Diagrammatic Methods for Deriving and Relating Temporal Neural Network Algorithms

Eric A. Wan[1] and Françoise Beaufays[2]

[1] Department of Electrical Engineering, Oregon Graduate Institute,
P.O. Box 91000, Portland, OR 97291
[2] Speech Technology and Research Laboratory, SRI International,
Menlo Park, CA 94025

Abstract. Deriving gradient algorithms for time-dependent neural network structures typically requires numerous chain rule expansions, diligent bookkeeping, and careful manipulation of terms. While principled methods using Euler-Lagrange or ordered derivative approaches exist, we present an alternative approach based on a set of simple block diagram manipulation rules. The approach provides a common framework to derive popular algorithms, including backpropagation and backpropagation-through-time, without a single chain rule expansion. We further illustrate a complementary approach using flow graph interreciprocity to show transformations between on-line and batch learning algorithms. This provides simple intuitive relationships between such algorithms as real-time recurrent learning, dynamic backpropagation, and backpropagation-through-time. Additional examples are provided for a variety of architectures to illustrate both the generality and the simplicity of the diagrammatic approaches.

1 Introduction

Deriving the appropriate gradient descent algorithm for a new network architecture or system configuration normally involves brute force derivative calculations. For example, the celebrated backpropagation algorithm for training feedforward neural networks was derived by repeatedly applying chain rule expansions backward through the network (Rumelhart *et al.*, 1986; Werbos, 1974; Parker, 1982). However, the actual implementation of backpropagation may be viewed as a simple reversal of signal flow through the network. Another popular algorithm, backpropagation-through-time for recurrent networks, can be derived by Euler-Lagrange or ordered derivative methods, and involves both a signal flow reversal and *time* reversal (Werbos, 1992; Nguyen and Widrow, 1989). For both these algorithms, there is a *reciprocal* nature to the forward propagation of states and the backward propagation of gradient terms. These properties are often attributed to the clever manner in which the algorithms were derived for a specific network architecture. We will show, however, that these properties are *universal* to all network architectures and that the associated gradient algorithm may be formulated directly with virtually no effort.

The approach presented in this chapter makes use of *flow graph theory* to construct and manipulate block diagrams representing neural networks. Flow graph theory finds its origins in the field of electrical circuits, in the 1950s. Of particular importance to the field is the theorem of Tellegen which, by relating topologically identical networks, allowed many properties of analog electrical circuits to be demonstrated (Tellegen, 1952; Bordewijk, 1956). In the 1970s, the growing interest for digital filters led to the generalization of Tellegen's theorem and its corollaries to discrete-time systems, and to signals that do not necessarily obey Kirchhoff's laws of electricity (Fettweiss, 1972; Crochiere and Oppenheim, 1975). In particular, a corollary often referred to as the *principle of interreciprocity* (Penfield *et al.*, 1970), found applications in a wide variety of engineering disciplines, such as the reciprocity of emitting and receiving antennas in electromagnetism (Ramo *et al.*, 1984), the relationship between controller and observer canonical forms in control theory (Kailath, 1980), and the duality between decimation in time and decimation in frequency formulations of the FFT algorithm in signal processing (Oppenheim and Schafer, 1989). In this work, we use the framework of flow graph theory and apply the principle of interreciprocity to derive and relate various algorithms for adapting nonlinear time-dependent neural networks with arbitrary architectures.

The approach consists of a simple diagrammatic method for representing the network and constructing an *adjoint* network that directly specifies the gradient derivation formula. This is in contrast to graphical methods that simply *illustrate* the relationship between signal and gradient flow *after* derivation of the algorithm by an alternative method (Narendra and Parthasarathy, 1990; Nerrand *et al.*, 1993). The adjoint networks are in principle identical to adjoint systems seen in N-stage optimal control problems (Bryson and Ho, 1975). While adjoint methods have been applied to neural networks, such approaches have been restricted to specific architectures where the adjoint systems resulted from a disciplined Euler-Lagrange optimization technique (Parisini and Zoppoli, 1994; Toomarian and Barhen, 1992; Matsuoka, 1991). Bottou and Gallinari (1991) present an approach for memoryless feedforward topolgies in which gradient rules are specified for different *modules* or components in the system. Each module and associated gradient rule can then be represented graphically. Here we use a graphical approach for the complete derivation of the algorithms. In addition, we also also consider structures with memory to allow for internal dynamics and feedback.[3]

The application of the diagrammatic approach to recurrent structures involving optimization over time results in off-line or batch algorithms. Backpropagation-through-time, for example, requires a forward sweep of the system followed by a backward sweep through a transposed structure to accumulate the gradients. On the other hand, on-line optimization may be achieved through algorithms referred to as real-time recurrent learning, on-line backpropagation, or dy-

[3] The method presented here is also similar in spirit to *Automatic Differentiation* (Rall, 1981; Griewank and Corliss, 1991). Automatic Differentiation is a simple method for finding derivatives of functions and algorithms that can be represented by acyclic graphs.

namic backpropagation (Williams and Zipser, 1989; Narendra and Parthasarathy, 1990; Hertz *et al.*, 1991). In the second part of this chapter, we show how the principle of flow graph interreciprocity may be used to convert the adjoint-based off-line algorithm to an on-line algorithm. The technique involves creating a second flow graph obtained by transposing the original one, i.e., by reversing the arrows that link the graph nodes, and by interchanging the source and sink nodes. Flow graph theory shows that transposed flow graphs are interreciprocal, and for single input single output (SISO) systems, have identical transfer functions. This provides the exact relationship between backpropagation-through-time and real-time recurrent learning. More generally, the approach provides a mechanism for deriving either on-line or off-line algorithms completely based on simple block-diagram rules.

The concepts in this chapter represent a collation of ideas presented by the authors in prior papers (Beaufays and Wan, 1994a; Wan and Beaufays, 1996).

2 Block Diagram Representation

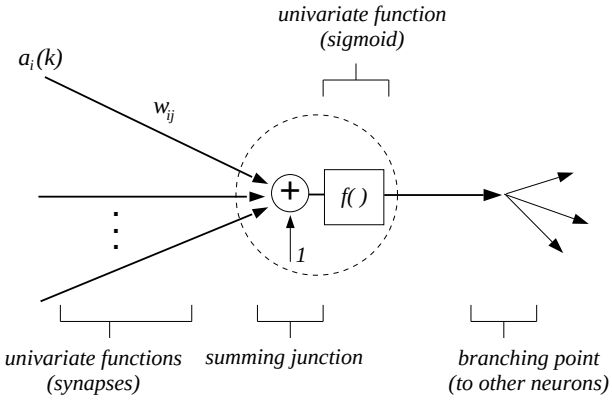


Fig. 1. Block diagram representation of a basic neuron.

An arbitrary neural network can be represented as a block diagram whose building blocks are summing junctions, branching points, univariate functions, multivariate functions, and time-delay operators. For simplicity, only discrete-time systems will be considered. A synaptic weight, for example, may be thought of as a linear transmittance, which is a special case of a univariate function. The basic neuron is simply a sum of linear transmittances followed by a univariate sigmoid function as illustrated in Figure 1. Networks can then be constructed from individual neurons, and may include additional functional blocks and time-delay operators that allow for buffering of signals and internal memory. This block diagram representation is really nothing more than the typical pictorial description of a neural network with a bit of added formalism. However, we will

show shortly that this view of a neural network allows for several important properties to be exploited.

3 Learning Algorithms

In deriving algorithms for supervised learning, the goal is to find a set of network weights W that minimize a given cost function

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

where k is the discrete time index, $\mathbf{y}(k)$ is the output of the network, $\mathbf{d}(k)$ is the desired response, and J_k is a generic error metric that may contain additional weight regularization terms. For illustrative purposes, we will work with the squared error metric, $J_k = \frac{1}{2}\mathbf{e}(k)^T\mathbf{e}(k)$, where $\mathbf{e}(k)$ is the error vector. In most problems, a desired output is specified at each time step. However, the desired output may also be defined only at the final times $k = K$ (*e.g.*, terminal control), or at intermittent time increments. Therefore, we define the error vector $\mathbf{e}(k)$ as the difference between the desired and the actual output vectors when a desired output is available, and as zero otherwise.

Optimization of the weights involves calculation of the gradient of the cost function with respect to the weights. There are two fundamental ways in which the gradient may be evaluated. In the first case, 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. The partial derivative[4] effectively evaluates the change in the total cost function into the future given a change in the weights at the current time step. Alternatively, gradient descent may by specified as

$$\Delta \tilde{W}(k) = -\mu \frac{\partial J_k}{\partial W}, \tag{3}$$

where the partial looks at the change in the current cost function at time k due to changing the weights at all prior time steps. We use the "tilde" notation, $\tilde{W}(k)$, to emphasize that the updates are not equivalent.

In the first case, evaluation of the gradient leads to an off-line implementation (i.e., the system must be run to the final time step K before a gradient update is performed), and is equivalent to an Euler-Lagrange approach. The second evaluation leads to an on-line algorithm in which the weights may be updated at each time increment, and falls into the class of algorithms known as recursive

[4] By $\partial J/\partial W(k)$ we mean a $N_w \times 1$ vector (N_w denotes the number of weights) whose i^{th} element, $\partial J/\partial w_i(k)$, is the derivative of the cost function with respect to the i^{th} weight at time k, all the other weights being kept constant.

estimation. If there is no internal memory in the network (i.e. a static network), then the two equations are equivalent. Note that if the weights are kept constant during the whole forward and backward passes, the total weight changes for the on-line and off-line algorithms are equal, i.e.

$$\sum_{k=1}^{K} \frac{\partial J}{\partial W(k)} = \sum_{k=1}^{K} \frac{\partial J_k}{\partial W}. \tag{4}$$

In practice however, the weights are updated at each time step k, resulting in differences in the properties of the two approaches. To begin with, we will consider only the first gradient evaluation from Equation 2. We will return to the on-line update algorithms in Section 6.

Let us consider evaluating the gradient in Equation 2 for some vector subset of the weights $\mathbf{w}_{ij} \in W$. At the architectural level, we isolate this parameter vector between two internal vector signals in the network with corresponding labels $\mathbf{a}_i(k)$ and $\mathbf{a}_j(k)$, such that

$$\mathbf{a}_j(k) = F(\mathbf{w}_{ij}, \mathbf{a}_i(k)).$$

Here $F(\cdot)$ represents an arbitrary continuous function internal to our block diagram representation of the network. At the lowest level, with scalar signals, w_{ij} may correspond to a single synaptic weight, and we have simply $a_j(k) = w_{ij}\ a_i(k)$.

Using the chain rule, we obtain

$$\frac{\partial J}{\partial \mathbf{w}_{ij}(k)} = \frac{\partial \mathbf{a}_j(k)}{\partial \mathbf{w}_{ij}(k)} \frac{\partial J}{\partial \mathbf{a}_j(k)} = G^F_{\mathbf{W}}(k)\ \boldsymbol{\delta}_j(k), \tag{5}$$

where we define the error gradient

$$\boldsymbol{\delta}_j(k) \triangleq \frac{\partial J}{\partial \mathbf{a}_j(k)}$$

and the Jacobian

$$G^F_{\mathbf{W}}(k) \triangleq \frac{\partial \mathbf{a}_j(k)}{\partial \mathbf{w}_{ij}(k)}.$$

If the vector $\mathbf{a}_j(k)$ has dimensions $N \times 1$, and the weight vector $\mathbf{w}_{ij}(k)$ has dimensions $N_w \times 1$, the dimensionalities of $\partial J/\partial \mathbf{w}_{ij}(k)$, $\partial \mathbf{a}_j(k)/\partial \mathbf{w}_{ij}(k)$, and $\partial J/\partial \mathbf{a}_j(k)$ are, respectively, $N_w \times 1$, $N_w \times N_y$, and $N_y \times 1$.

