

## A Signal-Flow-Graph Approach to On-line Gradient Calculation

**Paolo Campolucci\***

**Aurelio Uncini**

**Francesco Piazza**

*Dipartimento di Elettronica ed Automatica, Università di Ancona, 60121 Ancona, Italy*

A large class of nonlinear dynamic adaptive systems such as dynamic recurrent neural networks can be effectively represented by signal flow graphs (SFGs). By this method, complex systems are described as a general connection of many simple components, each of them implementing a simple one-input, one-output transformation, as in an electrical circuit. Even if graph representations are popular in the neural network community, they are often used for qualitative description rather than for rigorous representation and computational purposes. In this article, a method for both on-line and batch-backward gradient computation of a system output or cost function with respect to system parameters is derived by the SFG representation theory and its known properties. The system can be any causal, in general nonlinear and time-variant, dynamic system represented by an SFG, in particular any feedforward, time-delay, or recurrent neural network. In this work, we use discrete-time notation, but the same theory holds for the continuous-time case. The gradient is obtained in a straightforward way by the analysis of two SFGs, the original one and its adjoint (obtained from the first by simple transformations), without the complex chain rule expansions of derivatives usually employed.

This method can be used for sensitivity analysis and for learning both off-line and on-line. On-line learning is particularly important since it is required by many real applications, such as digital signal processing, system identification and control, channel equalization, and predistortion.

### 1 Introduction ---

For many practical problems such as time-series forecasting and classification, identification, and control of nonlinear dynamic systems, feedforward neural networks (NNs) are not adequate; therefore several recurrent neural network (RNN) architectures have been proposed (Tsoi & Back, 1994;

---

\* Note: Address reprint request or comments to Paolo Campolucci at [campoluc@tiscalinet.it](mailto:campoluc@tiscalinet.it).

Nerrand, Roussel-Ragot, Personnaz, Dreyfus, & Marcos, 1993; Narendra & Parthasarathy, 1991). RNNs can be essentially divided into three classes: fully recurrent NNs (FRNNs) (Williams & Peng, 1990; Williams & Zipser, 1989; Haykin, 1994), locally recurrent globally feedforward NNs (Back & Tsoi, 1991; Tsoi & Back, 1994; Campolucci, Piazza, & Uncini, 1995; Campolucci, Uncini, Piazza, & Rao, 1999), and NARX architectures (Haykin, 1994; Narendra & Parthasarathy, 1991; Horne & Giles, 1995).

In order to train an RNN adaptively, an on-line learning algorithm must be derived for each specific architecture. In many learning algorithms, the gradient of a cost function must be estimated and then used in a parameter updating scheme (e.g., steepest descent or conjugate gradient). By gradient we mean the vector of derivatives of a given variable (here a system output or a function of it, e.g., a cost function) with respect to a set of parameters that influence such a variable (here, the parameters of a subset of network components).

When RNNs (or, in general, systems with feedback) are involved, the calculation of the gradient is more complex and difficult than in the case of feedforward neural networks (or systems without feedback). Due to feedback, in fact, the current output depends on the past outputs of the RNN, so that the present output depends not only on the present parameters but also on the past parameter values, and this dependence has to be considered in calculating the gradient. Quite often in the neural network literature, this dependence has been totally or partially neglected to simplify the derivation of the gradient.

There are two different schemes to compute the gradient: the forward computation approach and the backward computation approach. These two names stem from the order in time in which the computations take place. Forward means that both time and propagation of gradient information across the network flow as in the forward pass (i.e., the computation of the network outputs from its inputs); backward indicates the opposite flow. Real-time recurrent learning (RTRL) (Williams & Zipser, 1989) and truncated backpropagation through time (truncated BPTT) (Williams & Peng, 1990) implement the forward and backward on-line computation, respectively.

The major drawback of the forward computation over the backward one is its high computational complexity (Williams & Peng, 1990; Williams & Zipser, 1994; Srinivasan, Prasad, & Rao, 1994; Nerrand et al., 1993). On the other hand, the use of the backward technique in the case of RNNs with arbitrary architectures is not straightforward; the mathematics of each particular architecture has to be worked out. For systems with feedback or with internal time delays, the chain rule derivation can easily become complex and sometimes intractable (Werbos, 1990; Williams & Zipser, 1994; Back & Tsoi, 1991; Pearlmutter, 1995; Campolucci et al., 1999).

An interesting approach to the computation of gradient information in RNNs with arbitrary architectures was proposed by Wan and Beaufays (1996), who used a diagrammatic derivation to obtain the BPTT batch al-

gorithm. The idea underlying their method is to describe the neural network by a graph model composed of some basic blocks, including summing junctions, branching points, nonlinear functions, weights, and delay operators. Then another graph is introduced to compute the gradient substituting summing junctions with branching points, and vice versa; nonlinear functions with gains; and delay operators with "advance" operators, while the weights are just replicated in the new graph (named reciprocal network).

This method considers the derivative system obtained in linearizing the original network. Unfolding the network, multiplying the network output by the error at the proper time step, and applying the interreciprocity property, it is possible to get the desired gradient information by the transposed graph corresponding to the unraveled derivative network. The reciprocal network is this transposed network raveled back in time.

The derived method can be computed only in batch mode since the reciprocal network is not causal due to the presence of the noncausal advance operators. The advance operators arise from the unfolding and transposing operations on which the method is based.

Following the graph approach, now using the signal-flow-graph (SFG) representation theory and its known properties (Mason, 1953, 1956; Lee, 1974; Oppenheim & Schaffer, 1975), in this article we propose a backward computation (BC) method that implements the BPTT algorithm for both on-line and batch-backward gradient computation of a function (e.g., a cost function) of the system output with respect to some parameters. This system can be any causal, nonlinear, and time-variant dynamic system represented by an SFG—in particular, any feedforward (static), time-delay, or RNN. In the following, we shall usually consider the NN framework, but the method is fully general. In this work, we use discrete-time notation; however, the theory holds for the continuous-time case.

The new method has been developed mainly for on-line learning. On-line training of adaptive systems is very important in real applications such as digital signal processing, system identification and control, channel equalization, and predistortion, since it allows the model to follow time-varying features of the real system without interrupting the system operation itself (such as a communication channel). It can also be used for off-line learning; in this case, when a mean-squared-error (MSE) cost function is considered, this approach gives the batch BPTT algorithm and is equivalent to the method by Wan and Beaufays (1996).

While we provide a proof based on a theorem (Lee, 1974), similar to Tellegen's theorem for analog network, the proof by Wan and Beaufays (1996) for their diagrammatic derivation is based on the interreciprocity property of transposed graphs, that is, a consequence of the theorem used here. They also propose a proof of the equivalence of RTRL and BPTT in batch mode to show that the computed weight variations are the same, though the complexities of the two algorithms are different. Beaufays and Wan (1994) do not

address the problem of graphical derivation of RTRL, as in Wan and Beaufays (1996) for BPTT; nevertheless, the two approaches have been combined for that purpose in Wan and Beaufays (1998), where a diagrammatic derivation of an on-line learning method is reported. However, the resulting RTRL algorithm has a very high computational complexity— $O(n^2)$  compared to  $O(n)$  of the on-line backward method presented here.

In this comparison, it should be considered that any on-line backward method needs a truncation of the past history; forward methods such as RTRL do not. However, it is known (Williams & Peng, 1990) that RTRL also involves a different kind of approximation due to the fact that the weights change in time during learning. To mitigate this problem, an exponential decay on the contributions from past times can be used (Gherry, 1989); nevertheless, no reduction of the computational complexity can be achieved as provided by the BPTT truncation strategy.

The work presented here also generalizes previous work by Osowski (1994), which proposed an SFG approach to neural network learning. That work was basically developed for feedforward networks, and the proposed extension to recurrent networks is valid only for networks that relax to a fixed point, as in the work of Almeida (1987); moreover, it is not possible to accommodate delay branches in this framework. General recurrent networks able to process temporal sequences cannot be trained by the proposed adjoint equations.

Another related work is that by Martinelli and Perfetti (1991), which applies an adjoint transformation to an analog real circuit implementing a multilayer perceptron (MLP) and obtaining a circuit that implements back-propagation. Since this work is more related to hardware and the network is feedforward and static, it is less general and can be considered a particular case of the method presented here.

Finally, our formulation allows dealing with both the learning problem and the sensitivity calculation problem, that is, the problem of computing the derivatives of the system output with respect to some internal parameters. These derivatives can be used for designing robust electrical or numerical circuits.

The concepts detailed in this article were developed as part of Paolo Campolucci's doctorate research and presented in detail in Campolucci (1998) and briefly introduced in (Campolucci, Marchegiani, Uncini, and Piazza (1997) and Campolucci, Uncini, and Piazza (1998).

## 2 Signal Flow Graphs and Lee's Theorem

A large class of complex systems, such as RNNs, can be represented by an SFG. Therefore in this section, SFG notation and properties will be outlined following Lee's approach (1974). A different notation and properties of SFGs were also introduced by Oppenheim and Schaffer (1975); the SFG theory was first developed by Mason (1953, 1956).

