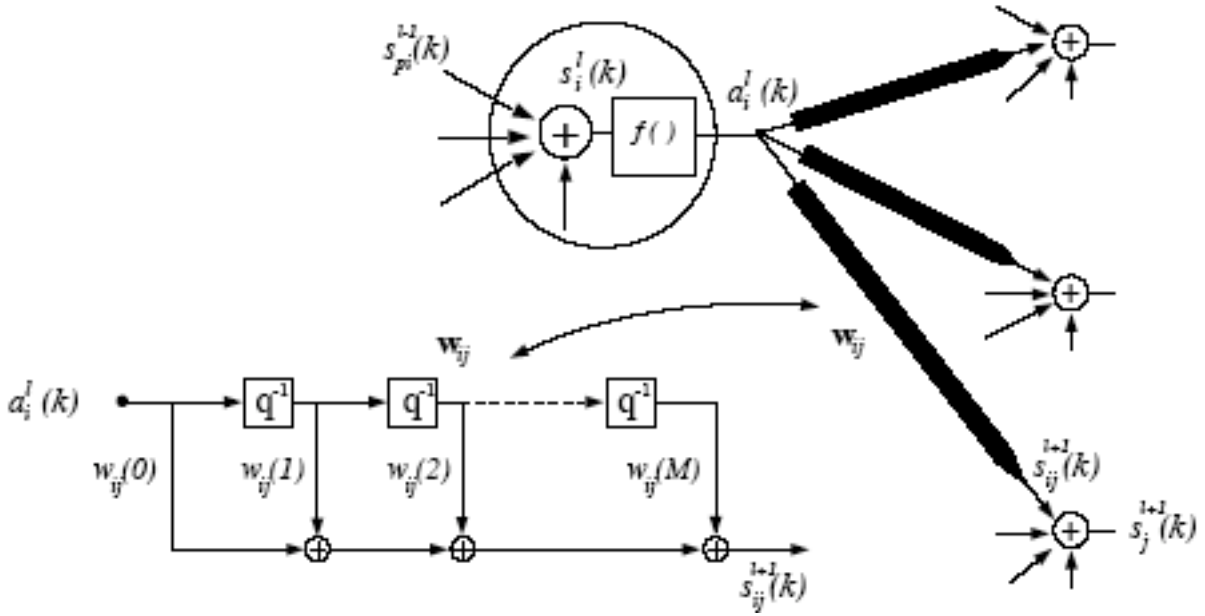


The δ corresponding to u is used as the output error for N_1 in backpropagating.