Note that the error gradient $\boldsymbol{\delta}_j(k)$ depends on the entire topology of the network, as opposed to the Jacobian which depends only locally on F. In the scalar weight case $G^F_w(k) = a_i(k)$. The weight update is given by

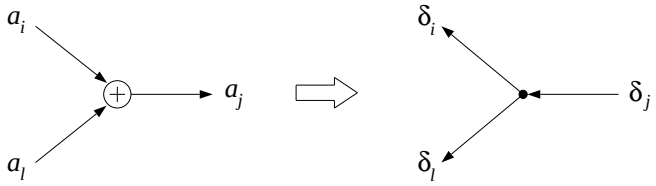
$$\Delta \mathbf{w}_{ij}(k) = -\mu \frac{\partial J}{\partial \mathbf{w}_{ij}(k)} = -\mu\, G^F_{\mathbf{W}}(k) \boldsymbol{\delta}_j(k). \tag{6}$$

To complete the algorithm specification we must find an explicit formula for calculating the delta terms. Backpropagation, for example, is nothing more than an algorithm for generating these terms in a feedforward network. In the next section, we develop a simple nonalgebraic method to derive the delta terms associated with any network architecture.

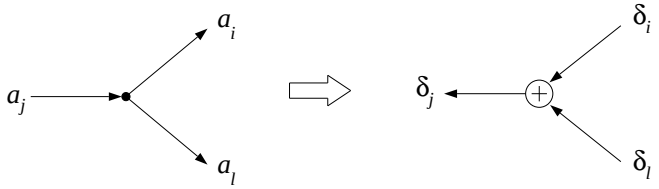
4 Adjoint Flow Graph Construction Rules

Given the block diagram representation of a network and the goal of determining the error gradients $\delta_i(k)$, we need only apply a set of simple block diagram transformations. Directly from the block diagram we may construct the *adjoint network* by reversing the flow direction in the original network, labeling all resulting signals $\delta_i(k)$, and performing the following operations:

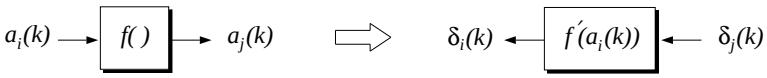
1. *Summing junctions are replaced with branching points.*



2. *Branching points are replaced with summing junctions.*

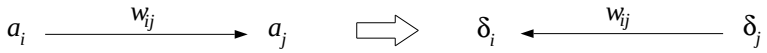


3. *Univariate functions are replaced with their derivatives.*

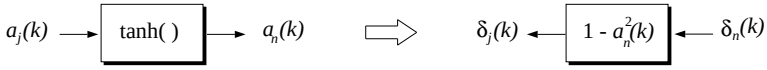


Explicitly, the scalar *continuous* function $a_j(k) = f(a_i(k))$ is replaced by $\delta_i(k) = f'(a_i(k))\ \delta_j(k)$, where $f'(a_i(k)) \triangleq \partial a_j(k)/\partial a_i(k)$. Note that this rule replaces a nonlinear function by a *linear time-dependent* transmittance. Special cases are

- Weights: $a_j = w_{ij}\ a_i$, in which case $\delta_i = w_{ij}\ \delta_j$.



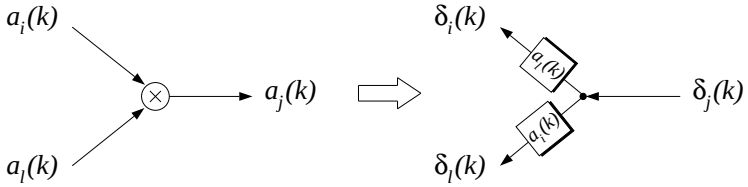
- Activation functions: $a_n(k) = \tanh(a_j(k))$. In this case, $f'(a_j(k)) = 1 - a_n^2(k)$.



4. *Multivariate functions are replaced with their Jacobians.*

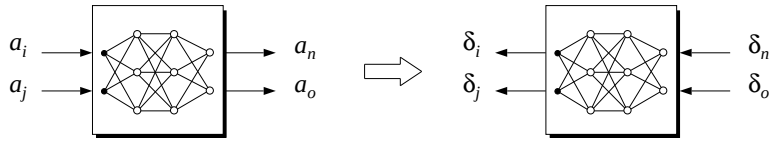
A multivariate function maps a vector of input signals into a vector of output signals, $\mathbf{a}_{out} = F(\mathbf{a}_{in})$. In the transformed network, we have $\boldsymbol{\delta}_{in}(k) = G^F_{\mathbf{a}_{in}}(k)\ \boldsymbol{\delta}_{out}(k)$, where the Jacobian $G^F_{\mathbf{a}_{in}}(k) \triangleq \partial \mathbf{a}_{out}(k)/\partial \mathbf{a}_{in}(k)$ corresponds to a matrix of partial derivatives. Clearly, both summing junctions and univariate functions are special cases of multivariate functions. Other important cases include

- Product junctions: $a_j(k) = a_i(k)\ a_l(k)$, in which case $G^F_{\mathbf{a}}(k) = [a_l(k)\ a_i(k)]^T$.



  Product terms, for example, allow construction of sigma-pi units or gated experts (Rumelhart *et al.*, 1986; Jordan and Jacobs, 1992).

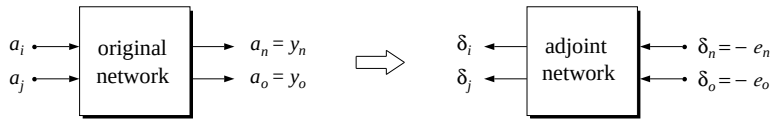
- Layered networks. A multivariate function may itself represent a multilayer network. In this case, the product $G^F_{\mathbf{a}_{in}}(k)\ \boldsymbol{\delta}_{out}(k)$ may be found directly by backpropagating $\boldsymbol{\delta}_{out}$ through the network without the explicit need to calculate the matrix $G^F_{\mathbf{a}_{in}}(k)$.



5. *Delay operators are replaced with advance operators.*

   A delay operator q^{-1} performs a unit time delay on its argument: $a_j(k) = q^{-1}a_i(k) = a_i(k-1)$. In the adjoint system, we form a unit time advance: $\delta_i(k) = q^{+1}\delta_j(k) = \delta_j(k+1)$. The resulting system is thus noncausal. Actual implementation of the adjoint network in a causal manner is addressed in specific examples. Note that a generalization of this rule is that any linear subsystem $H(q^{-1})$ in the original network is transformed to $H(q^{+1})$ in the adjoint system.

6. *Outputs become inputs.*



   By reversing the signal flow, output nodes $a_n(k) = y_n(k)$ in the original network become input nodes in the adjoint network. These inputs are then set at each time step to $e_n(k)$. (For cost functions other than squared error, the input should be set to $\partial J_k/\partial y_n(k)$.)

These six rules allow direct construction of the adjoint network from the original network.[5] Note that there is a topological equivalence between the two networks. The order of computations in the adjoint network is thus identical to the order of computations in the forward network. Whereas the original network corresponds to a nonlinear time-independent system (assuming the weights are fixed), the *adjoint network* is a linear time-dependent system. The signals $\delta_j(k)$ that propagate through the adjoint network correspond to the terms $\partial J/\partial a_j(k)$ necessary for gradient adaptation. Exact equations may then be "read out" directly from the adjoint network, completing the derivation. A formal proof of the validity and generality of this method is presented in Appendix B.

5 Examples

The power and simplicity of the diagrammatic approach can best be demonstrated through examples.

5.1 Example 1: Backpropagation

We start by re-deriving standard backpropagation. Figure 2 shows a hidden neuron feeding other neurons and an output neuron in a multilayer network. For consistency with traditional notation, we have labeled the summing junction signal s_i^l with the added superscript to denote the layer. In addition, since multilayer networks are static structures, we omit the time index k.

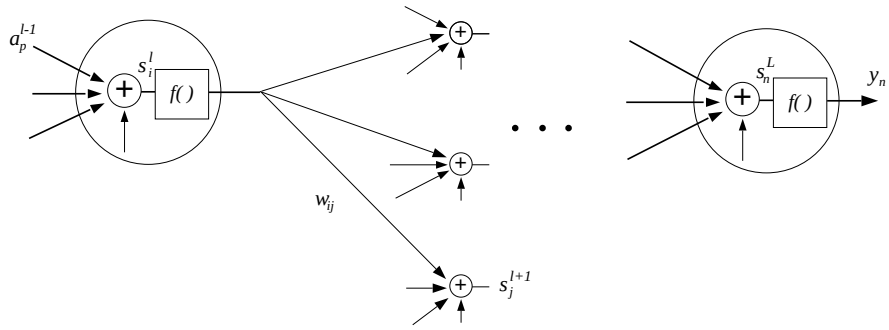


Fig. 2. Block diagram construction of a multilayer network.

[5] Clearly, this set of rules is not a minimal set, *e.g.*, a summing junction can be considered a special case of a multivariate function. However, we chose this set for ease and clarity of construction.

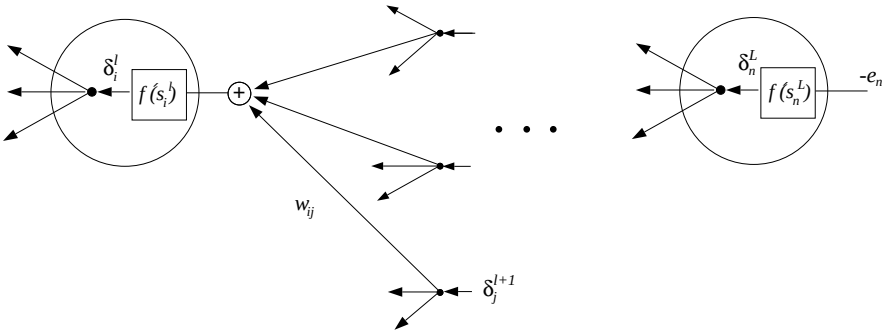


Fig. 3. Adjoint multilayer network.

The adjoint network shown in Figure 3 is found by applying the construction rules of the previous section. From this figure, we may immediately write down the equations for calculating the delta terms:

$$\delta_i^l = \begin{cases} -e_i \, f'(s_i^L) & l = L \\ f'(s_i^l) \cdot \sum_j \delta_j^{l+1} \cdot w_{ij}^{l+1} & 0 \leq l \leq L-1. \end{cases} \tag{7}$$

By Equation 2, the weight update is formulated as

$$\Delta w_{pi}^l = -\mu \, \delta_i^l \, a_p^{l-1}. \tag{8}$$

These are precisely the equations describing standard backpropagation. In this case, there are no delay operators, and $\delta_j = \delta_j(k) \triangleq \frac{\partial J}{\partial s_j(k)} = \frac{\partial \mathbf{e}^T(k)\mathbf{e}(k)}{\partial s_j(k)}$ represents an *instantaneous* gradient. Readers familiar with neural networks have undoubtedly seen these diagrams before. What we emphasize is the concept that the diagrams themselves may be used directly to derive the learning algorithm, completely circumventing all intermediate steps involving tedious algebra.

5.2 Example 2: Backpropagation-Through-Time

For the next example, consider a network with output feedback (see Figure 4) described by

$$\mathbf{y}(k) = \mathcal{N}(W, \mathbf{x}(k), \mathbf{y}(k-1)), \tag{9}$$

where $\mathbf{x}(k)$ are external inputs, and $\mathbf{y}(k)$ represents the *vector* of outputs that form feedback connections. $\mathcal{N}$ is a multilayer neural network. If $\mathcal{N}$ has only one layer of neurons, every neuron output has a feedback connection to the input of

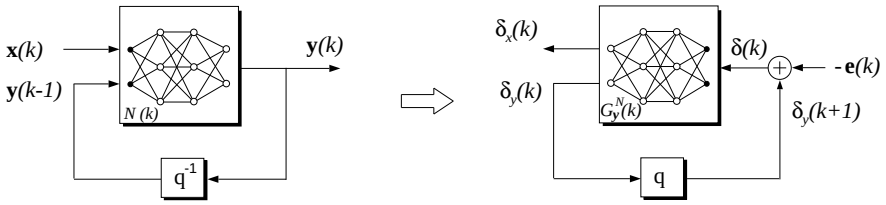


Fig. 4. Recurrent network and backpropagation-through-time.

every other neuron and the structure is referred to as a *fully recurrent network* (Williams and Zipser, 1989). Typically, only a select set of the outputs have an actual desired response. The remaining outputs have no desired response (error equals zero) and are used for internal computation.