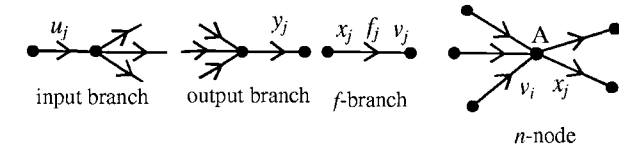


Figure 1: Signal-flow-graph definitions. If  $I_A$  is the set of indexes of branches that comes in node  $A$  and  $Z_A$  the set of indexes of branches that comes out from node  $A$ , then the following holds:  $x_j = \sum_{i \in I_A} v_i, \forall j \in Z_A$ .

An SFG is defined as a set of nodes and oriented branches. A branch is oriented from node  $a$  to node  $b$  if node  $a$  is the initial node and node  $b$  is the final node of the branch. An input node is one that has only one outgoing branch, the input branch, and no incoming branch. Associated with the input branch is an input variable  $u_j$ . An output node is one that has only one incoming branch, the output branch. Associated with the output branch is an output variable  $y_j$ . All other nodes besides the input and output nodes are called  $n$ -nodes. All other branches besides the input and output branches are called  $f$ -branches.

There are two variables related by a function associated with each of the  $f$ -branches: the initial variable  $x_j$ , at the tail of the branch, and the final variable  $v_j$ , at the head of the branch. The relationship between the initial and final variables for branch  $j$  is  $v_j = f_j[x_j]$ , where  $f_j[\cdot]$  is a general function describing the operator (or circuit component) of the  $j$ th branch. By definition, for each  $n$ -node, the value of the  $x_j$  variables associated with its outgoing branches is the sum of all the  $v_j$  variables associated with its incoming branches (see Figure 1). Let the SFG consist of  $p + m + r$  nodes ( $m$  input nodes,  $r$  output nodes, and  $p$   $n$ -nodes), and  $q + m + r$  branches ( $m$  input branches,  $r$  output branches, and  $q$   $f$ -branches). Note that only the branches (and not the nodes) are indexed; therefore, all the reported indexes are to be considered as branch indexes.

In the case of discrete-time systems, the functional relationship between the variables  $x_j$  and  $v_j$  of the  $j$ th  $f$ -branch (expressed as  $v_j = f_j[x_j]$ ) is usually assumed to be:

$$\begin{cases} v_j(t) = g_j[x_j(t), \alpha_j(t), t] & \text{for a static branch (without memory)} \end{cases} \quad (2.1)$$

$$\begin{cases} v_j(t) = \mathbf{q}^{-1} x_j(t) \triangleq x_j(t-1) & \text{for a delay branch (one unit memory),} \end{cases} \quad (2.2)$$

where  $\mathbf{q}^{-1}$  is the delay operator and  $g_j$  is a general differentiable function, which can depend on an adaptable parameter (or vector of parameters)  $\alpha_j$  and also on time  $t$  to allow shape adaptation or variation in time of the nonlinearity. For a static branch, the relationship between  $x_j$  and  $v_j$  often

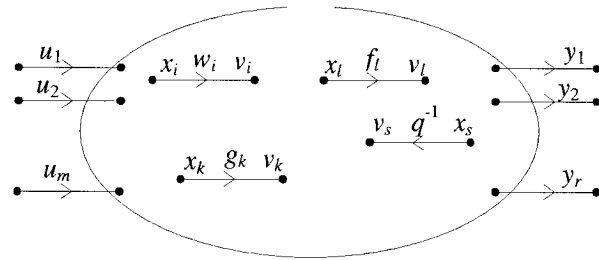


Figure 2: SFG representation of a generic system.

particularizes as

$$\begin{cases} v_j(t) = w_j(t)x_j(t) & \text{for a weight branch} \\ v_j(t) = f_j(x_j(t)) & \text{for a fixed nonlinear branch,} \end{cases} \quad (2.3)$$

where  $w_j(t)$  is the  $j$ th (i.e., belonging to the  $j$ th branch) weight parameter of the system at time  $t$ , and  $f_j$  is a differentiable function again of the  $j$ th branch. In the neural networks context,  $w_j(t)$  is a synaptic weight and  $f_j$  a sigmoidal activation function. However, equation 2.1 can represent more general nonlinear adaptable functions controlled by a parameter (or vector of parameters)  $\alpha_j$ , such as the recently introduced spline-based activation functions (Uncini, Vecci, Campolucci, & Piazza, 1999). We are now able to state the following definitions:

**Definition 1 (SFG).** The SFG  $G_N$  that represents the system is given by the connection (with or without feedback) of the branches described by equations 2.1, 2.2, and possibly 2.3 (see Figure 2).

The branches (components) are conceptually equivalent to the components of an electrical circuit, and the relationship between the SFG and the implemented system is the same as that between the electrical circuit and the system it implements. Not all the SFG can represent real systems. It is known, in fact, that in order to be computed, an SFG has to satisfy the constraint that each loop must include at least one delay branch.

**Definition 2 (reversed SFG).** A reversed SFG  $\hat{G}_N$  of a given SFG  $G_N$  is a member of the set of SFGs obtained by reversing the orientation of all the branches in  $G_N$ , that is, by replacing the summing junctions with branching points, and vice versa.

Let  $\hat{u}_j$ ,  $\hat{y}_j$ ,  $\hat{x}_j$  and  $\hat{v}_j$ , be the variables associated with the branches in  $\hat{G}_N$  corresponding to  $u_j$ ,  $y_j$ ,  $x_j$ , and  $v_j$  in  $G_N$ . In  $\hat{G}_N$  the input variables are  $\hat{y}_j$

$j = 1, \dots, r$ , the output variables are  $\hat{u}_j$   $j = 1, \dots, m$ , and  $\hat{v}_j$  and  $\hat{x}_j$  are the initial and the final variable of the  $j$ th  $f$ -branch, respectively. Let the indexes assigned to the branches of  $\hat{G}_N$  be the same as the indexes assigned to the corresponding branches of  $G_N$ . Node indexes are not required by this development. The functional relationships between the variables  $\hat{x}_j$  and  $\hat{v}_j$  in  $\hat{G}_N$  are generic; therefore, the graph  $\hat{G}_N$  effectively belongs to a set of SFGs. For an example of SFG, see Figures 3 (a-b).

**Definition 3 (transposed SFG).** The transposed SFG is a particular reversed SFG such that the relationships of the corresponding branches of  $G_N$  and  $\hat{G}_N$  remain the same.

For a generic SFG, the following theorem, similar to Tellegen's theorem for electrical network (Tellegen, 1952; Penfield, Spence, & Duiker, 1970), was derived by Lee (1974). It relies on the topological properties of the original and reversed graphs, not on the functional relationships between the branch variables, which therefore can be described by any mathematical relation.

**Theorem 2.1.** Consider a causal dynamic system (Lee, 1974). Let  $G_N$  be the corresponding SFG and  $\hat{G}_N$  a generic reversed SFG. If  $\hat{u}_j$ ,  $\hat{y}_j$ ,  $\hat{x}_j$ , and  $\hat{v}_j$ , are the variables in  $\hat{G}_N$  corresponding to  $u_j$ ,  $y_j$ ,  $x_j$ , and  $v_j$  in  $G_N$ , respectively, then

$$\sum_{j=1}^r \hat{y}_j(t) * y_j(t) + \sum_{j=1}^q \hat{x}_j(t) * x_j(t) = \sum_{j=1}^m \hat{u}_j(t) * u_j(t) + \sum_{j=1}^q \hat{v}_j(t) * v_j(t) \quad (2.4)$$

holds true, where “\*” denotes the convolution operator; that is,

$$y(t) * x(t) = \sum_{s=t_0}^t y(t-s)x(s) \quad (2.5)$$

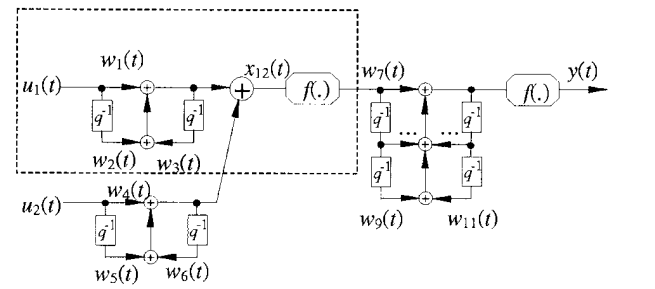
where  $t_0$  is the initial time.