Corresponding Reciprocal Network

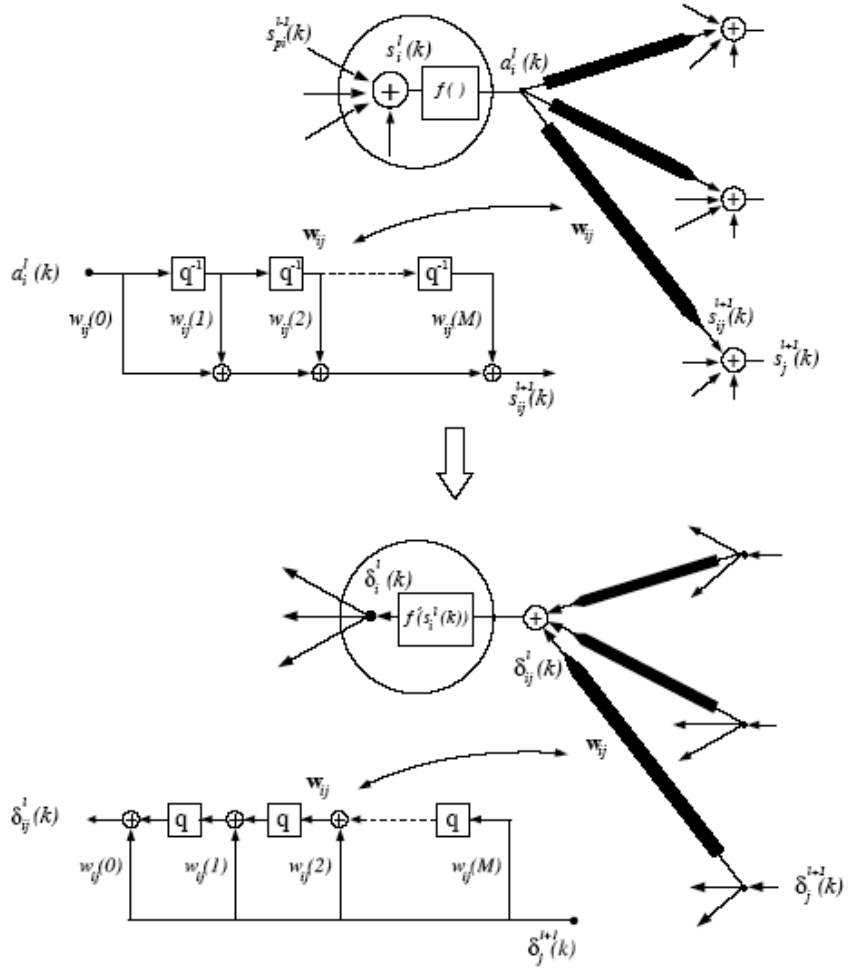


Example: FIR Network (Wan, 1993)



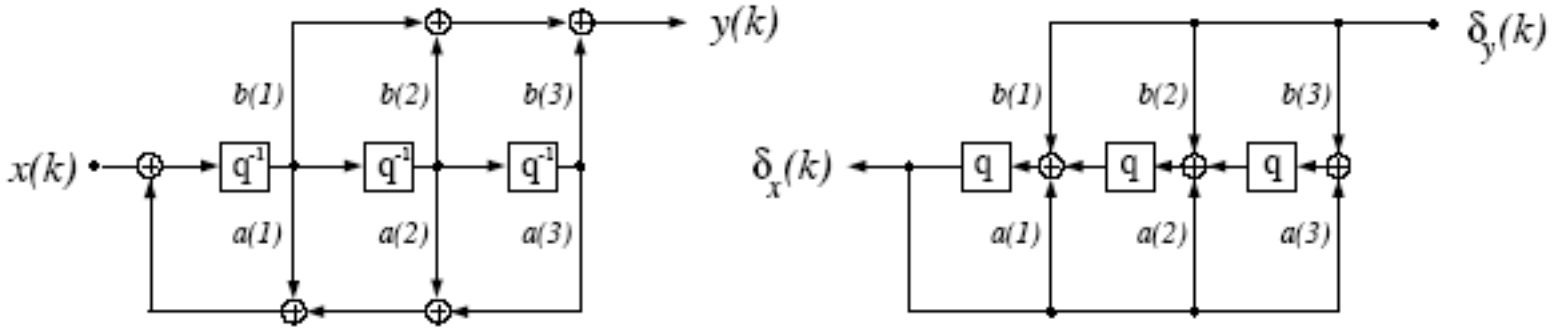
Weights in a standard MLP are replaced with FIR (Finite Impulse Response) filters.