Direct calculation of gradient terms by using chain rule expansions is complicated by the time recurrence. A weight perturbation at a specified time step affects not only the output at future time steps, but future inputs as well. However, applying the adjoint construction rules (see Figure 4), we find immediately:

$$\begin{aligned} \boldsymbol{\delta}(k) &= \boldsymbol{\delta}_y(k+1) - \mathbf{e}(k) \\ &= G_{\mathbf{y}}^{\mathcal{N}}(k+1)\boldsymbol{\delta}(k+1) - \mathbf{e}(k). \end{aligned} \tag{10}$$

By Equation 2 the weight change is given by

$$\Delta W(k) \ = -\mu \, G_W^{\mathcal{N}}(k) \cdot \boldsymbol{\delta}(k), \tag{11}$$

where $\Delta W(k)$, $G_W^{\mathcal{N}}(k)$ and $\boldsymbol{\delta}(k)$ are, respectively, of dimensionality $N_w \times 1$, $N_w \times N_y$ and $N_y \times 1$, N_y being the number of feedback states.

These are precisely the equations describing *backpropagation-through-time*, which have been derived in the past using either ordered derivatives (Werbos, 1990) or Euler-Lagrange techniques (Plumer, 1993b). The diagrammatic approach is by far the simplest and most direct method.

Note that the causality constraints require these equations to be run backward in time. This implies a forward sweep of the system to generate the output states and internal activation values, followed by a backward sweep through the adjoint network. Furthermore, it is typical to accumulate the weight changes at each backward time step followed by a single batch gradient update. Also from *rule 4* in the previous section, the product $G_{\mathbf{y}}^{\mathcal{N}}(k+1)\boldsymbol{\delta}(k+1)$ may be calculated directly by a standard backpropagation of $\boldsymbol{\delta}(k+1)$ through the network at time k. Similarly, the product for the weight update, $[G_W^{\mathcal{N}}(k)] \cdot \boldsymbol{\delta}(k)$, is interpreted as static backpropagation applied to the network with error $\boldsymbol{\delta}(k)$.

Note that it is possible to construct an on-line algorithm by *truncating* the backward sweep to a fixed number of time steps, and then repeating the process at each iteration (Williams and Peng, 1990). An extension of the flow-graph approach to truncated gradient algorithms is presented in (Campolucci *et al.*, 1997).

Backpropagation-Through-Time and Neural Control Architectures

Backpropagation-through-time can be extended to various neural control architectures (Nguyen and Widrow, 1989; Werbos, 1992). A generic nonlinear plant may be described by

$$\mathbf{x}(k) = \mathcal{P}(u(k), \mathbf{x}(k-1)), \tag{12}$$

where $\mathbf{x}(k)$ is the state vector for the system and $u(k)$ is the control signal. The goal is to drive some of the states to desired values by using a multilayer neural network to generate the control signal,

$$u(k) = \mathcal{N}(W, \mathbf{r}(k), \mathbf{x}(k-1)), \tag{13}$$

where $\mathbf{r}(k)$ constitutes some external reference signal. It is assumed that full state information is available for feedback[6]. (see Figure 5)

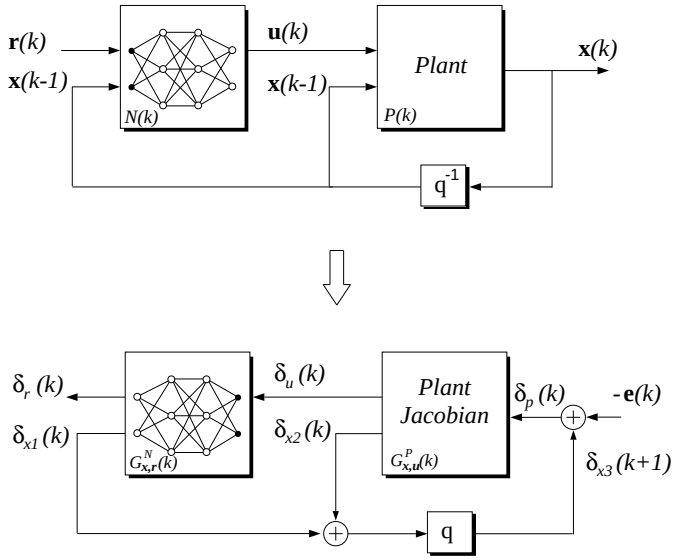


Fig. 5. Neural network control architecture and adjoint counterpart.

To adapt the weights of the controller, the gradient term $\delta_u(k) = \partial J / \partial u(k)$, constitutes an effective error for the neural network. Directly from Figure 5 we

[6] The desired response for the system may be acquired using a *model reference* (*e.g.*, a second-order linear plant), or using a linear-quadratic-regulator approach (Narendra and Parthasarathy, 1990; Landau, 1979). For terminal control problems such as robotics, the desired response may exist only at the final time step, and the necessary trajectory of controlled states must be generated by the network (Nguyen and Widrow, 1989; Plumer, 1993a; Bryson and Ho, 1975).

have

$$\delta_p(k) = G^{\mathcal{N}}_{\mathbf{x}}(k+1)\delta_u(k+1) + G^{\mathcal{P}}_{\mathbf{x}}(k+1)\delta_p(k+1) - \mathbf{e}(k) \tag{14}$$

$$\delta_u(k) = G^{\mathcal{P}}_{u}(k)\delta_p(k). \tag{15}$$

Note that $G^{\mathcal{P}}_{\mathbf{x}}(k) = \partial\mathbf{x}(k)/\partial\mathbf{x}(k-1)$ and $G^{\mathcal{P}}_{u}(k) = \partial\mathbf{x}(k)/\partial u(k)$ are Jacobian matrices for the plant. Hence, we assumed the availability of either a mathematical model for the plant or possibly a neural network model of it.

If full-state information is not available, one may use more complicated ARMA (AutoRegressive Moving Average) models as illustrated in Figure 6. While a direct chain rule application is still possible, Figure 7 illustrates how gradients may be directly specified using the adjoint network. A variety of other architectures typically used in control may also be considered, such as hierarchical structures, radial basis function networks, and feedforward locally recurrent networks (Puskorious and Feldkamp, 1994). In all cases, the diagrammatic approach provides a direct derivation of the gradient algorithm.

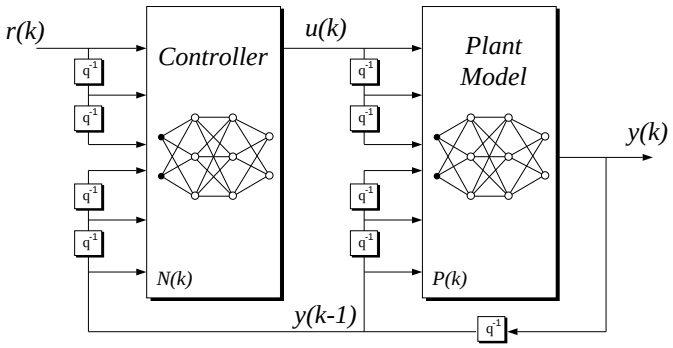


Fig. 6. Neural network control using nonlinear ARMA models.

5.3 Example 3: Cascaded Neural Networks

Let us now turn to an example of two cascaded neural networks (Figure 8), which will further illustrate the advantages of the diagrammatic approach. The inputs to the first network are samples from a time sequence $x(k)$. Delayed outputs of the first network are fed to the second network. The cascaded networks are defined as

$$u(k) = \mathcal{N}_1(W_1, x(k), x(k-1), x(k-2)), \tag{16}$$

$$y(k) = \mathcal{N}_2(W_2, u(k), u(k-1), u(k-2)), \tag{17}$$

where W_1 and W_2 represent the weights parameterizing the networks, $y(k)$ is the output, and $u(k)$ the intermediate signal. Given a desired response for the

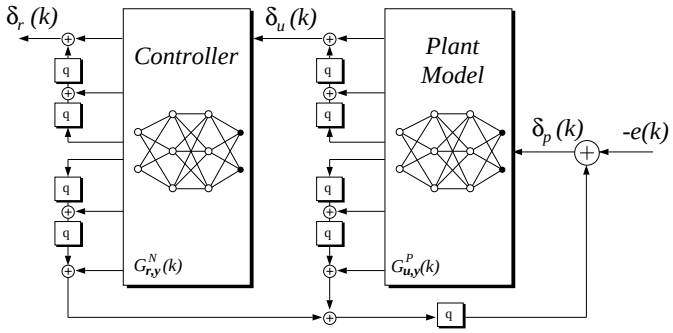


Fig. 7. Adjoint network for control using nonlinear ARMA models.

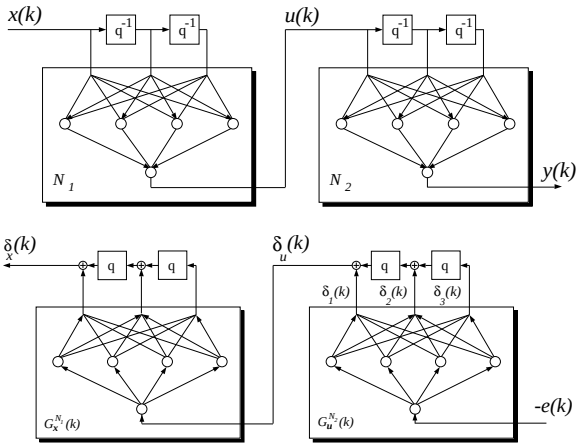


Fig. 8. Cascaded neural network filters and adjoint counterpart.

output y of the second network, it is straightforward to use backpropagation for adapting the second network. It is not obvious, however, what the effective error should be for the first network.

From the adjoint network also shown in Figure 8, we simply label the desired signals and write down the gradient relations:

$$\delta_u(k) = \delta_1(k) + \delta_2(k+1) + \delta_3(k+2), \tag{18}$$

with

$$[\delta_1(k)\ \delta_2(k)\ \delta_3(k)] = -G_u^{\mathcal{N}}(k)\, e(k), \tag{19}$$

i.e., each $\delta_i(k)$ is found by backpropagation through the output network, and the δ_i's (after appropriate advance operations) are summed together. The weight update for the first network is thus given by

$$\Delta W_1(k) = \mu \frac{\partial u(k)}{\partial W_1(k)}\, \delta_u(k) = \mu\, G_{W_1}^{\mathcal{N}_1}(k)\, \delta_u(k), \tag{20}$$

in which the product term is found by a single backpropagation with $\delta_u(k)$ acting as the error to the first network. Equations can be made causal by simply delaying the weight update for a few time steps. Clearly, extrapolating to an arbitrary number of taps is also straightforward.

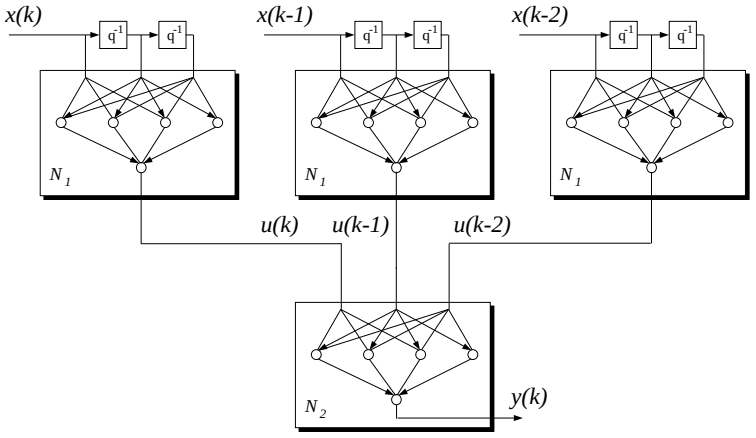


Fig. 9. Cascaded neural network filters unfolded-in-time.