For a static system the following equation holds:

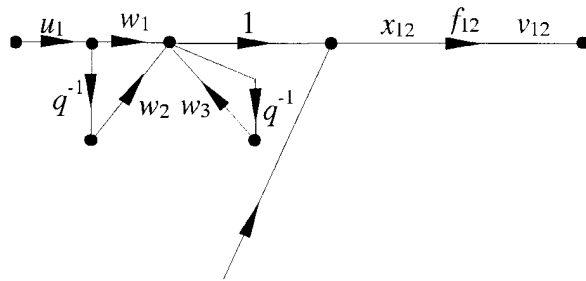
$$\sum_{j=1}^r \hat{y}_j y_j + \sum_{j=1}^q \hat{x}_j x_j = \sum_{j=1}^m \hat{u}_j u_j + \sum_{j=1}^q \hat{v}_j v_j. \quad (2.6)$$

### 3 Sensitivity Computation

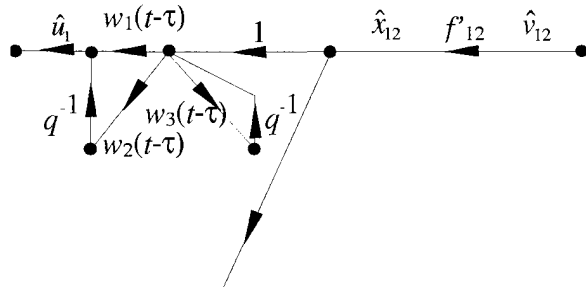
In network theory, a sensitivity is usually defined as the derivative of a certain output with respect to a given parameter. A general method to compute sensitivities of a dynamic continuous-time system represented by a SFG was



(a)



(b)



(c)

Figure 3: (a) A simple IIR-MLP neural network with two layers, two inputs, one output, and one neuron in the first layer, with two IIR filter synapses (one zero, one pole transfer function) and one neuron in the second layer, with one IIR filter synapsis (two zeros, two poles, transfer function). (b) A section of its SFG  $G_N$ . (c) Its adjoint SFG  $\hat{G}_N^{(a)}$ .

derived by Lee (1974) for constant parameters. Similarly, an analogous result can be found for a discrete-time system; however, such a derivation can be used only for sensitivity computation in batch mode. Here we present a method to compute the sensitivity with respect to time-varying parameters, that is, the derivative of a system output with respect to past or present parameters (the weights  $w_i$  and the nonlinearity control parameters  $\alpha_i$ ). For an adaptive system, they will depend on time; therefore, the derivatives with respect to such parameters will also depend on time. This is a generalization of work by Lee (1974) and Wan and Beaufays (1996).

The derivative of an output  $y_k(t)$  of the system with respect to the parameter  $w_i(t - \tau)$ , where  $t$  is the time of the original SFG and  $\tau$  a positive integer ( $t \geq 0$  and  $0 \leq \tau \leq t$ ), by the first equation in equation 2.3 is

$$\frac{\partial y_k(t)}{\partial w_i(t - \tau)} = \frac{\partial y_k(t)}{\partial v_i(t - \tau)} \frac{\partial v_i(t - \tau)}{\partial w_i(t - \tau)} = \frac{\partial y_k(t)}{\partial v_i(t - \tau)} x_i(t - \tau), \quad (3.1)$$

where  $k = 1, \dots, r$  and  $i$  spans over the set of indexes of the weight branches. For the parameters of the nonlinearity, by equation 2.1, the following more general expression holds:

$$\begin{aligned} \frac{\partial y_k(t)}{\partial \alpha_i(t - \tau)} &= \frac{\partial y_k(t)}{\partial v_i(t - \tau)} \frac{\partial v_i(t - \tau)}{\partial \alpha_i(t - \tau)} = \frac{\partial y_k(t)}{\partial v_i(t - \tau)} \frac{\partial g_i}{\partial \alpha_i} \Big|_{x_i(t - \tau), \alpha_i(t - \tau), t - \tau} \\ &\triangleq \frac{\partial y_k(t)}{\partial v_i(t - \tau)} g'_i(t - \tau), \end{aligned} \quad (3.2)$$

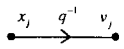
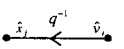
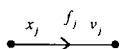
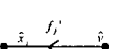
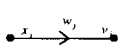
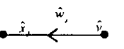
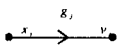
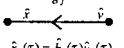
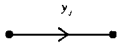
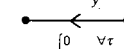
where  $k = 1, \dots, r$  and  $i$  spans over the set of indexes of the nonlinear parametric branches, and a simplified notation for the derivative of  $g_i$  is implicitly defined.

In both cases we need to compute the term  $\partial y_k(t)/\partial v_i(t - \tau)$ , which is the derivative of the  $k$ th output variable at time  $t$  with respect to a signal in the system, that is, the final variable of the  $i$ th  $f$ -branch at time  $t - \tau$ . As for the computation of the sensitivity in an electrical circuit, in this case an adjoint network can be introduced to perform this task. The main idea is to apply Lee's theorem (equation 2.4) to the SFG  $G_N$  of the original system and a particular reversed SFG  $\hat{G}_N$ , here called adjoint SFG or  $\hat{G}_N^{(a)}$ . It can be easily seen (see the proof in the appendix) that expression 2.4 holds true also when small variations of the variables are considered:

$$\begin{aligned} \sum_{j=1}^r \hat{y}_j(t) * \Delta y_j(t) + \sum_{j=1}^q \hat{x}_j(t) * \Delta x_j(t) &= \sum_{j=1}^{m+1} \hat{u}_j(t) * \Delta u_j(t) \\ &+ \sum_{j=1}^q \hat{v}_j(t) * \Delta v_j(t). \end{aligned}$$

Since the functional relationships of the branches of  $\hat{G}_N^{(a)}$  can be freely chosen, because equation 2.4 does not depend on them, these relationships are

Table 1: Adjoint SFG Construction Rules.

Branch Classification	Original Signal-Flow-Graph	Adjoint Signal-Flow-Graph for Parameter Sensitivity on $y_k$
Delay	 $v_j(t) = q^{-1}x_j(t)$ , $v_j(0) = 0$	 $\hat{x}_j(\tau) = q^{-1}\hat{v}_j(\tau)$ , $\hat{x}_j(0) = 0$
Static	 $v_j(t) = f_j(x_j(t))$	 $\hat{x}_j(\tau) = \hat{h}_j(\tau)\hat{v}_j(\tau)$ $\hat{h}_j(\tau) = f_j'(x_j(t-\tau))$
	 $v_j(t) = w_j(t)x_j(t)$	 $\hat{x}_j(\tau) = \hat{w}_j(\tau)\hat{v}_j(\tau)$ $\hat{w}_j(\tau) = w_j(t-\tau)$
	 $v_j(t) = g_j(x_j(t), \alpha_j(t), t)$	 $\hat{x}_j(\tau) = \hat{h}_j(\tau)\hat{v}_j(\tau)$ $\hat{h}_j(\tau) = \frac{\partial g_j}{\partial x_j} \Big _{x_j(t-\tau), \alpha_j(t-\tau), t-\tau}$
System Output		 $\hat{y}_j(\tau) = \begin{cases} 0 & \forall \tau \text{ if } j \neq k \\ 1, \tau = 0 & \text{if } j = k \\ 0, \tau > 0 \end{cases}$

Note: For batch adaptation, the adjoint branches must be defined, remembering that  $t = T$  in this case, where  $T$  is the final instant of the epoch.

chosen in order to obtain an estimate of  $\frac{\partial y_k(t)}{\partial v_i(t-\tau)}$  from this equation. Thus we can now state the following:

**Definition 4 (adjoint SFG).** *The adjoint SFG  $\hat{G}_N^{(a)}$  is the particular reversed graph whose  $f$ -branch relationships are related to the  $f$ -branch equations of the original graph  $G_N$  by the correspondence reported in Table 1.*

Table 1 shows that for each delay branch in the original SFG, the corresponding branch of the adjoint SFG is a delay branch (with zero initial condition). The nonlinearity without parameters  $f_j$  in the original SFG corresponds to a gain equal to the derivative of  $f_j$  evaluated at the value of the initial variable of the branch at time  $t - \tau$  in the adjoint SFG, where  $t$  is the time of the original SFG and  $\tau$  is that of the adjoint SFG. The weight at time  $t$  corresponds to the weight at time  $t - \tau$  in the adjoint SFG. A similar

transformation should be performed for the general nonlinearity with control parameters. The outputs of the original SFG correspond to the inputs of the adjoint SFG. For an example of adjoint SFG construction for a simple MLP with infinite impulse response (IIR) filter synapses, see Figure 3.