Reciprocal Network



Example: IIR Network

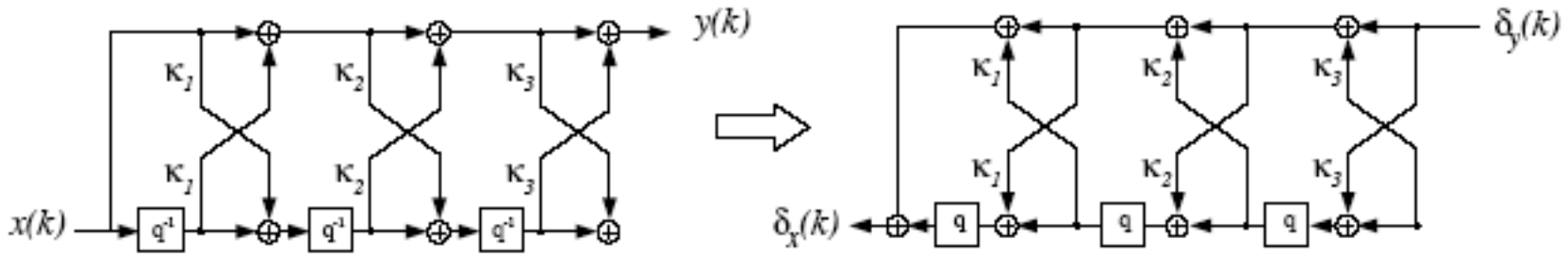
As in FIR case, but weights are replaced with IIR (Infinite Impulse Response) filters.



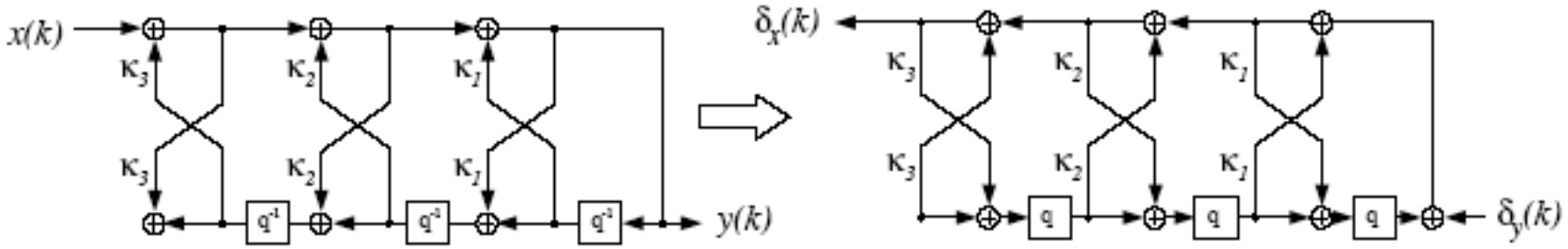
given: $$y(k) = \sum_{m=1}^{M} a(m)y(k-m) + \sum_{m=0}^{M} b(m)x(k-m) = \frac{\sum_{m=0}^{M} b(m)q^{-m}}{1 - \sum_{m=1}^{M} a(m)q^{-m}} x(k)$$