For comparison, let us consider the brute force derivative approach to finding the gradient. Using the chain rule, the instantaneous error gradient is evaluated as

$$\frac{\partial e^2(k)}{\partial W_1} = -\mu e(k) \frac{\partial y(k)}{\partial W_1} \tag{21}$$

$$= -\mu e(k) \left[\frac{\partial u(k)}{\partial W_1} \frac{\partial y(k)}{\partial u(k)} + \frac{\partial u(k-1)}{\partial W_1} \frac{\partial y(k)}{\partial u(k-1)} + \frac{\partial u(k-2)}{\partial W_1} \frac{\partial y(k)}{\partial u(k-2)} \right]$$

$$= \mu \left[\frac{\partial u(k)}{\partial W_1} \delta_1(k) + \frac{\partial u(k-1)}{\partial W_1} \delta_2(k) + \frac{\partial u(k-2)}{\partial W_1} \delta_3(k) \right], \tag{22}$$

where we define

$$\delta_i(k) \triangleq -\frac{\partial y(k)}{\partial u(k-i)} e(k) \qquad i = 1, 2, 3.$$

Again, the δ_i terms are found simultaneously by a single backpropagation of the error through the second network. Each product $\partial u(k-i-1)/\partial W_1 \delta_i(k)$ is then found by backpropagation applied to the first network with $\delta_i(k)$ acting as the error. However, since the derivatives used in backpropagation are time-dependent, *separate* backpropagations are necessary for each $\delta_i(k)$. These equations, in fact, imply backpropagation through an unfolded structure, and are equivalent to *weight sharing* (LeCun *et al.*, 1989) as illustrated in Figure 9. In situations where there may be hundreds of taps in the second network, this

algorithm is far less efficient than the one derived directly using adjoint networks. Similar arguments can be used to derive an efficient on-line algorithm for adapting time-delay neural networks (Waibel *et al.*, 1989). In Section 7.2 we will return to this example to show how the on-line weight-sharing algorithm may also be derived using flow graphs.

5.4 Example 4: Temporal Backpropagation

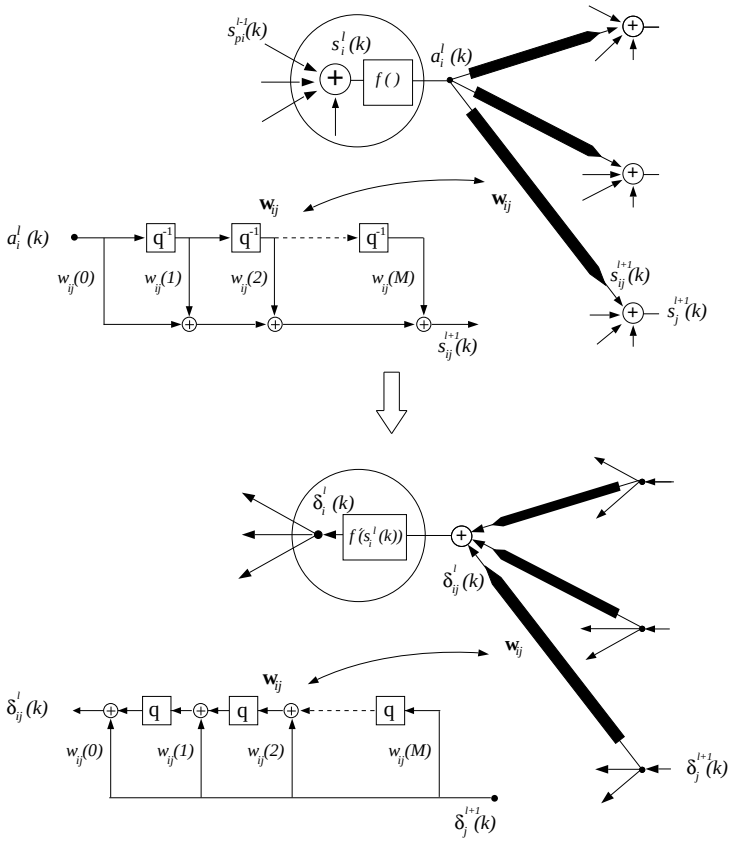


Fig. 10. Block diagram construction of an FIR network and corresponding adjoint network.

An extension of the feedforward network can be constructed by replacing all scalar weights with discrete time linear filters to provide dynamic interconnectivity between neurons. The filters represent a more accurate model of axonal transport, synaptic modulation, and membrane charge dissipation (Koch and Segev, 1989; MacGregor, 1987). Mathematically, a neuron i in layer l may be

specified as

$$s_i^l(k) = \sum_j W_{ij}^l(q^{-1})\, a_j^{l-1}(k) + w_{bias} \tag{23}$$

$$a_i^l(k) = f(s_i^l(k)), \tag{24}$$

where $a(k)$ are neuron output values, $s(k)$ are summing junctions, $f(\cdot)$ are sigmoid functions, and $W(q^{-1})$ are synaptic filters. Three possible forms for $W(q^{-1})$ are

$$W(q^{-1}) = \begin{cases} w & \text{Case I} \\ \sum_{m=0}^{M} w(m) q^{-m} & \text{Case II} \\ \dfrac{\sum_{m=0}^{M} a(m) q^{-m}}{1 - \sum_{m=1}^{M} b(m) q^{-m}} & \text{Case III.} \end{cases} \tag{25}$$

In Case I, the filter reduces to a scalar weight and we have the standard definition of a neuron for feedforward networks. Case II corresponds to a Finite Impulse Response (FIR) filter in which the synapse forms a weighted sum of past values of its input. Such networks have been utilized for time series and system identification problems (Wan, 1993a,b,c). Case III represents the more general Infinite Impulse Response (IIR) filter, in which feedback is permitted. In all cases, coefficients are assumed to be adaptive.

Figure 10 illustrates a network composed of FIR filter synapses realized as tap-delay lines. Deriving the gradient descent rule for adapting filter coefficients is quite formidable if we use a direct chain rule approach. However, using the construction rules described earlier, we may trivially form the adjoint network also shown in Figure 10. By inspection we have

$$\delta_i^l(k) = f'(s_i^l(k)) \sum_j \delta_{ij}^l(k) = f'(s_i^l(k)) \sum_j W_{ij}^{l+1}(q^{+1}) \delta_j^{l+1}(k). \tag{26}$$

Consideration of an output neuron at layer L yields

$$\delta_j^L(k) = -e_j(k) f'(s_j^L(k)).$$

These equations define the algorithm known as *temporal backpropagation* (Wan, 1993a,b). The algorithm may be viewed as a temporal generalization of backpropagation in which error gradients are propagated not by simply taking weighted sums, but by backward filtering. Note that in the adjoint network, backpropagation is achieved through the reciprocal filters $W(q^{+1})$. Since this is a noncausal filter, it is necessary to introduce a delay of a few time steps to implement the on-line adaptation.

In the IIR case, it is easy to verify that Equation 26 for temporal backpropagation still applies with $W(q^{+1})$ representing a noncausal IIR filter. This also

follows directly from the generalization of *rule 5* for linear subsystems. As with backpropagation-through-time, the network must be trained using a forward and backward sweep necessitating storage of all activation values at each step in time. Different realizations for the filters dictate how signals flow through the adjoint structure as illustrated in Figure 11. In all cases, computations remain $\mathcal{O}(N)$ (this is in contrast with $\mathcal{O}(N^2)$ algorithms derived by Back and Tsoi (Back and Tsoi, 1991) using direct chain rule methods). Note that the poles of the forward IIR filters are reciprocal to the poles of the reciprocal filters. Stability monitoring can be made easier if we consider lattice realizations in which case stability is guaranteed if the magnitude of each coefficient is less than 1. Regardless of the choice of the filter realization, adjoint networks provide a simple unified approach for deriving a learning algorithm[7].

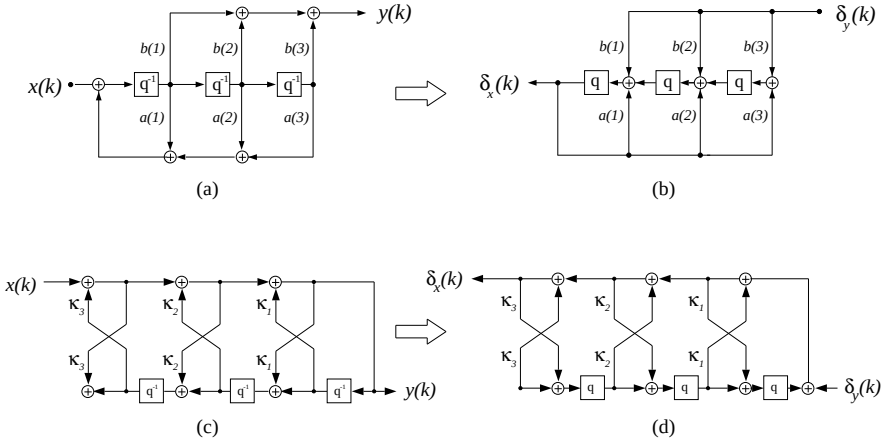


Fig. 11. IIR filter realizations: (a) controller canonical, (b) adjoint observer canonical, (c) lattice, (d) adjoint lattice.

6 On-line Algorithms and Transposed Networks

For recurrent architectures, the algorithms derived using the adjoint networks result in off-line approaches. This is a fundamental consequence of the weight update of Equation 2, which evaluates the change in the total cost function into the future due to a change in a weight at the current time step. In general the

[7] In related work (Beaufays and Wan, 1994b), the diagrammatic method was used to derive an algorithm to minimize the output power at each stage of an FIR lattice filter. This provides an adaptive lattice predictor used as a decorrelating preprocessor to a second adaptive filter. The new algorithm is more effective than Griffiths' algorithm (Griffiths, 1977).

algorithms can be seen as different versions of backpropagation-through-time (BPTT) which result from Euler-Lagrange type optimization. We now turn to the weight update given by Equation 3, repeated here for convenience:

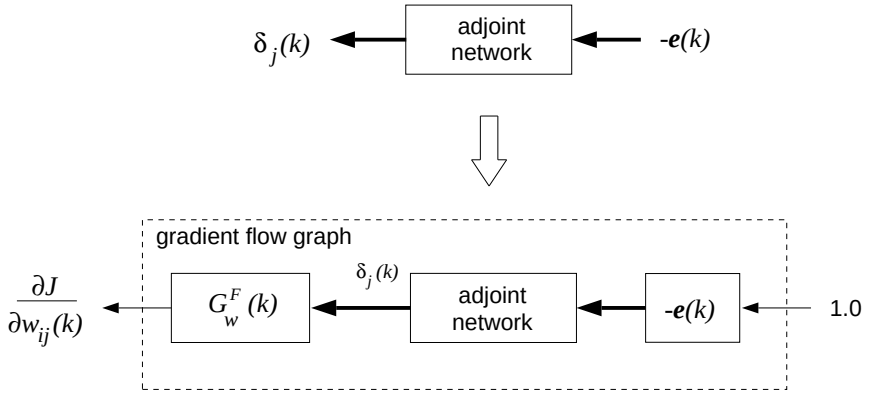
$$\Delta\tilde{W}(k) = -\mu \frac{\partial J_k}{\partial W}. \tag{27}$$

This leads to on-line algorithms by evaluating the gradient as the change in the cost function only up to the current time step due to the changes in the weights at all prior time steps. Such algorithms require a recursive estimation of the gradient and in the neural network literature go by such names as real-time backpropagation, real-time recurrent learning, on-line backpropagation, or dynamic backpropagation (Williams and Zipser, 1989; Narendra and Parthasarathy, 1990; Hertz *et al.*, 1991).

Continuing the development of diagrammatic approaches, we show how such on-line algorithms may again be constructed directly by using simple flow graph manipulation. The approach is to transform the gradient flow-graph implied by the adjoint network into a transposed flow-graph. By flow-graph interreciprocity, the two flow-graphs will be shown to have identical transfer functions.

6.1 Gradient Flow Graph

Consider the adjoint network represented below. By adding to the output of the adjoint network the branch $G_w^F(k)$, and re-drawing the input $-\mathbf{e}(k)$ as a linear branch with gain $-\mathbf{e}(k)$ and input 1.0, we obtain a *gradient flow graph* that explicitly gives the gradient $\partial J/\partial w_{ij}(k)$.