Moreover, the inputs of the adjoint SFG must be set to an impulse in correspondence with the output of  $G_N$  of which the sensitivity has to be computed, and to a constant null signal in correspondence with all the other outputs. In other words, to compute the value of  $\partial y_k(t)/\partial v_i(t - \tau)$ , the input of the adjoint SFG should be:

$$\hat{y}_j(\tau) = \begin{cases} 0 & \forall \tau \text{ if } j \neq k \\ 1, \tau = 0 & \text{if } j = k \\ 0, \tau > 0 \end{cases} \quad j = 1, \dots, r. \quad (3.3)$$

Let  $\hat{y}_j$ ,  $\hat{x}_j$ , and  $\hat{v}_j$  be the variables associated with the branches in  $\hat{G}_N^{(a)}$  corresponding to  $u_j$ ,  $y_j$ ,  $x_j$ , and  $v_j$  in  $G_N$ ; it follows (see the proof in the appendix) that

$$\frac{\partial y_k(t)}{\partial v_i}(t - \tau) = \hat{v}_i(\tau), \quad (3.4)$$

where  $k = 1, \dots, r$  and  $i = 1, \dots, q$ . Using this result in equation 3.1, we get

$$\frac{\partial y_k(t)}{\partial w_i(t - \tau)} = \hat{v}_i(\tau)x_i(t - \tau). \quad (3.5)$$

This result states that the derivative of an output variable of the SFG at time  $t$  with respect to its weight parameter  $w_i(t - \tau)$  at time  $t - \tau$  is the product of the initial variable of the  $i$ th branch in the original SFG at time  $t - \tau$  and the initial variable of the corresponding branch in the adjoint graph at  $\tau$  time units. This result can be easily generalized for the nonlinear parametric function as follows:

$$\frac{\partial y_k(t)}{\partial \alpha_i(t - \tau)} = \hat{v}_i(\tau)g_i'(t - \tau). \quad (3.6)$$

#### 4 SFG Approach to Learning in Nonlinear Dynamic Systems

In the previous section, we showed how the SFG of the original network  $G_N$  and the SFG of the adjoint network  $\hat{G}_N^{(a)}$  can be used to compute the sensitivity of an output with respect to a past or present system parameter (equations 3.5 and 3.6).

Now we show how to use this technique to adapt a dynamic network, minimizing a supervised cost function with respect to the network parameters. The idea is to consider the cost function itself as a network connected

in cascade to the dynamical network to be adapted. Therefore, the entire system (named  $G_S$  in the following) has as inputs the inputs of the original network and the desired outputs, while the value of the cost function is its unique output. The gradient of this output (i.e., of the cost function) with respect to the system parameters (e.g., weights) now corresponds to the sensitivity of this output, which can be computed by using equations 3.5 and 3.6 again, applied now to the cascade system  $G_S$ .

**4.1 Cost Functions and Parameter Updating Rules.** Let us consider a discrete-time nonlinear dynamic system with inputs  $u_k$ ,  $k = 1, \dots, m$ , outputs  $y_k$ ,  $k = 1, \dots, r$ , and parameters  $w_i$  and  $\alpha_i$ , which have to be adapted with respect to an output error. Using gradient-based learning and following the steepest-descent updating rule, it holds, for example, for the weight  $w_i$

$$\Delta w_i = -\mu \frac{\partial J}{\partial w_i}, \quad \mu > 0, \quad (4.1)$$

where  $J$  is the cost function,  $\Delta w_i$  is the variation of the parameter  $w_i$ , and  $\mu$  is the learning rate. The major problem is given by the calculation of the derivative  $\partial J / \partial w_i$ .

Since for on-line learning the parameters can change at each time instant,

$$\frac{\partial J}{\partial w_i} = \sum_{\tau=0}^t \frac{\partial J}{\partial w_i(\tau)} = \sum_{\tau=0}^t \frac{\partial J}{\partial w_i(t-\tau)}. \quad (4.2)$$

As the length of the summation linearly increases with the current time step  $t$ , the summation in equation 4.2 must be truncated in order to implement the algorithm,

$$\Delta w_i(t) = -\mu \sum_{\tau=t-h+1}^t \frac{\partial J}{\partial w_i(\tau)} = -\mu \sum_{\tau=0}^{h-1} \frac{\partial J}{\partial w_i(t-\tau)}, \quad (4.3)$$

where  $h$  is a fixed positive integer. In this way, not all the history of the system is considered but only the most recent part in the interval  $[t-h+1, t]$ . It is easy to show that a real truncation is necessary only for circuits with feedback, such as RNN, while for feedforward networks with delays (e.g., time delay NN, or TDNN) a finite  $h$  can be chosen, so that all the memory of the system is taken into account. An optimal selection of  $h$  requires an appropriate choice for each parameter to be adopted. For layered TDNN,  $h$  should depend on the layer and should be increased moving from the last to the first layer, since more memory is involved.

The updating rule (see equations 4.2 and 4.3) holds theoretically in the hypothesis of constant weights, but practically it is only a good approximation. Equations 3.5 and 3.6 do not require that hypothesis and can be used

in a more general context. Equations equivalent to 4.1, 4.2, and 4.3 can also be written for the parameters  $\alpha_i$ .

The most common choice for the cost function  $J$  is the MSE. The instantaneous squared error at time  $t$  is defined as

$$e^2(t) = \sum_{k=1}^r e_k^2(t) \quad \text{with } e_k(t) = d_k(t) - y_k(t), \quad (4.4)$$

where  $d_k(t)$   $k = 1, \dots, r$  are the desired outputs. So the MSE over a time interval  $[t_0, t_1]$  is given by

$$E(t_0, t_1) = \sum_{t=t_0}^{t_1} e^2(t). \quad (4.5)$$

In the case of batch training, the cost function can be chosen as the MSE over the entire learning epoch  $E(0, T)$ , where  $T$  is the final time of the epoch, whereas for on-line training only the most recent errors  $e^2(t)$  must be considered, for example, using  $E(t-N_c+1, t)$  with the constant  $N_c$  properly chosen (Nerrand et al., 1993; Narendra & Parthasarathy, 1991). Therefore it holds:

$$\begin{cases} E(t-N_c+1, t) = \sum_{s=t-N_c+1}^t e^2(s) & \text{on-line training} \\ E(0, T) = \sum_{s=0}^T e^2(s) & \text{batch training.} \end{cases}$$

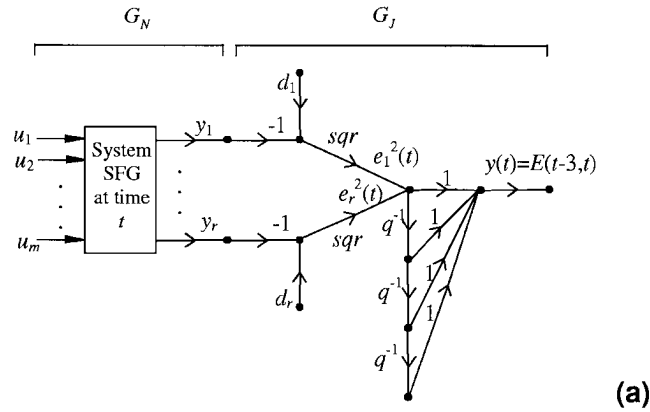
Assuming  $J = E(t_0, t_1)$ , it follows that

$$\Delta w_i(t) = -\mu \sum_{\tau=0}^{h-1} \frac{\partial E(t-N_c+1, t)}{\partial w_i(t-\tau)} \quad \text{for on-line training} \quad (4.6a)$$

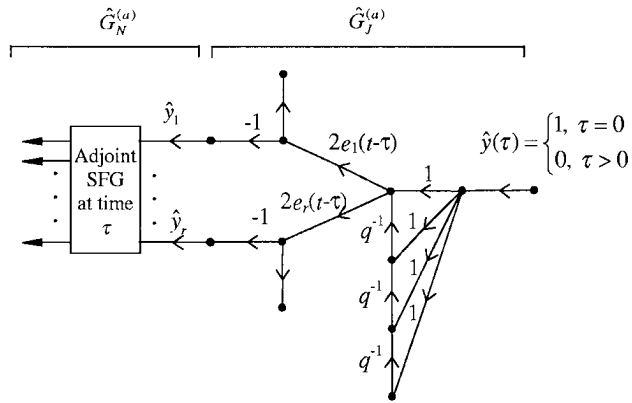
and

$$\Delta w_i = -\mu \sum_{\tau=0}^T \frac{\partial E(0, T)}{\partial w_i(T-\tau)} \quad \text{for batch training.} \quad (4.6b)$$

**4.2 Cost Function Gradient Computation by SFGs.** Let us consider the SFG  $G_S$  obtained by a cascade connection of the SFG  $G_N$  of the system to be adapted and the SFG that implements the cost function  $J$  (named  $G_J$  in the following) (see Figure 4 for an example). The SFG  $G_J$  has as inputs the outputs of the adaptable system  $y_k$  and the targets  $d_k$ , while the value of  $J$  is its unique output. Obviously  $G_J$  can contain the same kind of operators as  $G_N$ : delays, nonlinearities, parameters to be adapted (e.g., for regulariza-



(a)



(b)

Figure 4: (a) SFG  $G_S$ . The system SFG  $G_N$  in cascade with the error calculation SFG  $G_J$  with  $J = E(t - 3, t)$ .  $sqr(x)$  means  $x^2$ , and  $d_j$   $j = 1, \dots, r$  are the desired outputs. (b) Its adjoint  $\hat{G}_S^{(a)}$ , that is,  $\hat{G}_J^{(a)}$  in cascade with  $\hat{G}_N^{(a)}$ .

tion purposes), and feedback. The class of cost functions allowed by this approach is therefore enlarged with respect to other approaches since the cost expression can have memory and be recursive. Although some applications would require very complex cost functions, the SFG approach can easily handle them.