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

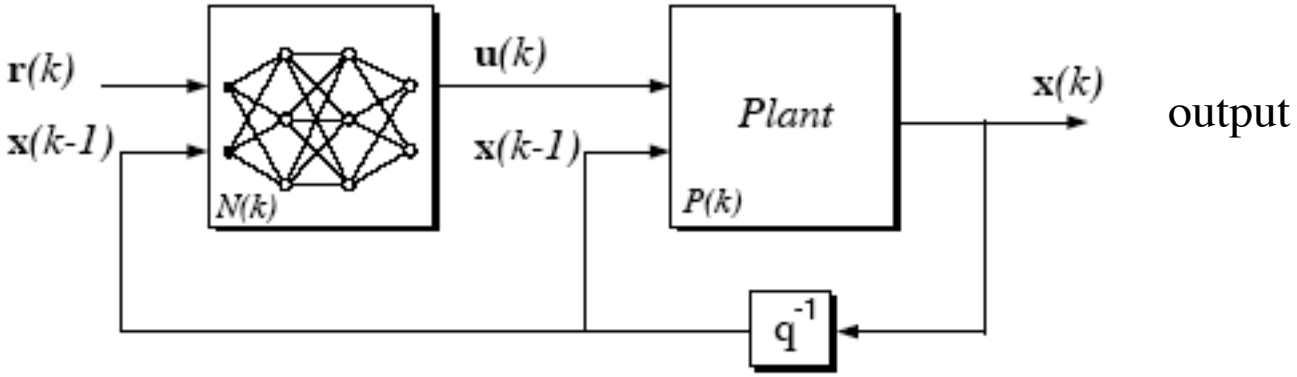


Application: Neural Controllers

Controller

Plant =
Thing being controlled

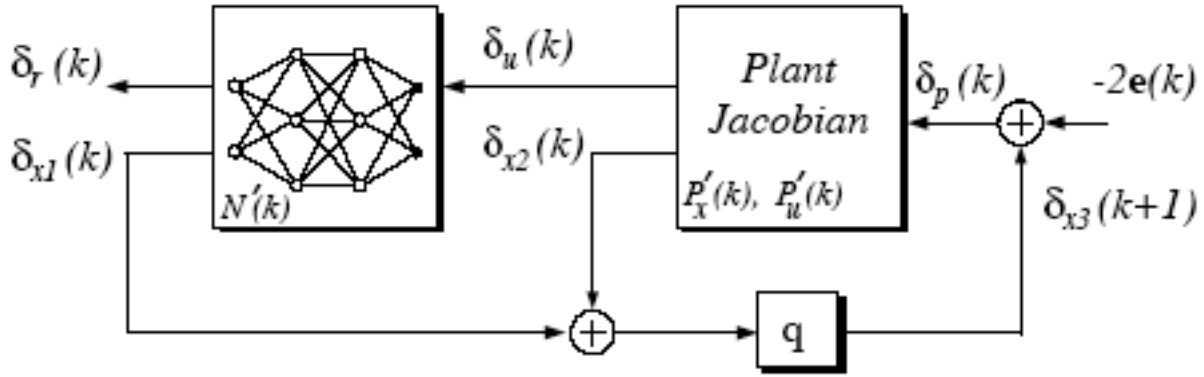
reference signal:
determines how
plant is to behave



The approach is again like BPTT.