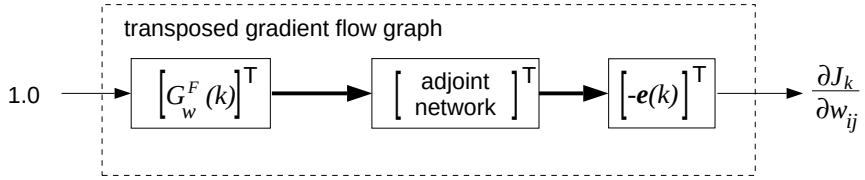


Note that we have constructed a single-input-single-output (SISO) flow graph for adapting a single *scalar* weight. The actual weight update is given by Equation 2. This representation of the gradient calculation will be used to derive on-line gradient algorithms through a set of block diagram manipulations.

6.2 Construction Rules for On-line Gradient Flow Graphs

We begin by providing the step-by-step procedure for constructing the on-line gradient network from the off-line flow graph. The necessary operations are as follows:

1. Select a scalar weight $w_{ij}(k)$ for which an on-line update is desired, and locate the output $\delta_j(k)$ in the adjoint network associated with these weights. (All other outputs $\delta_p(k)$ not associated with the weights of interest can be ignored.)
2. Form the gradient network as described above.
3. Form the *transposed gradient flow-graph* by reversing the branch directions, transposing the branch gains, replacing summing junctions by branching points and vice versa, replacing q^{+1} with q^{-1}, and interchanging source and sink nodes. The input to this new transposed flow-graph is 1.0 and the output $\partial J_k/\partial w_{ij}$.

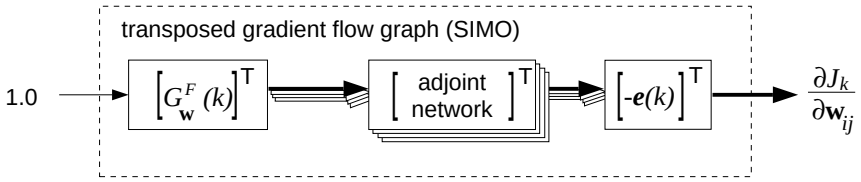


   Note that, strictly speaking, the transposition of a flow graph does not include the replacement of time delay operators with time advances and vice versa. Here we use the term "transpose" by abuse of notation, because if we were to unravel in time the gradient flow graph and the so-called transposed gradient flow graph, the two unraveled flow graphs would be the true transpose of one another. We will come back to this point in Section 6.4.
4. Set to zero any input in the transposed weight-update flow graph that corresponds to an external input $x(k)$ in the original network (outputs $\delta_x(k)$ in the adjoint network). This is justified by noting that $\partial \mathbf{x}(k)/\partial w = 0$.
5. Repeat the process from *step 1* for any other desired weight in the network.

By simply labeling necessary signals, exact equations may be "read-out" directly from the transposed flow-graph completing the derivation. The on-line weight update is then given as in Equation 27. Note that the output $\partial J_k/\partial w$ of the new transposed flow-graph is not equivalent to the output $\partial J/\partial w(k)$ of the first gradient flow graph.

In the adjoint network, a *single* network suffices to yield all the deltas associated with each weight in the network. In the on-line approach, however, a *different* transposed flow graph is needed for each scalar weight to be adapted. Often it may be convenient to work directly with a vector of weights $\mathbf{w}_{ij}(k)$. In this case, $G_{\mathbf{W}}^F$ becomes a matrix and the original off-line gradient flow graph is single input multiple output (SIMO). The procedure described in rule 3 is still valid although, strictly speaking, the resulting SIMO flow graph is *not* the

transpose of the original. There is a loss of topological equivalence as branches in the "transposed" flow-graph change dimension (this will be illustrated in an example shortly). For convenience we will still refer to this as a transposed flow graph. However, the interpretation of the resulting flow graph is really of a stack of SISO transposed graphs. The computational complexity is thus equivalent to successively forming SISO transposed flow graphs for each weight in the network. The vector transposed flow graph is illustrated here:



Finally, it should be noted that it is possible to specify a set of rules that go directly from the original neural network to the on-line gradient flow graph. However, this is not recommended. For example, constructing the transposed flow graph directly for a static feedforward network does not lead to the obvious implementation for standard backpropagation. In general, the adjoint network should be constructed first. The specific set of weights that require conversion to an on-line update can then be identified. For these weights, the associated transposed flow graphs may then be formed.

6.3 Relating Real-Time Recurrent-Learning and Backpropagation-Through-Time

The procedure described above can be illustrated for a network with output feedback as redrawn in Figure 12(a). Note that any connectionist architecture with feedback units can, in fact, be represented canonically as a feedforward network with output feedback (Piche, 1994). The results presented here are thus general for all architectures (additional examples for some specific architectures are given in Section 6).

Recall that the equations for the system are given by

$$\mathbf{y}(k) = \mathcal{N}(W, \mathbf{x}(k), \mathbf{y}(k-1)), \tag{28}$$

and that from the adjoint network in Figure 12(b), the equations for BPTT are given by

$$\boldsymbol{\delta}(k) = G_{\mathbf{y}}^{\mathcal{N}}(k+1)\boldsymbol{\delta}(k+1) - \mathbf{e}(k). \tag{29}$$

Also, the weight update was given by

$$\Delta W(k) \;=\; G_{W}^{\mathcal{N}}(k) \cdot \boldsymbol{\delta}(k). \tag{30}$$

In Figure 12(c) we form the gradient flow graph from the adjoint network. Figure 12(d) shows the final transposed gradient network. Note that q^{+1} has

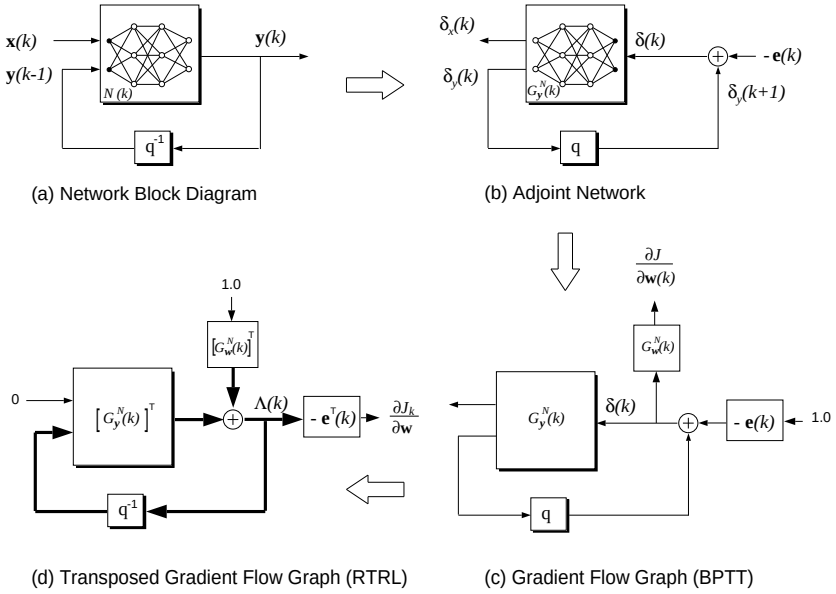


Fig. 12. Diagrammatic derivation of real-time recurrent learning and backpropagation-through-time.

been replaced with q^{-1} as in the original network. Labeling the signals within the transposed graph, we directly read-off the equations:

$$\Lambda(k) = [G^{\mathcal{N}}_{\mathbf{y}}(k)]^T \cdot \Lambda(k-1) + [G^{\mathcal{N}}_{W}(k)]^T \qquad \forall k = 1 \ldots K. \tag{31}$$

Note that this is a matrix update equation as opposed to the vector update of Equation 29. This is reflected in the change in dimensionality of the branches in the transposed graph. The Jacobians $G^{\mathcal{N}}_{\mathbf{y}}(k)$ and $G^{\mathcal{N}}_{W}(k)$ in Equation 31 have, respectively, dimensions $N_y \times N_y$ and $N_w \times N_y$; the gradient term $\Lambda(k)$ has dimensions $N_y \times N_w$, where N_w is the number of weights in the neural network and N_y is the number of feedback states.

From the flow graph, the weight update at each time step is given by

$$\Delta\tilde{\mathbf{w}}^T(k) \;=\; -\mu\, \mathbf{e}(k)^T \cdot \Lambda(k). \tag{32}$$

By comparing this to Equation 3 and assuming a squared-error cost function, it is immediate that $\Lambda(k) = \partial \mathbf{y}(k)/\partial W$. Equations 31 and 32 describe the algorithm known as Real-Time Recurrent Learning (RTRL) for feedforward networks with output feedback (see, e.g., Narendra and Parthasarathy, 1990). RTRL or equivalently real-time backpropagation (RTBP) was first introduced for single-layer fully recurrent networks by Williams and Zipser[8] (Williams and Zipser, 1989). The algorithm is well suited for on-line adaptation of dynamic networks where a desired response is specified at each time step.

[8] The name "recurrent backpropagation" is also occasionally used, although this

Computational Cost of RTRL and BPTT

Another nice feature of flow graph representations is that the computational and complexity differences between RTRL and BPTT can be directly observed from their respective flow graphs. By observing the dimension of the terms flowing in the graphs and the necessary matrix calculations and multiplications, it can be verified that BPTT requires only $\mathcal{O}(N_w)$ operations. This holds for any adjoint network. On the other hand, we see that RTRL requires $\mathcal{O}(N_y^2 N_w)$ operations. For fully recurrent networks $N_w = N_y^2$ and RTRL is $\mathcal{O}(N_y^4)$. However, for certain architectures, where the feedback is restricted, the computational complexity of the on-line algorithms may be less severe (see the description of Gamma networks in Section 7.3).

An additional source of complexity in RTRL involves the calculation of the Jacobians. In the adjoint network, it is not necessary to calculate the Jacobian matrices $G_y^{\mathcal{N}}$ and $G_w^{\mathcal{N}}$ explicitly. The product $G_y^{\mathcal{N}}(k) \cdot \boldsymbol{\delta}(k)$ is formed directly by backpropagating through the network to the inputs, while the product $G_w^{\mathcal{N}}(k) \cdot \boldsymbol{\delta}(k)$ constitutes a standard backpropagation weight update with error $\boldsymbol{\delta}(k)$. In the transposed flow graph, however, the Jacobians are propagated through the graph and require explicit calculation. Note that the j^{th} row of either Jacobian matrix may be calculated by backpropagating the unit vectors $\mathbf{e}_j = [0, \cdots, 0, 1, 0 \cdots 0]$ through the network.

6.4 Transposed Flow-Graphs and Interreciprocity

The above example illustrated the simple diagrammatic procedure to derive an on-line algorithm. We now provide a justification for this procedure, and at the same time show an equivalence between the computations performed in RTRL versus BPTT. As stated earlier, the generality of the architecture considered validates these arguments for all network architectures.

Again we start with the adjoint network and form the gradient flow graph. However, rather than directly transposing this flow-graph, we first convert the time-dependent flow graph into a time-independent flow graph. This is accomplished by taking the gradient flow graph (see step 2 and Figure 12(c)) and unraveling it backward in time. The result is illustrated in Figure 13, which shows explicitly the reverse time flow for implementing BPTT. The input (source) to this flow-graph, 1.0, is distributed along the top branch; the individual outputs $\partial J/\partial w(k)$ are summed together along the lower branch to form the accumulated gradient $\partial J/\partial w$. Note that all transmittances in the flow graph are linear and time-invariant.

Let us now build a new flow graph by transposing the flow graph of Figure 13. The new flow graph is represented in Figure 14. This corresponds to the unraveled representation (with accumulated gradient) of the transposed flow graph and explicitly gives the equations for RTRL.

---

should not be confused with the approach developed by Pineda (Pineda, 1987) for learning fixed points in feedback networks. The linear equivalent of the RTRL algorithm was first introduced in the context of IIR filter adaptation (White, 1975).