Using the adjoint network  $\hat{G}_S^{(a)}$  of the new SFG  $G_S$ , it is easy to calculate the sensitivities of the output with respect to the system parameters (see equations 3.5 and 3.6). Since the output  $y_k$  ( $k = 1$ ) of  $G_S$  is the cost function  $J$ , combining equation 4.2 with 3.5 or 3.6 and truncating the past history to  $h$  steps (on-line learning) it holds:

$$\left\{ \begin{array}{l} \frac{\partial J(t)}{\partial w_i} = \sum_{\tau=0}^{h-1} \hat{v}_i(\tau) x_i(t - \tau) \quad \text{on-line} \end{array} \right. \quad (4.7a)$$

$$\left\{ \begin{array}{l} \frac{\partial J}{\partial w_i} = \sum_{\tau=0}^T \hat{v}_i(\tau) x_i(T - \tau) \quad \text{batch} \end{array} \right. \quad (4.7b)$$

$$\left\{ \begin{array}{l} \frac{\partial J(t)}{\partial \alpha_i} = \sum_{\tau=0}^{h-1} \hat{v}_i(\tau) g'_i(t - \tau) \quad \text{on-line} \end{array} \right. \quad (4.8a)$$

$$\left\{ \begin{array}{l} \frac{\partial J}{\partial \alpha_i} = \sum_{\tau=0}^T \hat{v}_i(\tau) g'_i(T - \tau) \quad \text{batch.} \end{array} \right. \quad (4.8b)$$

Equations 4.7 and 4.8 hold for any cost function  $J$  that can be described by an SFG. In the particular case when  $J$  is chosen to be a standard MSE cost function, we show that the explicit implementation of  $G_J$  can be avoided since the involved derivatives can be analytically derived.

For on-line learning, using the instantaneous squared error  $J = e^2(t)$  the input-output relationship of  $G_J$  is given by equation 4.4; thus,

$$\frac{\partial e^2(t)}{\partial y_k(t - \tau)} = \begin{cases} -2e_k(t), & \tau = 0 \\ 0, & \tau > 0 \end{cases}, \quad k = 1, \dots, r \quad (4.9)$$

Using equations 3.4 and 4.9 and considering the output  $y_k$  of  $G_N$  as an internal signal  $v_i$ , we can explicitly compute the outputs of the adjoint network  $\hat{G}_J^{(a)}$

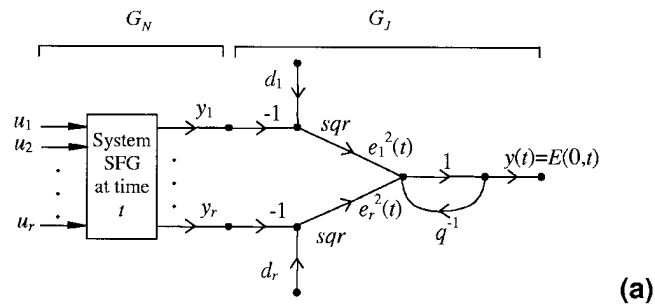
$$\hat{y}_k(\tau) = \frac{\partial e^2(t)}{\partial y_k(t - \tau)} = \begin{cases} -2e_k(t), & \tau = 0 \\ 0, & \tau > 0 \end{cases}, \quad k = 1, \dots, r \quad (4.10)$$

Such outputs can be used as the inputs of  $\hat{G}_N^{(a)}$ . Therefore if the signal, equation 4.10, feeds the network  $\hat{G}_N^{(a)}$  instead of equation 3.3, the explicit use of  $\hat{G}_J^{(a)}$  can be avoided.

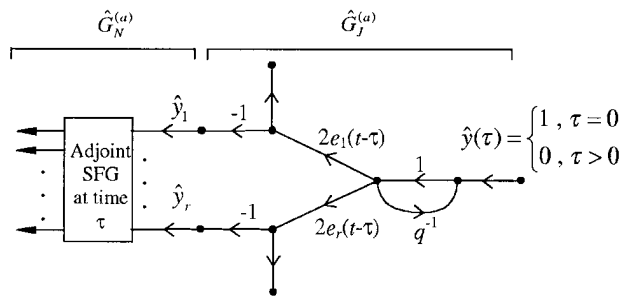
If  $J = E(t - N_c + 1, t)$  is used, then equation 4.10 generalizes to the following (see Figure 4):

$$\begin{aligned} \hat{y}_k(\tau) &= \frac{\partial E(t - N_c + 1, t)}{\partial y_k(t - \tau)} \\ &= \begin{cases} -2e_k(t - \tau), & \tau \leq N_c - 1 \\ 0, & \tau \geq N_c \end{cases}, \quad k = 1, \dots, r \end{aligned} \quad (4.11)$$





(a)



(b)

Figure 5: (a) SFG  $G_S$ , that is, the system SFG  $G_N$  in cascade with the error calculation SFG  $G_J$  with  $J = E(0, t)$ ,  $sqr(x)$  means  $x^2$ ,  $d_j$ ,  $j = 1, \dots, r$  are the desired outputs; (b) its adjoint  $\hat{G}_S^{(a)}$ , that is,  $\hat{G}_J^{(a)}$  in cascade with  $\hat{G}_N^{(a)}$ . The feedback in (a) sums the error terms from the initial instant. The feedback in (b) injects into the SFG's left side a constant value equal to 1 for all  $\tau$ .

When batch learning is involved, that is, when  $J = E(0, T)$ , the following equation has to be used instead of 4.10 or 4.11:

$$\hat{y}_k(\tau) = \frac{\partial E(0, T)}{\partial y_k(T - \tau)} = -2e_k(T - \tau), \quad k=1, \dots, r, \quad \tau=0, 1, \dots, T. \quad (4.12)$$

Note that the parameter variations computed by the summation of all gradient terms in  $[0, T]$  are the same as those obtained by BPTT (Wan & Beaufays, 1996) (see Figure 5).

Thus, for standard MSE cost functions, the derivative calculation can be performed considering  $G_N$  and  $\hat{G}_N^{(a)}$  only, but using either equation 4.10, 4.11, or 4.12 as inputs for  $\hat{G}_N^{(a)}$  instead of equation 3.3.

Srinivasan et al. (1994) derive a theorem that substantially states what is expressed in equation 4.7 when  $J = e^2(t)$ ; however, such a theorem has been proved only for a neural network with a particular structure and by a completely different and very specific approach, whereas the method presented here is very general, including any NN as a special case.

In the following, we call the resulting algorithm BC( $h$ ) (backward computation), where  $h$  is the truncation parameter, or BC for a short notation or the batch case.

**4.3 Detailed Steps of the BC Procedure.** To derive the BC algorithm for training an arbitrary SFG  $G_N$  using an MSE cost function, the adjoint SFG  $\hat{G}_N^{(a)}$  must be drawn by reversing the graph and applying the transformation of Table 1.

*On-Line Learning.* Choosing  $J = E(t - N_c + 1, t)$ , then the following steps have to be performed for each time instant  $t$ :

1. The system SFG  $G_N$  is computed one time step forward, storing in memory the internal states for the last  $h$  time steps.
2. The adjoint SFG  $\hat{G}_N^{(a)}$  is reset (setting null initial conditions for the delays).
3.  $\hat{G}_N^{(a)}$  is computed for  $\tau = 0, 1, \dots, h - 1$  with the input given by equation 4.11 computing the terms of summations 4.7a and 4.8a.
4. The parameters  $w_j$  and  $\alpha_j$  are adapted by equation 4.1.

*Batch Learning.* Choosing  $J = \sum_{t=0}^T e^2(t)$ , then the procedure is simpler. For each epoch:

1. The system SFG  $G_N$  is computed, storing its internal states from time  $t = 0$  up to time  $T$ .
2. The adjoint SFG  $\hat{G}_N^{(a)}$  is reset.
3.  $\hat{G}_N^{(a)}$  is evaluated for  $\tau = 0$  up to  $T$ , with  $t = T$  (see the appendix, accumulating the terms needed by equations 4.7b and 4.8b).
4. The parameters  $w_j$  and  $\alpha_j$  are adapted by equation 4.1.

However, not all the variables of the adjoint SFG have to be computed—only the initial variables of the branches containing adaptable parameters since they must be used in equations 4.7 and 4.8. It must be stressed that the delay operator of the adjoint SFG delays the  $\tau$  and not the  $t$  index.