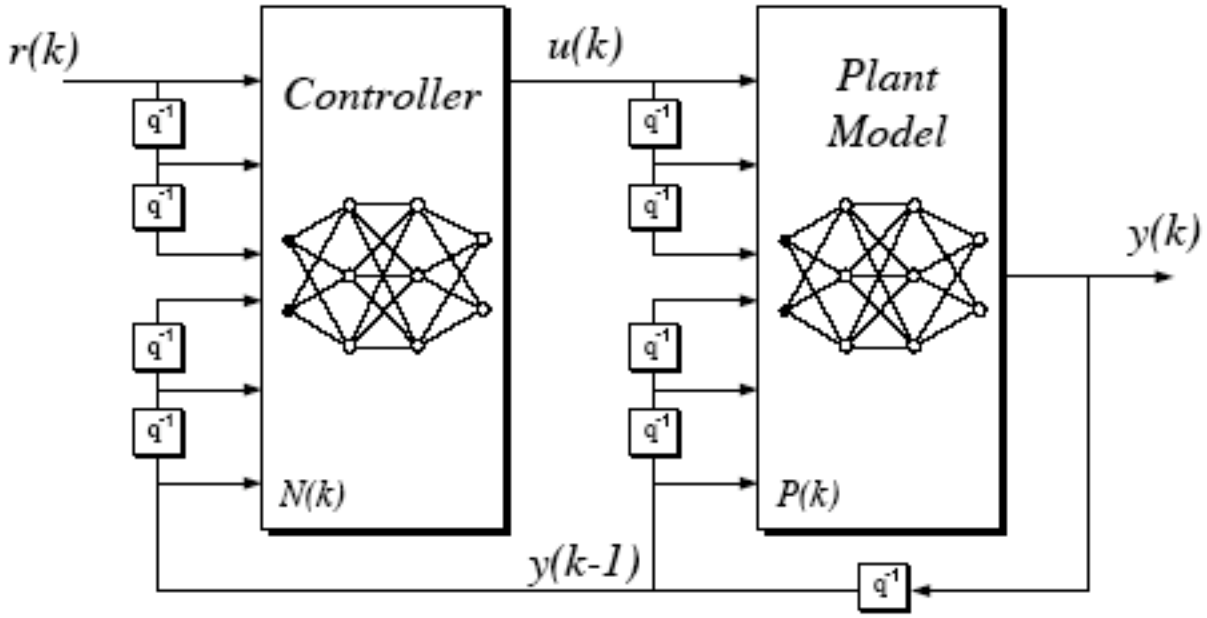
Training the Controller

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

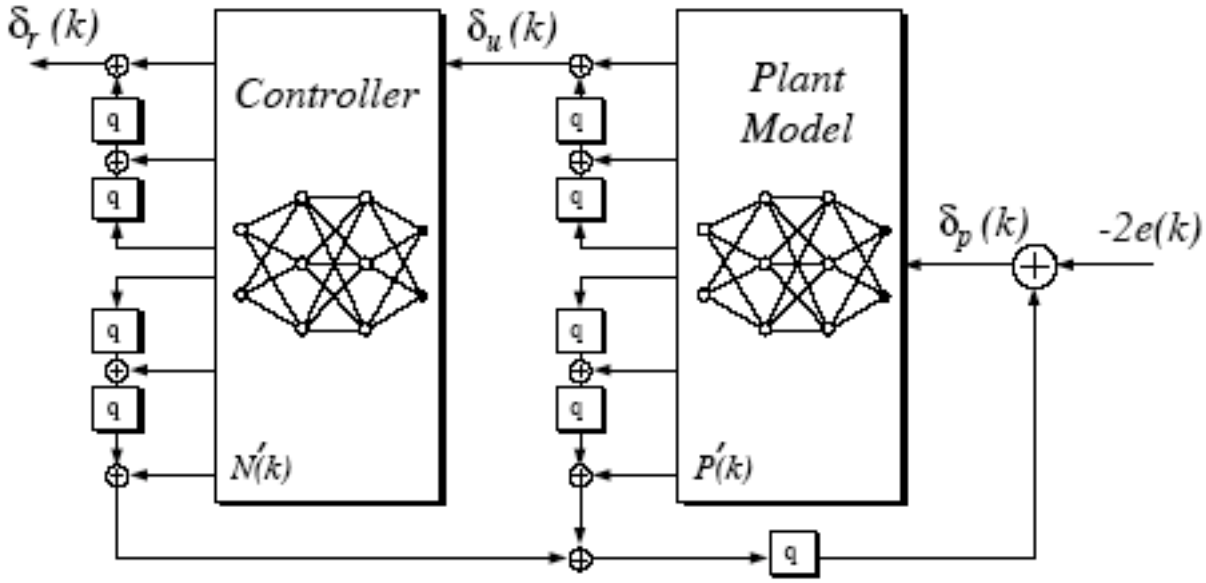


A neural model could be substituted for the Plant, as in the truck-backer problem.

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.