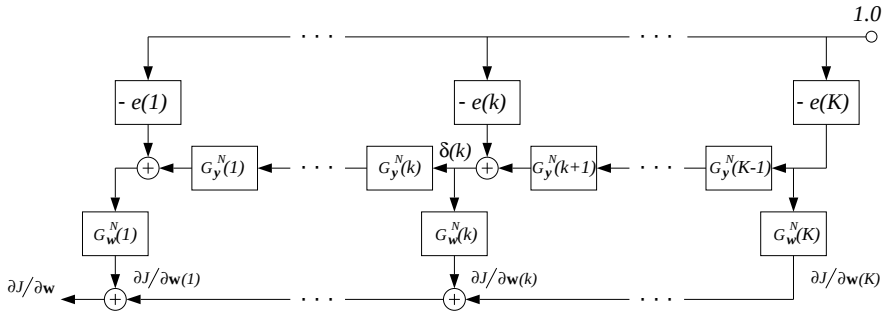


Fig. 13. Flow-graph representation of the backpropagation-through-time algorithm.

Flow graph theory shows that transposed flow graphs with time-independent transmittances are a particular case of interreciprocal graphs (see Appendix A). This means, in a SISO case, that the sink value obtained in one graph, when exciting the source with a given input, is the same as the sink value of the transposed graph, when exciting its source by the same input. Thus, if an input of 1.0 is distributed along the lower horizontal branch of the transposed graph, the output, which is now accumulated on the upper horizontal branch, will be equal to $\partial J/\partial w$. This accumulated gradient is identical to the output of our original flow graph, *i.e.* $\sum_{k=1}^{K} \Delta W(k) = \sum_{k=1}^{K} \Delta \tilde{W}(k)$. This of course does not imply that the instantaneous weight-update terms are equal, *i.e.* $\Delta W(k) \neq \Delta \tilde{W}(k)$.

As explained in Section 6.2, the flow graphs introduced here are SIMO, and the arguments of interreciprocity may be applied by considering the SIMO graph to be a stack of SISO graphs, each of which can be independently transposed.

To complete the reconstruction of the transposed gradient flow graph, we need only ravel the network back into the more compact recurrent representation. Due to the reversal of signal flow, the operator q^{+1} in the original gradient flow

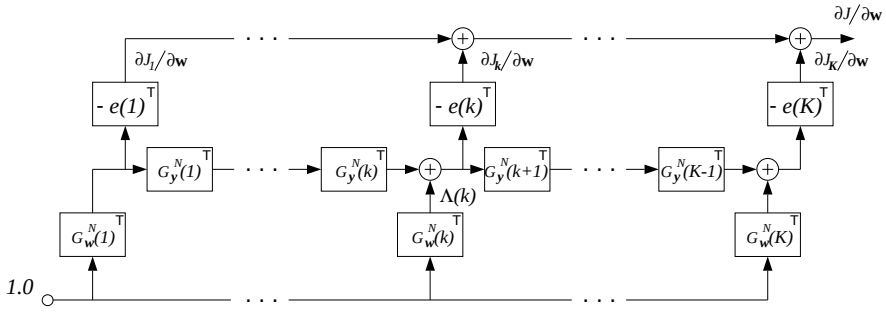


Fig. 14. Flow graph associated with the real-time recurrent learning algorithm.

graph becomes q^{-1} in the final transposed graph.

7 Examples of On-line Learning Algorithms

We provide some additional examples for specific architectures, which further illustrates the procedure.

7.1 Example 1: Neural Control - Dynamic Backpropagation

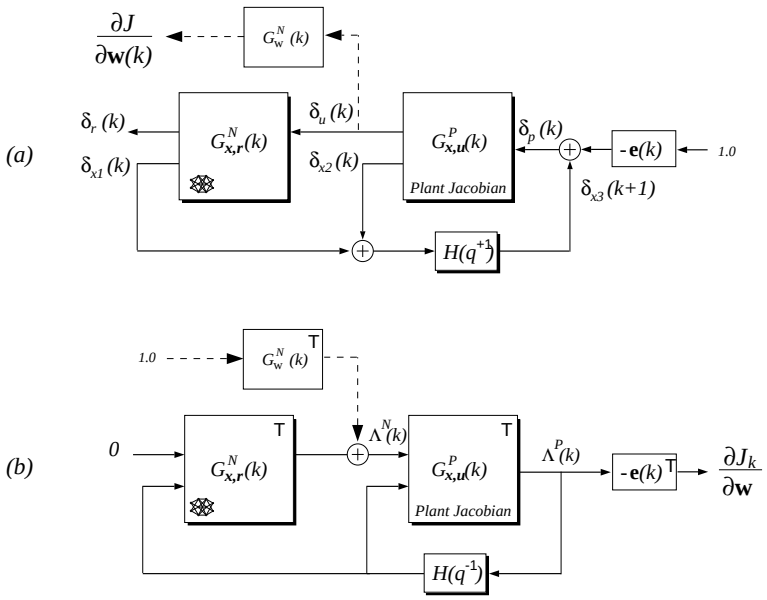


Fig. 15. Neural control architecture: (a) gradient flow graph, (b) transposed gradient flow graph.

Figure 15 shows the flow graph approach applied to a recurrent network and control architecture (see Section 5.2 and Figure 5 for the neural network structure and adjoint network). Note that we have made a slight modification in which feedback is through a fixed linear matrix transformation $\mathbf{H}(q^{-1})$. This provides for a compact notation to allow a variety of alternatives. For example, if $\mathbf{H}(q^{-1}) = q^{-1}\mathbf{I}$, then we have standard state-feedback. If $\mathbf{H}(q^{-1}) = (\sum_{i=0}^{M} q^{-i})\mathbf{I}$, then this specifies a tap-delay feedback for each output. Directly from the transposed graph, we may read-off the equations

$$\Lambda^P(k) = [G_u^P(k)]^T \cdot \Lambda^N(k) + [G_{\mathbf{x}}^P(k)]^T \mathbf{H}(q^{-1})\Lambda^P(k) \tag{33}$$

$$\Lambda^N(k) = [G^{\mathcal{N}}_{\mathbf{x}}(k)]^T \mathbf{H}(q^{-1}) \Lambda^P(k) + [G^{\mathcal{N}}_{W}(k)]^T \tag{34}$$

$$\Delta \tilde{\mathbf{w}}^T(k) = -\mu\, \mathbf{e}(k)^T \cdot \Lambda^P(k). \tag{35}$$

These equations specify a variation of RTRL, which is referred to by Narendra as Dynamic Backpropagation (Narendra and Parthasarathy, 1990).

7.2 Example 2: Cascaded Neural Networks - Weight Sharing

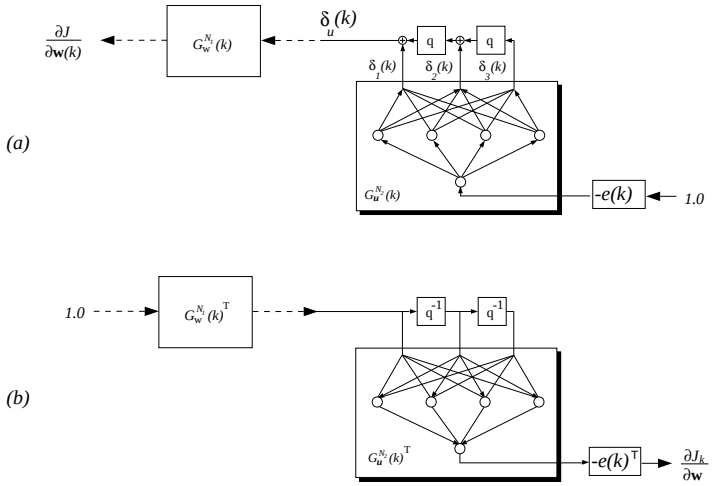


Fig. 16. Cascaded neural networks: (a) gradient flow graph, (b) transposed gradient flow graph.

In Section 5.3 we showed how the adjoint network may be used to derive an efficient on-line algorithm for cascaded neural networks (causality was achieved by buffering of states). By using brute force chain rules, an alternative on-line algorithm was also derived. It is now shown how the later algorithm corresponds to the transposed gradient flow graph.

In Figure 16 we show the transformation from the gradient flow graph to the transposed gradient flow graph (see Figure 8 for original cascade architecture and associated adjoint network). Note that we consider only the transpose for finding the weight update associated with the first neural network. As argued earlier, the weights in the second neural network may be adapted on-line by standard backpropagation. Directly from the transposed flow graph we may read the equation:

$$\Delta \tilde{W}_1(k) = \mu[G^{\mathcal{N}_1}_{W_1}(k) \cdot \delta_1(k) + G^{\mathcal{N}_1}_{W_1}(k-1) \cdot \delta_2(k) + G^{\mathcal{N}_1}_{W_1}(k-2) \cdot \delta_3(k)], \tag{36}$$

where we have made the substitution $-G^{\mathcal{N}_2}_i(k)e(k) = \delta_i(k)$ corresponding to a backpropagation of the error through the second network. The products $G^{\mathcal{N}_1}_{W_1}(k-$

$i) \cdot \delta_i(k)$ correspond to separate backpropagations of $\delta_i(k)$ to adapt the weights in the first network. These equations are the same as those found earlier (see Equation 22), and are again equivalent to backpropagation through the unfolded structure shown in Figure 9.

7.3 Example 3: Gamma Neural Networks

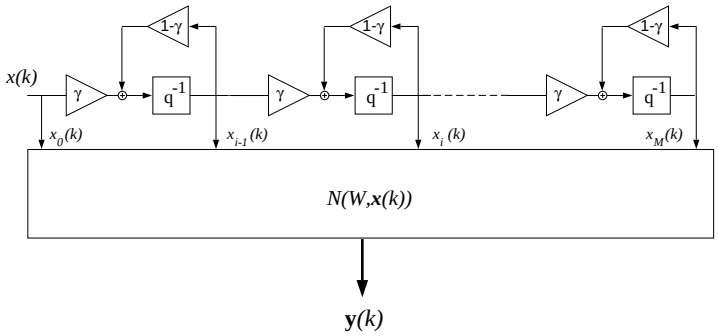


Fig. 17. Gamma neural network.

Consider the single-input *focused Gamma network* shown in Figure 17 (De Vries and Principe, 1992). A tapped delay line feeds a feedforward network where the standard delay q^{-1} has been replaced by the simple low-pass filter of transfer function $\frac{\gamma}{q-(1-\gamma)}$. This allows for an IIR filter where the memory depth is controlled by the variable parameter γ. The gamma filter may be defined in the time domain as

$$x_i(k) = (1-\gamma)x_i(k-1) + \gamma x_{i-1}(k), \qquad i = 1, ...M \tag{37}$$

$$\mathbf{y}(k) = \mathcal{N}(W, \mathbf{x}(k)), \tag{38}$$

where $x_0(k) = x(k)$ is the input. The first step is to create the adjoint network. The weight-update may then be found from the adjoint network by noting that $G_\gamma = x_{i-1}(k)$ and $G_{1-\gamma} = -x_i(k)$, as illustrated in Figure 18. Note that γ is constrained to be the same for all segments of the filter; thus, a single weight update is formed by summing the contributions from each segment. While the weights in the feedforward network may be adapted through standard backpropagation without delay, adaptation of γ involves an off-line algorithm in which the gamma filter is run backward in time.