When the MSE is considered, the batch mode BC learning method corresponds to batch BPTT and Wan and Beaufays' method (1996), while the

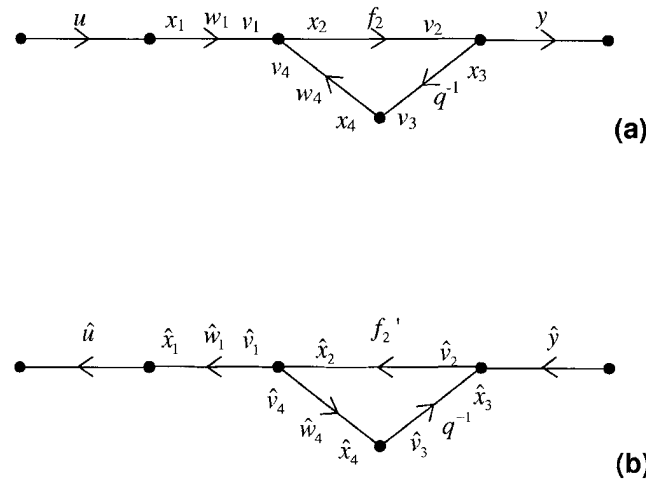


Figure 6: (a) SFG representing a recurrent neuron. (b) Its adjoint.

on-line mode BC procedure corresponds to truncated BPTT (Williams & Peng, 1990).

If the desired cost function  $J$  is not a conventional MSE, then the cost function SFG, giving the overall SFG  $G_S$ , whose adjoint SFG  $\hat{G}_S^{(a)}$  must be drawn by reversing the graph and applying the transformation of Table 1. In the previous steps of the BC procedures, now  $G_S$  must be considered instead of  $G_N$  and  $\hat{G}_S^{(a)}$  instead of  $\hat{G}_N^{(a)}$  while the input of the adjoint SFG  $\hat{G}_S^{(a)}$  is given by equation 3.3 instead of 4.10, 4.11, or 4.12.

**4.4 Example of Application of the BC Algorithm.** As an example, a very simple SFG, a recurrent neuron with two adaptable parameters, is considered (see Figure 6a). The equations of the BC algorithm will be detailed to make the method clear and to show the relationship with truncated BPTT( $h$ ) (Williams & Peng, 1990) in on-line mode and with BPTT in batch mode.

The equations of the forward phase of the system can be written looking at Figure 6a. Let us assume the following order of the branches: for the two weights and the nonlinearity operator, the branch index is the subindex shown in the figure, and for the delay operator it is equal to three. Since the system is single-input, single-output (SISO), the  $u$  and  $y$  variables do not

need any index:

$$s(t) \triangleq x_2(t) = w_1(t)u(t) + w_4(t)y(t-1) \quad (4.13)$$

$$y(t) = f_2(s(t)). \quad (4.14)$$

**4.4.1 On-line Case.** For the simplest case  $J = e^2(t) = (d(t) - y(t))^2$ , according to equation 4.10, the input of the corresponding adjoint SFG (see Figure 6b) must be:

$$\hat{y}(\tau) = \begin{cases} -2e(t), & \tau = 0 \\ 0, & \tau > 0. \end{cases} \quad (4.15)$$

From the adjoint SFG, it results in

$$\begin{aligned} \hat{v}_1(\tau) &= \hat{v}_4(\tau) \\ &= f_2'(s(t-\tau)) \left( \hat{y}(\tau) + \begin{cases} \hat{w}_4(\tau-1)\hat{v}_4(\tau-1), & \tau > 0 \\ 0, & \tau = 0 \end{cases} \right), \end{aligned} \quad (4.16)$$

where (see Table 1)

$$\hat{w}_4(\tau-1) = w_4(t-\tau+1). \quad (4.17)$$

According to equations 3.5 and 4.3, the weights can be updated using the following two equations:

$$\Delta w_1(t) = \mu \sum_{\tau=0}^{h-1} \delta(t-\tau)u(t-\tau) \quad (4.18)$$

$$\Delta w_4(t) = \mu \sum_{\tau=0}^{h-1} \delta(t-\tau)y(t-\tau-1), \quad (4.19)$$

where

$$\delta(t-\tau) \triangleq -\frac{\partial e^2(t)}{\partial s(t-\tau)} = -\frac{\partial e^2(t)}{\partial v_1(t-\tau)} = -\hat{v}_1(\tau). \quad (4.20)$$

It can be easily seen that these equations do indeed correspond to truncated BPTT( $h$ ), since the quantity  $\delta$  is simply the usual  $\delta$  of truncated BPTT. Note that here a weight buffer has to be used as stated by the theory, while sometimes in the literature only the current weights are used to obtain a simpler approximated implementation (Williams & Peng, 1990).

It is worth noting that in spite of the simplicity of the system (a single recurrent neuron), the backward equations are not easy to derive by chain rule; in fact, forward (recursive) equations are usually proposed for adaptation.

**4.4.2 Batch Case.** The graph is computed for  $t = 0, \dots, T$  saving the internal states, without any evaluation of the adjoint SFG. The weight time index can be neglected since the weights are now constant.

In this case, the MSE over the entire epoch is taken,  $J = \sum_{t=0}^T e^2(t)$ ; therefore, the input of the adjoint SFG (see Figure 6b) must be, according to equation 4.12:

$$\hat{y}(\tau) = -2e(T - \tau), \quad \tau = 0, 1, \dots, T \quad (4.21)$$

From the adjoint SFG, remembering that  $t = T$ , it results in

$$\hat{v}_1(\tau) = \hat{v}_4(\tau) = f_2'(s(T - \tau)) \left( \hat{y}(\tau) + \begin{cases} \hat{w}_4 \hat{v}_4(\tau - 1), & \tau > 0 \\ 0, & \tau = 0 \end{cases} \right), \quad (4.22)$$

where  $\hat{w}_4 = w_4$ .

Therefore the weights can be updated according to the following two equations:

$$\Delta w_1 = \mu \sum_{\tau=0}^T \delta(T - \tau) u(T - \tau) \quad (4.23)$$

$$\Delta w_4 = \mu \sum_{\tau=0}^T \delta(T - \tau) y(T - \tau - 1) \quad (4.24)$$

where

$$\delta(T - \tau) \triangleq -\frac{\partial J}{\partial s(T - \tau)} = -\frac{\partial J}{\partial v_1(T - \tau)} = -\hat{v}_1(\tau). \quad (4.25)$$

It is easy to see that these equations correspond to BPTT, since the quantity  $\delta$  is simply the usual  $\delta$  of BPTT.

**4.5 Complexity Analysis.** The complexity of the proposed gradient computation is very low, since it linearly increases with the number of adaptable parameters. In particular, since the adjoint graph has the same topology as the original one, the complexity of its computation is about the same; in practice, it is lower, since not all the variables of the adjoint graph have to be computed, as stated in section 4.3. For on-line learning, the adjoint graph must be evaluated  $h$  times for each time step; therefore, the number of operations for the computation of the gradient terms with respect to all the parameters is roughly (in practice it is lower)  $h$  times the number of operations of the forward phase, plus the computation needed by equations 4.7 and 4.8 for each time step (whose complexity linearly increases with  $h$  and the number of parameters).

**4.5.1 Fully Recurrent Neural Networks.** For the simple case of a fully recurrent, single-layer, single-delay neural network composed of  $n$  neurons, the computational complexity is  $O(n^2)$  operations per time step for batch BC or  $O(n^2 h)$  for on-line BC compared with  $O(n^4)$  for RTRL (Williams and Zipser, 1989). The memory requirement is  $O(nT)$  for batch BC or  $O(nh)$  for on-line BC, and  $O(n^3)$  for RTRL. Therefore, as far as computational complexity is concerned, in batch mode, BC is significantly simpler than RTRL, whereas in on-line mode, the complexity and also the memory requirement ratios depend on  $n^2/h$ . However in many practical cases,  $n^2$  is large compared to  $h$ , and therefore RTRL will be more complex than on-line BC.

**4.5.2 Locally Recurrent Neural Networks.** For a complex architecture such as a locally recurrent layered network, a mathematical evaluation of complexity can be carried out by computing the number of multiplications and additions for one iteration of the on-line learning (i.e., for each input sample). Results for on-line BC are reported in the significant special case of a two-layer MLP with IIR temporal filter synapses (IIR-MLP) (Tsoi & Back, 1994; Campolucci et al., 1999) with bias and moving average (MA) and autoregressive (AR) orders ( $L^{(l)}$  and  $I^{(l)}$ , respectively), depending only on the layer index  $l$ .

The number of additions and multiplications is, respectively:

$$N_2 + h \left[ N_1 N_0 (2I^{(1)} + L^{(1)}) + 2N_2 N_1 (I^{(2)} + L^{(2)}) - N_2 (N_1 - 1) + N_1 \right],$$

$$N_2 + h \left[ N_1 N_0 (2I^{(1)} + L^{(1)} + 1) + 2N_2 N_1 (I^{(2)} + L^{(2)}) + N_1 (N_2 + 1) \right]$$

where  $N_i$  is the number of neurons of layer  $i$ th ( $i = 0$  corresponds to the input layer). These numbers must be added to the number of operations of the forward phase, which should always be performed before the backward phase,  $N_1 N_0 (L^{(1)} + I^{(1)}) + N_2 N_1 (L^{(2)} + I^{(2)})$ , that is, the same for both addition and multiplication.