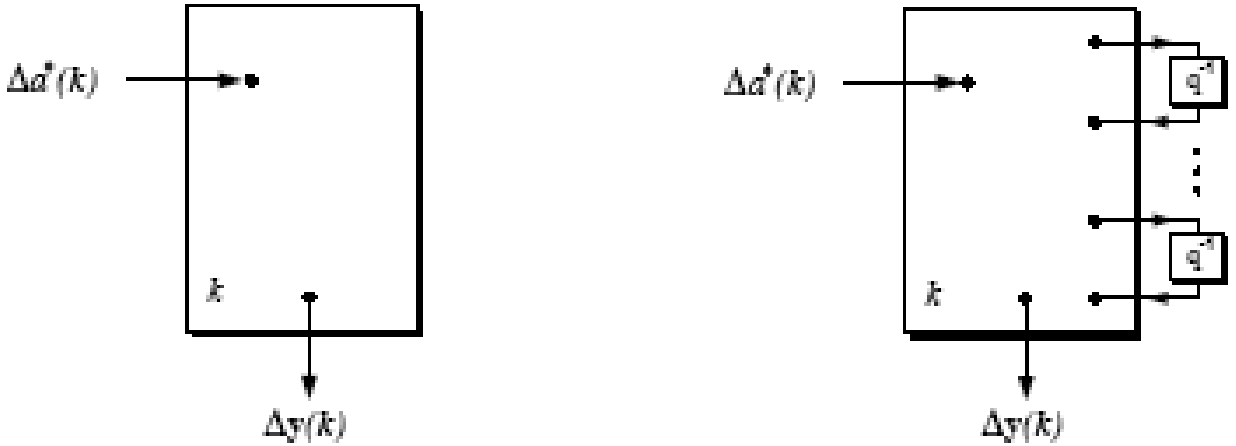
Proof Idea

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

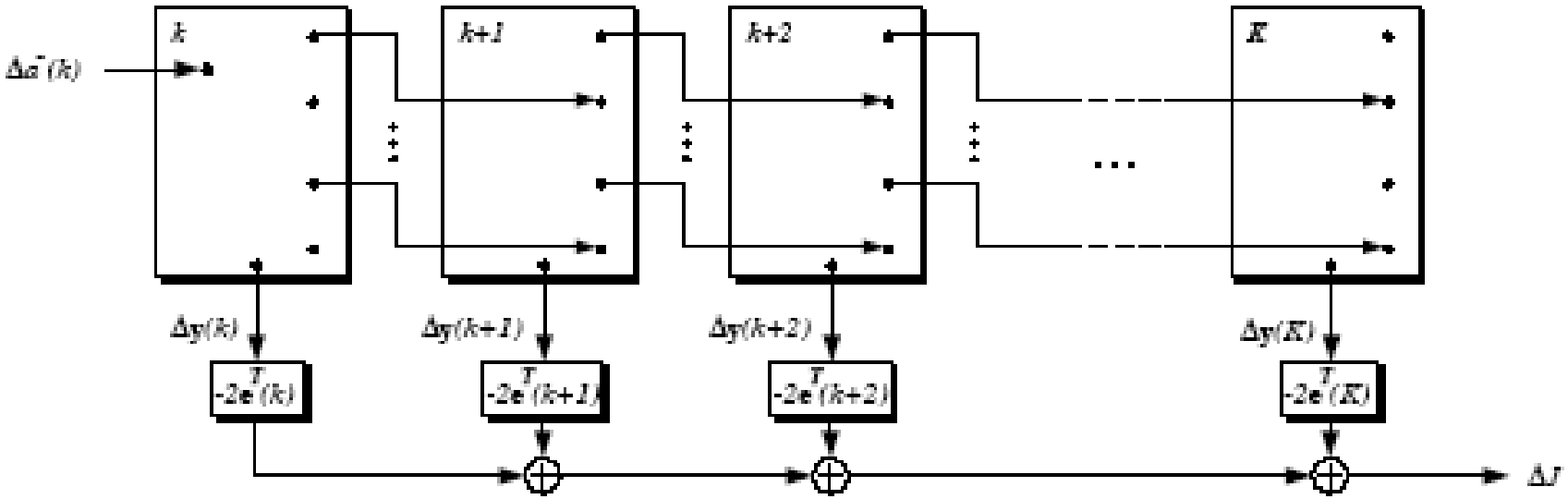
Proof Idea

- We drag all delays outside of the network.