To form an on-line algorithm, we transpose the gradient flow graph as shown in Figure 19. Directly from this flow-graph we may read-off the on-line equations:

$$\Delta\gamma = -\mu[\mathbf{e}(k)]^T[G^{\mathcal{N}}_{\mathbf{x}}(k)]^T\boldsymbol{\alpha}(k), \tag{39}$$

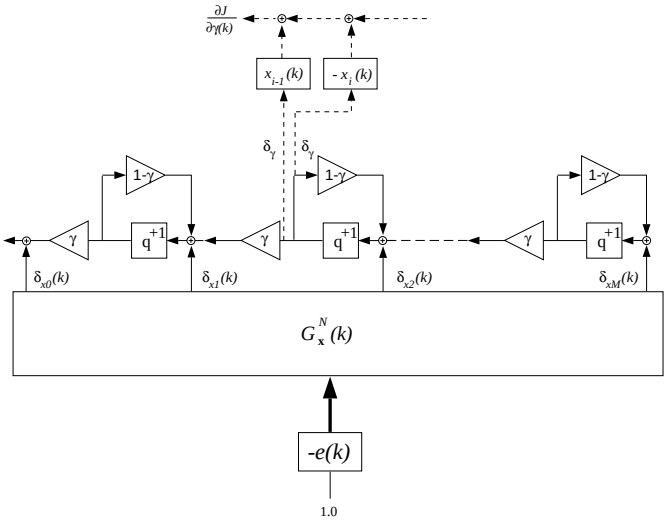


Fig. 18. Gamma network: gradient flow graph.

where

$$\alpha_i(k) = (1-\gamma)\alpha_i(k-1) + \gamma\alpha_{i-1}(k) + [x_{i-1}(k-1) - x_i(k-1)], \qquad (40)$$

for $i = 1, ...M$. Note that the product $[\mathbf{e}(k)]^T[G^{\mathcal{N}}_{\mathbf{X}}(k)]^T$ may still be efficiently calculated with standard backpropagation of the error $\mathbf{e}(k)$, so that implementation of Equation 40 is only order M. Thus, the overall implementation of the on-line algorithm for the gamma network is computationally equivalent to the off-line algorithm. This example also illustrates how the transposed flow-graph is used for only a subset of weights in the network (in this case γ), while the rest of the weights (W within the feedforward network) are adapted according to the adjoint network.

8 Summary

In this chapter, we have provided a unified framework for the derivation of gradient-based algorithms for arbitrary network architectures, network configurations, and systems. One starts with a diagrammatic representation of the network of interest. An adjoint network is then constructed by simply swapping summing junctions with branching points, continuous functions with derivative transmittances, and time delays with time advances. The final algorithm is read directly off the adjoint network. No explicit chain rules are needed.

In the case of recurrent architectures, an on-line algorithm may also be derived by starting with the adjoint network and then producing a transposed gradient flow graph. This allows us to relate backpropagation-through-time and

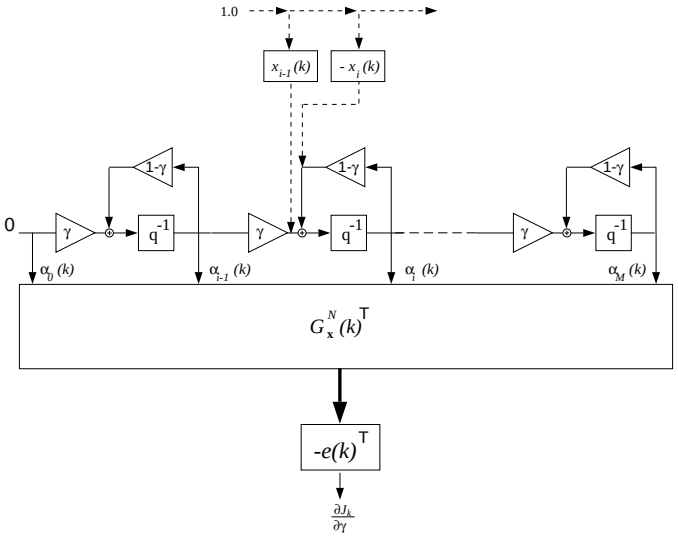


Fig. 19. Gamma network: transposed gradient flow graph.

real-time backpropagation, as well as to verify that the global gradient calculations performed by the algorithms are equivalent.

While we have concentrated on gradient descent methods, the approach can be extended to other optimization techniques. For example, second-order methods require the additional calculation of the matrix of second derivatives (Hessian). This can be derived diagrammatically by considering the gradient flow graph (with first derivative outputs) as the starting point for applying the method a second time to take the second derivative. The diagrammatic framework can also be applied to instances where the network interacts with *graphs* such as Hidden Markov Models (HMMs), and all parameters of the system are to be adapted. Additional extensions to reinforcement learning algorithms are also currently under development.

A Flow Graph Interreciprocity

In this appendix we provide the formal definition of interreciprocity. We then prove that transposed flow graphs are interreciprocal, and that the transfer functions of SISO interreciprocal flow graphs are identical.

Let $\mathcal{F}$ be a flow graph. In $\mathcal{F}$, we define Y_k, the value associated with node k; $T_{j,k}$, the transmittance of the branch (j,k); and $V_{j,k} = T_{j,k} \cdot Y_j$, the output of branch (j,k). Let us further assume that each node k of the graph has associated with it a source node, i.e. a node connected to it by a branch of unity transmittance. Let X_k be the value of this source node (if node k has no associated source node, X_k is simply set to zero). It results from these definitions that

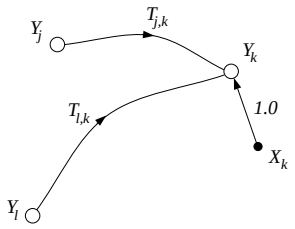


Fig. 20. Example of nodes and branches in a signal flow graph.

$Y_k = \sum_j V_{j,k} + X_k = \sum_j T_{j,k} \cdot Y_j + X_k$ (see Figure 20).

Let us now consider a second flow graph, $\tilde{\mathcal{F}}$, having the same topology as $\mathcal{F}$ (i.e., $\tilde{\mathcal{F}}$ has the same set of nodes and branches as $\mathcal{F}$, but the branch transmittances of both graphs may differ). $\tilde{\mathcal{F}}$ is described with the variables $\tilde{Y}_k, \tilde{T}_{j,k}, \tilde{V}_{j,k}$, and $\tilde{X}_k$.

Definition 1 *Two flow graphs, $\mathcal{F}$ and $\tilde{\mathcal{F}}$, are said to be the transpose of each other iff their transmittance matrices are transposed, i.e.*

$$\tilde{T}_{j,k} = T_{k,j} \qquad \forall \ j,k. \tag{41}$$

Definition 2 (Bordewijk, 1956): *Two flow graphs, $\mathcal{F}$ and $\tilde{\mathcal{F}}$, are said to be interreciprocal iff*

$$\sum_k (\tilde{Y}_k \cdot X_k - Y_k \cdot \tilde{X}_k) = 0. \tag{42}$$

We can now state the following theorem:

Theorem 1 *Transposed flow graphs are interreciprocal.*

Proof: Let $\mathcal{F}$ be a flow graph, and let $\tilde{\mathcal{F}}$ be the transpose of $\mathcal{F}$. We start from the identity $\sum_k \tilde{Y}_k \cdot Y_k \equiv \sum_k Y_k \cdot \tilde{Y}_k$, and replace Y_k by $\sum_j T_{j,k} \cdot Y_j + X_k$ in the first member, and $\tilde{Y}_k$ by $\sum_j \tilde{T}_{j,k} \cdot \tilde{Y}_j + \tilde{X}_k$ in the second member (Oppenheim and Schafer, 1989). Rearranging the terms, we get

$$\sum_{j,k} (\tilde{Y}_k \cdot V_{j,k} - Y_k \cdot \tilde{V}_{j,k}) + \sum_k (\tilde{Y}_k \cdot X_k - Y_k \cdot \tilde{X}_k) = 0. \tag{43}$$

Equation 43 is usually referred to as "the two-network form of Tellegen's theorem" (Tellegen, 1952; Penfield *et al.*, 1970). Since $\tilde{\mathcal{F}}$ is the transpose of $\mathcal{F}$, the first term of Equation 43 can be rewritten as $\sum_{j,k}(\tilde{Y}_k \cdot V_{j,k} - Y_k \cdot \tilde{V}_{j,k}) = \sum_{j,k}(\tilde{Y}_k \cdot T_{j,k} \cdot Y_j - Y_k \cdot \tilde{T}_{j,k} \cdot \tilde{Y}_j) = \sum_{j,k}(\tilde{Y}_k \cdot T_{j,k} \cdot Y_j - Y_k \cdot T_{k,j} \cdot \tilde{Y}_j) = 0$. Since the first term of Equation 43 is zero, the second term $\sum_k (\tilde{Y}_k \cdot X_k - Y_k \cdot \tilde{X}_k)$ is also zero. The flow graphs $\tilde{\mathcal{F}}$ and $\mathcal{F}$ are thus interreciprocal. QED.

The last step consists in showing that SISO interreciprocal flow graphs have the same transfer functions. Let node a be the unique source of $\mathcal{F}$ and node b its unique sink. From the definition of transposition, node a is the sink of $\tilde{\mathcal{F}}$, and node b is its source. We thus have $X_k = 0 \quad \forall k \neq a$ and $\tilde{X}_k = 0 \quad \forall k \neq b$. Therefore, Equation 42 reduces to:

$$X_a \cdot \tilde{Y}_a = \tilde{X}_b \cdot Y_b. \tag{44}$$

This last equality can be interpreted as follows (Penfield *et al.*, 1970; Oppenheim and Schafer, 1989): the output Y_b, obtained when exciting graph $\mathcal{F}$ with an input signal X_a, is identical to the output $\tilde{Y}_a$ of the transposed graph $\tilde{\mathcal{F}}$ when exciting it at node b with an input $\tilde{X}_b \equiv X_a$. The transfer functions of the SISO systems represented by the two flow graphs are thus identical, which is the desired conclusion.

B Proof of Adjoint Construction Rules

We show that the diagrammatic method constitutes a formal derivation for arbitrary network architectures. Intuitively, the chain rule applied to individual building blocks yields the adjoint architecture. However, delay operators, which cannot be differentiated, as well as feedback, prevent a straightforward chain rule approach to the proof. Instead, we use a more rigorous approach as follows:

1. We will initially assume that only *univariate* functions exist within the network. This is by no means restrictive. It has been shown (Hornik *et al.*, 1989; Cybenko, 1989) that a feedforward network with two or more layers and a sufficient number of internal neurons can approximate any *uniformly continuous* multivariate function to an arbitrary accuracy. A feedforward network is, of course, composed of simple univariate functions and summing junctions. Thus any multivariate function in the overall network architecture is assumed to be well approximated using a univariate composition.

2. We may completely specify the topology of a network by the set of equations

$$a_j(k) = \sum_i T_{ij} \circ a_i(k) \qquad \forall j \tag{45}$$

$$T \in \{f(),\ q^{-1}\}, \tag{46}$$

where $a_j(k)$ is the signal corresponding to the node a_j at time k. The sum is taken over all signals $a_i(k)$ that connect to $a_j(k)$, and T_{ij} is a transmittance operator corresponding to either a univariate function (*e.g.*, sigmoid function, constant multiplicative weight), or a delay operator. (The symbol $\circ$ is used to remind us that T is an *operator* whose argument is a.) The signals $a_j(k)$ may correspond to inputs ($a_j \triangleq x_j$), outputs ($a_j \triangleq y_j$), or internal signals to the network. Feedback of signals is permitted.

3. Let us add to a specific node a^* a perturbation $\Delta a^*(k)$ at time k. The perturbation propagates through the network, resulting in effective perturbations $\Delta a_j(k)$ for all nodes in the network. Through a continuous univariate function in which $a_j(k) = f(a_i(k))$ we have, to first order

$$\Delta a_j(k) = \frac{\partial a_j(k)}{\partial a_i(k)} \Delta a_i(k) = f'(a_j(k)) \Delta a_i(k), \tag{47}$$

where it must be clearly understood that $\Delta a_j(k)$ and $\Delta a_i(k)$ are the perturbations directly resulting from the external perturbation $\Delta a^*(k)$. Through a delay operator, $a_j(k) = q^{-1} a_i(k) = a_i(k-1)$, we have

$$\Delta a_j(k) = \Delta a_i(k-1) = q^{-1} \Delta a_i(k). \tag{48}$$

Combining these two results with Equation 45 gives

$$\Delta a_j(k) = \sum_i T'_{ij} \circ \Delta a_i(k) \qquad \forall j, \tag{49}$$

where we define $T' \in \{f'(a_i(k)), q^{-1}\}$. Note that $f'(a_i(k))$ is a linear time-dependent transmittance. Equation 49 defines a *derivative network* that is topologically identical to the original network (*i.e.*, one-to-one correspondence between signals and connections). Functions are simply replaced by their derivatives. This is a rather obvious result, and simply states that a perturbation propagates through the same connections and in the same direction as would normal signals.