## 5 Conclusion

We have presented a signal-flow-graph approach that allows an easy on-line computation of the gradient terms needed in sensitivity analysis and learning of nonlinear dynamic adaptive systems represented by SFG, even if their structure includes feedback (of any kind, even nested) or time delays. This method bypasses the complex chain rule derivative expansion traditionally needed to derive the gradient equations.

The gradient information obtained in this way can be useful for circuit optimization by output sensitivity minimization or for gradient-based training algorithms, such as conjugate gradient or other techniques. This method should allow the development of computer-aided design software by which

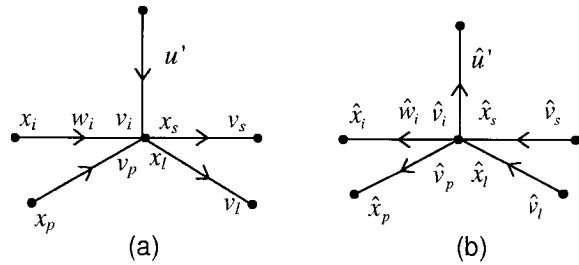


Figure 7: (a) The perturbation introduced in the original SFG. (b) The corresponding node in the adjoint SFG.

an operator could define an architecture of a system to be adapted or a circuit whose sensitivity must be computed, leaving the software the hard task of finding and implementing the needed gradient calculation algorithm. We have easily developed this software in a very general version.

This work is related to previous results in different fields and provides an elegant analogy with the sensitivity analysis of analog networks by the adjoint network technique obtained applying Tellegen's theorem which in that case relates voltages and currents of the network (Director & Rohrer, 1969).

#### Appendix: Proof of the Adjoint SFG Method

Let us consider a nonlinear dynamical system described by an SFG with the notation defined in section 2. Starting at time 0 and letting the system run up to time  $t$  we obtain the signals  $u_j(t)$ ,  $y_j(t)$ ,  $x_j(t)$ , and  $v_j(t)$ .

Now let us repeat the same process but introducing at time  $t - \tau$  a perturbation  $\varepsilon$  to a specific node of the graph (see Figure 7a). This is equivalent to considering a system with  $m + 1$  inputs, one more than the original system, where the input  $(m + 1)$ th is  $u_{m+1}(t) = u'(t)$  where  $u'(t)$  is defined such as

$$u'(t - \phi) = \begin{cases} 0, & \phi \neq \tau \\ \varepsilon, & \phi = \tau \end{cases}.$$

Therefore, at time  $t$ , all the variables  $u_j(t)$ ,  $y_j(t)$ ,  $x_j(t)$ , and  $v_j(t)$  will be changed to  $u_j(t) + \Delta u_j(t)$ ,  $y_j(t) + \Delta y_j(t)$ ,  $x_j(t) + \Delta x_j(t)$ , and  $v_j(t) + \Delta v_j(t)$ . Obviously it holds that  $\Delta u'(t) = u'(t)$ ,  $\forall t$ .

Let  $\hat{u}_j$ ,  $\hat{y}_j$ ,  $\hat{x}_j$ ,  $\hat{v}_j$ , and  $\hat{u}_{m+1}(t) = \hat{u}'(t)$  be the corresponding variables associated with the branches of a generic reversed SFG  $\hat{G}_N$  of the original graph

$G_N$ ; applying theorem 1 with  $\varepsilon = 0$  (no perturbation), it follows that

$$\sum_{j=1}^r \hat{y}_j(t) * y_j(t) + \sum_{j=1}^q \hat{x}_j(t) * x_j(t) = \sum_{j=1}^{m+1} \hat{u}_j(t) * u_j(t) + \sum_{j=1}^q \hat{v}_j(t) * v_j(t), \quad (\text{A.1})$$

while when  $\varepsilon \neq 0$  (with perturbation), the same equation rewrites as

$$\begin{aligned} \sum_{j=1}^r \hat{y}_j(t) * (y_j(t) + \Delta y_j(t)) + \sum_{j=1}^q \hat{x}_j(t) * (x_j(t) + \Delta x_j(t)) \\ = \sum_{j=1}^{m+1} \hat{u}_j(t) * (u_j(t) + \Delta u_j(t)) + \sum_{j=1}^q \hat{v}_j(t) * (v_j(t) + \Delta v_j(t)). \end{aligned} \quad (\text{A.2})$$

Since the convolution is a linear operator, subtracting equation A.1 from A.2 results in

$$\begin{aligned} \sum_{j=1}^r \hat{y}_j(t) * \Delta y_j(t) + \sum_{j=1}^q \hat{x}_j(t) * \Delta x_j(t) = \sum_{j=1}^{m+1} \hat{u}_j(t) * \Delta u_j(t) \\ + \sum_{j=1}^q \hat{v}_j(t) * \Delta v_j(t). \end{aligned} \quad (\text{A.3})$$

With regard to the input branches  $\hat{y}_j$  of the SFG  $\hat{G}_N$ , it is possible to choose

$$\hat{y}_j(\tau) = \begin{cases} 0 & \forall \tau \quad \text{if } j \neq k \\ 1, & \tau = 0 \\ 0, & \tau > 0 \end{cases} \quad \text{if } j = k \quad j = 1, \dots, r \quad (\text{A.4})$$

in order to obtain:

$$\sum_{j=1}^r \hat{y}_j(t) * \Delta y_j(t) = \Delta y_k(t), \quad (\text{A.5})$$

where  $y_k$  is the output of which we need to compute the gradient.

Since the inputs  $u_j$ ,  $j = 1, \dots, m$  of  $G_N$  are obviously not dependent on the perturbation  $\varepsilon$ , it holds that

$$\Delta u_j(t) = 0 \quad \forall t, \quad j = 1, \dots, m; \quad (\text{A.6})$$

therefore,

$$\begin{aligned} \sum_{j=1}^{m+1} \hat{u}_j(t) * \Delta u_j(t) &= \hat{u}_{m+1}(t) * \Delta u_{m+1}(t) = \hat{u}'(t) * \Delta u'(t) \\ &= \sum_{\phi=0}^t \hat{u}'(\phi) \Delta u'(t-\phi) = \hat{u}'(\tau) \Delta u'(t-\tau). \end{aligned} \quad (\text{A.7})$$

Now we use the degrees of freedom given by the definition of reversed graph to choose the  $f$ -branch operators in  $\hat{G}_N$  such that it holds:

$$\hat{v}_j(t) * \Delta v_j(t) = \hat{x}_j(t) * \Delta x_j(t) \quad j = 1, \dots, q, \quad (\text{A.8})$$

and equation A.3 can be greatly simplified. In this way, the adjoint graph  $\hat{G}_N^{(a)}$  will be obtained.

For delay branches:

$$v_j(t) = \mathbf{q}^{-1} x_j(t) \Rightarrow \Delta v_j(t) = \Delta x_j(t-1) = \mathbf{q}^{-1} \Delta x_j(t). \quad (\text{A.9})$$

Let us choose the relationship between the branch variables in  $\hat{G}_N$  as

$$\hat{x}_j(\tau) = \mathbf{q}^{-1} \hat{v}_j(\tau), \quad (\text{A.10})$$

where  $\tau$  is the time index of the adjoint SFG and  $t$  the time index of the original SFG; the same notation will be used in the following.

Since  $\Delta x_j(t-\phi)$  and  $\Delta v_j(t-\phi)$  are zero, if  $\phi > \tau$ , since the system is causal, and the initial conditions of  $\hat{G}_N$  are set to zero, that is

$$\hat{x}_j(0) = 0, \quad (\text{A.11})$$

it follows that

$$\begin{aligned} \hat{v}_j(t) * \Delta v_j(t) &= \sum_{\phi=0}^t \hat{v}_j(\phi) \Delta v_j(t-\phi) = \sum_{\phi=0}^{\tau} \hat{v}_j(\phi) \Delta v_j(t-\phi) \\ &= \sum_{\phi=0}^{\tau} \hat{x}_j(\phi+1) \Delta x_j(t-\phi-1) = \sum_{s=1}^{\tau+1} \hat{x}_j(s) \Delta x_j(t-s) \\ &= \sum_{s=0}^{\tau+1} \hat{x}_j(s) \Delta x_j(t-s) = \sum_{s=0}^{\tau} \hat{x}_j(s) \Delta x_j(t-s) \\ &= \hat{x}_j(t) * \Delta x_j(t). \end{aligned} \quad (\text{A.12})$$

For static branches:

$$v_j(t) = g_j(x_j(t), \alpha_j(t), t). \quad (\text{A.13})$$

Let us choose the relationship between the branch variables  $\hat{x}_j$  and  $\hat{v}_j$  in  $G_N$  as

$$\hat{x}_j(\tau) = \hat{v}_j(\tau) \left. \frac{\partial g_j}{\partial x_j} \right|_{x_j(t-\tau), \alpha_j(t-\tau), t-\tau} \triangleq \hat{v}_j(\tau) g'_j(t-\tau), \quad (\text{A.14})$$

where  $g'_j(\cdot)$  is implicitly defined as in equation 3.2. By differentiating

$$\Delta v_j(s) = \begin{cases} (\partial g_j / \partial x_j) |_{x_j(s), \alpha_j(s), s} \Delta x_j(s) = g'_j(s) \Delta x_j(s), & s \geq t-\tau \\ 0, & s < t-\tau \end{cases} \quad (\text{A.15})$$

$$\begin{aligned} \hat{v}_j(t) * \Delta v_j(t) &= \sum_{\phi=0}^t \hat{v}_j(\phi) \Delta v_j(t-\phi) \\ &= \sum_{\phi=0}^{\tau} \hat{v}_j(\phi) g'_j(t-\phi) \Delta x_j(t-\phi) = \hat{x}_j(t) * \Delta x_j(t). \end{aligned} \quad (\text{A.16})$$

Very interesting particular cases of equation A.14 for nonlinear adaptive filters or neural networks are:

$$\begin{cases} \hat{x}_j(\tau) = w_j(t-\tau), \hat{v}_j(\tau) & \text{for a weight branch} \\ \hat{x}_j(\tau) = f'_j(x_j(t-\tau)) \hat{v}_j(\tau) & \text{for a nonlinear branch.} \end{cases} \quad (\text{A.17})$$

Equations A.17 and A.18 correspond to the branches defined in equation 2.3, respectively.