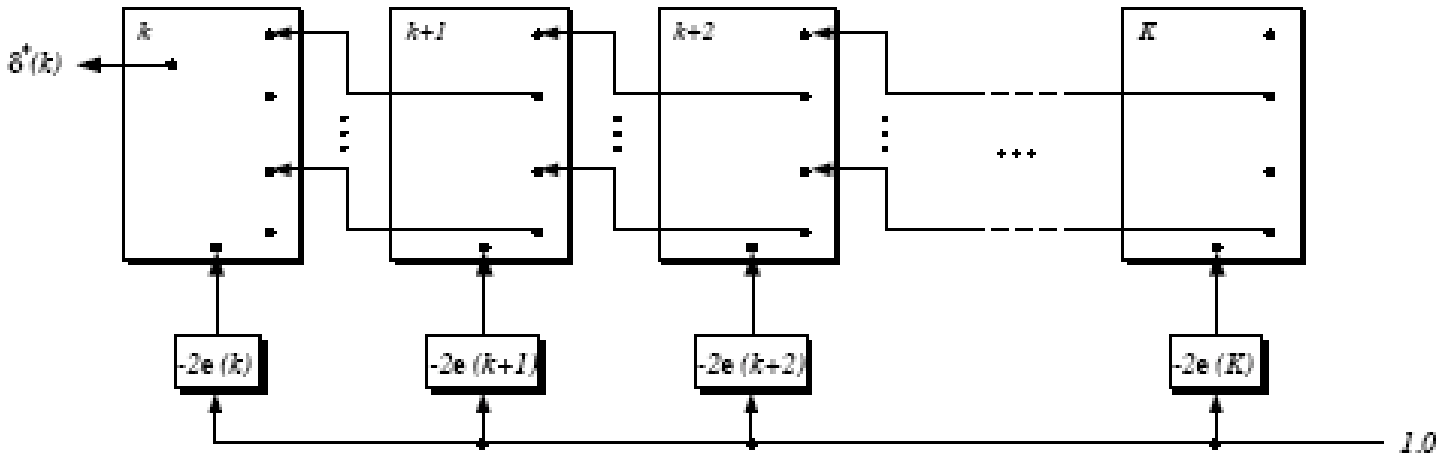
Proof Idea

- We then **unfold** the resulting network, to get rid of delays entirely, using the error values at different time steps:



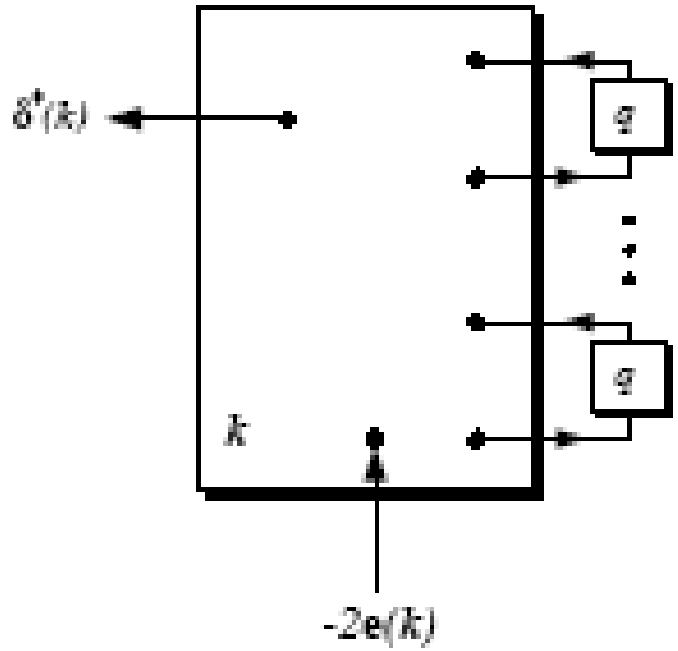
Proof Idea

- We then derive $\delta^*(k)$ at the input to the unfolded network, using the obvious reciprocal relationships. (The authors reference the network reciprocity theorem of Tellegen, 1952.)



Proof Idea

- We then **re-fold** to get back to the original network.



Concerns

- If the network has cycles, it is unclear that unfolding terminates.

- We assume that the number of iterations 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

- Not necessarily for this term, but:
- Develop a *modular* software implementation for network learning based on this method.