4. The derivative network may be considered a time-dependent system with input $\Delta a^*(k)$ and outputs $\Delta \mathbf{y}(k)$. We next convert this to a time-independent flow graph by unfolding the system in time (as was done for the gradient flow graph in Section 6.4. Each stage in the unfolded network has a different set of transmittance values corresponding to the time step. The unraveling process stops at the final time K (K is allowed to approach ∞). In addition, the outputs $\Delta \mathbf{y}(n)$ at each stage are multiplied by $-\mathbf{e}(n)^T$ and then summed over all stages to produce a single output $\Delta J \triangleq \sum_{n=k}^{K} -\mathbf{e}(n)^T \Delta \mathbf{y}(n)$.

By linearity, all signals in the flow graph can be divided by $\Delta a^*(k)$ so that the input is now 1, and the output is $\Delta J / \Delta a^*(k)$. In the limit of small $\Delta a^*(k)$

$$\lim_{\Delta a^*(k) \to 0} \frac{\Delta J}{\Delta a^*(k)} = \lim_{\Delta a^*(k) \to 0} \sum_{n=k}^{K} -\mathbf{e}(n)^T \frac{\Delta \mathbf{y}(n)}{\Delta a^*(k)} = \sum_{n=k}^{K} \frac{\partial \mathbf{e}(n)^T \mathbf{e}(n)}{\partial a^*(k)}. \tag{50}$$

Since the system is causal, the partial of the error at time n with respect to $a^*(k)$ is zero for $n < k$. Thus,

$$\sum_{n=k}^{K} \frac{\partial \mathbf{e}(n)^T \mathbf{e}(n)}{\partial a^*(k)} = \sum_{n=1}^{K} \frac{\partial \mathbf{e}(n)^T \mathbf{e}(n)}{\partial a^*(k)} = \frac{\partial J}{\partial a^*(k)} \triangleq \delta^*(k). \tag{51}$$

The term $\delta^*(k)$ is precisely what we were interested in finding. However, calculating all the $\delta_i(k)$ terms would require *separately* propagating signals through the unraveled network with an input of 1 at each location associated with $a_i(k)$. The entire process would then have to be repeated at every time step.

5. Next, take the unraveled network (*i.e.*, flow graph) and form its transpose. By the principle of flow-graph interreciprocity (see Appendix A), the two graphs have identical transfer functions. Thus, an input of 1.0 in the transposed flow-graph again results in $\delta^*(k)$ at the output.

6. The transposed graph can now be raveled back in time to produce the *adjoint network*. Since the direction of signal flow has been reversed, delay operators q^{-1} become advance operators q^{+1}. The node $a^*(k)$, which was the original source of an input perturbation, is now the output $\delta^*(k)$ as desired. The outputs of the original network become inputs with value $-\mathbf{e}(k)$.

7. The selection of the specific node $a^*(k)$ is arbitrary. Had we started with any node $a_i(k)$, we would still have arrived at the same result. In all cases, the input to the adjoint network would still be $-\mathbf{e}(k)$. Thus, by symmetry, *every* signal in the adjoint network provides $\delta_i(k) = \partial J / \partial a_i(k)$.

Summarizing the steps involved in finding the adjoint network, we start with the original network, form the derivative network, unravel in time, transpose, and then ravel back up in time. These steps are accomplished directly by starting with the original network and simply swapping branching points and summing junctions (*rules 1 and 2*), functions f for derivatives f' (*rule 3*), and q^{-1}'s for q^{+1}'s (*rule 5*).

Acknowledgments

This work was sponsored in part by the NSF under grant ECS-9410823.

References

Back, A., and Tsoi, A. 1991. FIR and IIR synapses, a new neural networks architecture for time series modeling. *Neural Computation*, vol. 3, no. 3, pages 375-85.

Beaufays, F., and Wan, E. 1994a. Relating real-time backpropagation and backpropagation-through-time: an application of flow graph interreciprocity. *Neural Computation*, vol. 6, no. 2, pages 296-306.

Beaufays, F., and Wan, E. 1994b. An efficient first-order stochastic algorithm for lattice filters, *ICANN'94*, vol. 2, pages 1021-1024, May 26-29, Sorrento, Italy.

Bordewijk, J. 1956. Inter-reciprocity applied to electrical networks. *Appl. Sci. Res.* 6B, pages 1-74.

Bottou, L., and Gallinari, P. 1991. A framework for the cooperation of learning algorithms. In *Advances in Neural Information Processing Systems*, vol. 4. Lippmann, R. P., Moody, J., Touretzky, D. eds. Morgan Kaufmann, pages 781-788.

Bryson, A. and Ho, Y. 1975. *Applied Optimal Control.* Hemisphere Publishing Corp., New York.

Campolucci, P., Marchegiani A., Uncini A., and Piazza F. Signal-Flow-Graph Derivation of on-line gradient learning algorithms. *Proc. ICNN-97*, Houston, June 1997, pages 1884-1889.

Crochiere, R. E. and Oppenheim, A. V. Analysis of linear digital networks. *Proc. IEEE*, vol. 63, no. 4, April 1975, pages 581-595.

Cybenko, G. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, vol. 2, no. 4.

De Vries, B. and Principe, J.C. The gamma model-a new neural model for temporal processing. *Neural Networks*, vol.5, no.4, pages 565-76, 1992.

Fettweiss, A. A general theorem for signal-flow networks, with applications. In *Digital Signal Processing*, Rabiner and Rader, eds. IEEE Press, 1972, pages 126-130.

Griewank, A., and Coliss, G., eds. 1991. Automatic Differentiation of Algorithms: Theory, Implementation, and Application. *Proceedings of the first SIAM workshop on automatic differentiation*, Brekenridge, CO.

Griffiths, L. 1977. A continuously adaptive filter implemented as a lattice structure. *Proc. ICASSP*, Hartford, CT, pages 683-686.

Hertz, J. A., Krogh, A., and Palmer, R. G. 1991. *Introduction to the Theory of Neural Computation.* Addison-Wesley, Reading, PA.

Hornik, K., Stinchombe, M., and White, H. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks*, vol. 2, pages 359-366.

Jordan, M. I. and Jacobs, R. A. Hierarchies of adaptive experts. 1992. In *Advances in Neural Information Processing Systems*, vol. 4. Moody, J. E., Hanson, S. J., and Lippmann, R. P. eds. Morgan Kaufmann, pages 985-992.

Kailath, T. 1980. *Linear Systems.* Prentice-Hall, Englewood Cliffs, NJ.

Koch, C. and Segev, I, eds. 1989. *Methods in Neuronal Modeling: From Synapses to Networks.* MIT Press, Cambridge, MA.

Landau, I. *Adaptive Control: The Model Reference Approach.* Marcel Dekker, New York, 1979.

LeCun, Y., Boser, B., *et al.* 1989. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, vol. 1, pages 541-551, winter.

MacGregor, R.J. 1987, *Neural Brain Modeling*, Academic Press, Orlando, FL.

Matsuoka, K. 1991. Learning of neural networks using their adjoint systems. *Systems and Computers in Japan*, vol. 22, no. 11, pages 31-41.

Narendra, K. and Parthasarathy, K. 1990. Identification and control of dynamic

systems using neural networks. *IEEE Trans. on Neural Networks*, vol. 1, no. 1, pages 4-27.

Nerrand, 0., Roussel-Ragot, P., Personnaz, L., Dreyfus, G., and Marcos, S. 1993. Neural networks and nonlinear adaptive filtering: Unifying concepts and new algorithms. *Neural Computation*, vol. 5, no. 2, pages 165-199.

Nguyen, D., and Widrow, B. 1989. The truck backer-upper: an example of self-learning in neural networks. *Proceedings of the International Joint Conference on Neural Networks*, II, Washington, DC, pages 357-363.

Oppenheim, A., and Schafer, R. 1989. *Digital Signal Processing.* Prentice-Hall, Englewood Cliffs, NJ.

Parisini, T. and Zoppoli, R. 1994. Neural networks for feedback feedforward nonlinear control systems. *IEEE Trans. on Neural Networks*, vol. 5, no. 3, pages 436-439.

Parker, D., 1982. Learning-logic. *Invention Report S81-64, File 1, Office of Technology Licensing*, Stanford University, October.

Penfield, P., Spence, R., and Duiker, S. 1970 *Tellegen's Theorem and Electrical Networks*, MIT Press, Cambridge, MA.

Piche, S.W. Steepest descent algorithms for neural network controllers and filters, *IEEE Trans. on Neural Networks*, vol.5, no. 2, pages 198-212, 1994.

Pineda, F. J. 1987. Generalization of back-propagation to recurrent neural networks. *IEEE Trans. on Neural Networks*, special issue on recurrent networks.

Plumer, E. 1993a. Time-optimal terminal control using neural networks. *Proc. of the IEEE International Conference on Neural Networks.* San Francisco, CA, pages 1926-1931.

Plumer, E. 1993b. *Optimal Terminal Control Using Feedforward Neural Networks.* Ph.D. dissertation. Stanford University.

Puskorious, G. and Feldkamp, L. 1994. Neural control of nonlinear dynamic systems with Kalman filter trained recurrent networks. *IEEE Trans. on Neural Networks*, vol. 5, no. 2.

Rall, B. 1981. *Automatic Differentiation: Techniques and Applications*, Lecture Notes in Computer Science, Springer-Verlag.

Ramo, S., Whinnery, J.R., and Van Duzer, T. 1984. *Fields and Waves in Communication Electronics*, Second Edition. John Wiley & Sons.

Rumelhart, D.E., McClelland, J.L., and the PDP Research Group. 1986. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1. MIT Press, Cambridge, MA.

Tellegen, D. 1952. A general network theorem, with applications. *Philips Res. Rep.* 7, pages 259-269.

Toomarian, N.B. and Barhen, J. 1992. Learning a trajectory using adjoint function and teacher forcing. *Neural Networks*, vol. 5, no. 3, pages 473-484.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. 1989. Phoneme recognition using time-delay neural networks. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 37, no. 3, pages 328-339, March.

Wan, E. 1993a. *Finite Impulse Response Neural Networks with Applications in Time Series Prediction.* Ph.D. dissertation, Stanford University.

Wan, E. 1993b. Time series prediction using a connectionist network with internal delay lines. In A. Weigend and N. Gershenfeld, editors, *Time Series Prediction: Forecasting the Future and Understanding the Past*, Addison-Wesley.

Wan, E. 1993c. Modeling Nonlinear Dynamics With Neural Networks: Examples in Time Series Prediction. In *Proc. of the Fifth Workshop on Neural Networks: Academic/Industrial/NASA/Defense, WNN93/FNN93,* San Francisco, pages 327-232, November.

Wan, E., and Beaufays, F. 1996. Diagrammatic derivation of gradient algorithms for neural networks. *Neural Computation*, vol. 8, no. 1, January 1996, pages 182-201.

Werbos, P., 1974. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Ph.D. thesis, Harvard University.

Werbos, P. 1992. Neurocontrol and supervised learning: An overview and evaluation. In D. White and D. Sofge, eds., *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, chapter 3. Van Nostrand Reinhold, New York.

Werbos, P. 1990. Backpropagation through time: what it does and how to do it. *Proc. IEEE*, special issue on neural networks, vol. 2, pages 1550-1560.

White, S. A. 1975. An adaptive recursive digital filter. *Proc. 9th Asilomar Conf. Circuits Syst. Comput.*, page 21.

Williams, R.J. and Peng J., 1990. An efficient gradient-based algorithm for on line training of recurrent network trajectories. *Neural Computation*, vol. 2, pages 490-501.

Williams, R. J. and Zipser, D. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, vol. 1, no. 2, pages 270-280.

This article was processed using the LaTeX macro package with LLNCS style