We have completely defined the reversed SFG  $\hat{G}_N$ , which is now called *adjoint signal-flow-graph*  $\hat{G}_N^{(a)}$ . Therefore the adjoint SFG  $\hat{G}_N^{(a)}$  is defined as a reversed SFG  $\hat{G}_N$  with the additional conditions given by equations A.10 and A.11 for delay branches, A.17 and A.18 (or A.14) for static branches, and A.4 for the  $\hat{y}$ -branches. These definitions are summarized in Table 1.

Combining equations A.3, A.5, A.7, and A.8 results in

$$\Delta y_k(t) = \hat{u}'(\tau) \Delta u'(t-\tau). \quad (\text{A.19})$$

Since  $\hat{u}'(\tau)$  obviously does not depend on  $\varepsilon$ , it holds that

$$\frac{\partial y_k(t)}{\partial u'(t-\tau)} = \lim_{\varepsilon \rightarrow 0} \frac{\Delta y_k(t)}{\Delta u'(t-\tau)} = \hat{u}'(\tau). \quad (\text{A.20})$$

From Figure 7a, we observe that  $u'$  and the  $i$ th branch comes into the same node; remembering that a node performs a summation, it holds that

$$\frac{\partial y_k(t)}{\partial u'(t-\tau)} = \frac{\partial y_k(t)}{\partial v_i(t-\tau)}, \quad (\text{A.21})$$

and from Figure 7b that  $\hat{u}'(\tau) = \hat{v}_i(\tau)$ ; therefore

$$\frac{\partial y_k(t)}{\partial v_i(t-\tau)} = \hat{v}_i(\tau). \quad (\text{A.22})$$

It is important to note that for batch learning, equation A.22 is evaluated only for  $t = T$  and, therefore the adjoint branches defined in Table 1 are considered with  $t = T$ , where  $T$  is the final instant of the epoch.

## References

- Almeida, L. B. (1987). A learning rule for asynchronous perceptrons with feedback in combinatorial environment. In *Proc. Int. Conf. on Neural Networks* (Vol. 2, pp. 609–618).
- Back, A. D., & Tsoi, A. C. (1991). FIR and IIR synapses, a new neural network architecture for time series modelling. *Neural Computation*, 3, 375–385.
- Beaufays, F., & Wan, E. (1994). Relating real-time backpropagation and backpropagation-through-time: An application of flow graph interreciprocity. *Neural Computation*, 6, 296–306.
- Campolucci, P. (1998). A circuit theory approach to recurrent neural network architectures and learning methods. Doctoral dissertation in English, University of Bologna, Italy. PDF available online at <http://nnspe.eealab.unian.it/campolucci.P> or requested from [campoluc@tiscalinet.it](mailto:campoluc@tiscalinet.it).
- Campolucci, P., Marchegiani, A., Uncini, A., & Piazza, F. (1997). Signal-flow-graph derivation of on-line gradient learning algorithms. In *Proc. ICNN-97, IEEE Int. Conference on Neural Networks* (Houston, TX).
- Campolucci, P., Piazza, F., & Uncini, A. (1995). On-line learning algorithms for neural networks with IIR synapses. In *Proc. IEEE International Conference of Neural Networks* (Perth).
- Campolucci, P., Uncini, A., & Piazza, F. (1998). Dynamical systems learning by a circuit theoretic approach. *Proc. ISCAS-98, IEEE Int. Symposium on Circuits and Systems*.
- Campolucci, P., Uncini, A., Piazza, F., & Rao, B. D. (1999). On-line learning algorithms for locally recurrent neural networks. *IEEE Trans. on Neural Networks*, 10, 253–271.
- Director, S. W., & Rohrer, R. A. (1969). The generalized adjoint network and network sensitivities. *IEEE Trans. on Circuit Theory, CT-16*, 318–323.
- Gherry, M. (1989). A learning algorithm for analog, fully recurrent neural networks. In *Proc. Int. Joint Conference Neural Networks*, (Vol. 1, pp. 643–644).
- Haykin, S. (1994). *Neural networks: A comprehensive foundation*. New York: IEEE Press–Macmillan.
- Horne, B. G., & Giles, C. L. (1995). An experimental comparison of recurrent neural networks. In G. Tesauro, D. Touretzky, & T. Leen (Eds.), *Advances in neural information processing systems*, 7 Cambridge, MA: MIT Press.
- Lee, A. Y. (1974). Signal flow graphs—Computer-aided system analysis and sensitivity calculations. *IEEE Transactions on Circuits and Systems, cas-21*, 209–216.
- Martinelli, G., & Perfetti, R. (1991). Circuit theoretic approach to the backpropagation learning algorithm. *Proc. IEEE Int. Symposium on Circuits and Systems*.

- Mason, S. J. (1953). Feedback theory—Some properties of signal-flow graphs. *Proc. Institute of Radio Engineers*, 41, 1144–1156.
- Mason, S. J. (1956). Feedback theory—Further properties of signal-flow graphs. *Proc. Institute of Radio Engineers*, 44, 920–926.
- Narendra, K. S., Parthasarathy, K. (1991). Gradient methods for the optimization of dynamical systems containing neural networks. *IEEE Trans. on Neural Networks*, 2, 252–262.
- Nerrand, O., Roussel-Ragot, P., Personnaz, L., Dreyfus, G., & Marcos, S. (1993). Neural networks and nonlinear adaptive filtering: Unifying concepts and new algorithms. *Neural Computation*, 5, 165–199.
- Oppenheim, A. V., & Schaffer, R. W. (1975). *Digital signal processing*. Englewood Cliffs, NJ: Prentice Hall.
- Osowski, S. (1994). Signal flow graphs and neural networks. *Biological Cybernetics*, 70, 387–395.
- Pearlmutter, B. A. (1995). Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Trans. on Neural Networks*, 6, 1212–1228.
- Penfield, P., Spence, R., & Duiker, S. (1970). *Tellegen's theorem and electrical networks*. Cambridge, MA: MIT Press.
- Srinivasan, B., Prasad, U. R., & Rao, N. J. (1994). Backpropagation through adjoints for the identification of non linear dynamic systems using recurrent neural models. *IEEE Trans. on Neural Networks*, 5, 213–228.
- Tellegen, B. D. H. (1952). A general network theorem, with applications. *Philips Res. Rep.*, 7, 259–269.
- Tsoi, A. C., & Back, A. D. (1994). Locally recurrent globally feedforward networks: A critical review of architectures. *IEEE Transactions on Neural Networks*, 5, 229–239.
- Uncini, A., Vecchi, L., Campolucci, P., & Piazza, F. (1999). Complex-valued neural networks with adaptive spline activation function for digital radio links nonlinear equalization. *IEEE Transactions on Signal Processing*, 47, 505–514.
- Wan, E. A., & Beaufays, F. (1996). Diagrammatic derivation of gradient algorithms for neural networks. *Neural Computation*, 8, 182–201.
- Wan, E. A., & Beaufays, F. (1998). Diagrammatic methods for deriving and relating temporal neural networks algorithms. In M. Gori & C. L. Giles (Eds.), *Adaptive processing of sequences and data structures*. Berlin: Springer-Verlag.
- Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proc. of IEEE*, 78, 1550–1560.
- Williams, R. J., & Peng, J. (1990). An efficient gradient-based algorithm for on line training of recurrent network trajectories. *Neural Computation*, 2, 490–501.
- Williams, R. J., & Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1, 270–280.
- Williams, R. J., & Zipser, D. (1994). Gradient-based learning algorithms for recurrent networks and their computational complexity. In Y. Chauvin & D. E. Rumelhart (Eds.), *Backpropagation: Theory, architectures and applications* (pp. 433–486). Hillsdale, NJ: Erlbaum.