January 1994

A diagrammatic approach to gradient derivations for neural networks

Eric A. Wan

Francoise Beaufays

Submitted to NIPS94 (categorey: algorithms and architectures, preference: oral).

A Diagrammatic Approach to Gradient Derivations for Neural Networks

Eric A. Wan
Department of Electrical Engineering and Applied Physics
Oregon Graduate Institute of Science & Technology
P.O. Box 91000, Portland, OR 97291
ericwan@eeap.ogi.edu

Françoise Beaufays
Department of Electrical Engineering
Stanford University
Stanford, CA 94305-4055
francois@simoon.stanford.edu

Abstract

Deriving gradient algorithms for time-dependent neural network structures typically requires numerous chain rule expansions, diligent bookkeeping, and careful manipulation of terms. We show, however, that an efficient gradient descent algorithm may be formulated for any network structure with virtually no effort using a set of simple block diagram manipulation rules. Examples are provided that illustrate the simplicity of the approach for a variety of structures, including feedforward and feedback systems.

1 Introduction

In supervised learning, the goal is to find a set of network weights W that minimize a cost function $J = \sum_{k=1}^{K} L_k(\mathbf{d}(k), \mathbf{y}(k))$, where k is used to specify a discrete time index (the actual order of presentation may be random or sequential), $\mathbf{y}(k)$ is the output of the network, $\mathbf{d}(k)$ is a desired response, and L_k is a generic error metric that may contain additional weight regularization terms. For illustrative purposes,

we will work with the squared error metric, $L_k = \mathbf{e}(k)^T \mathbf{e}(k)$, where $\mathbf{e}(k)$ is the error vector.

Optimization techniques invariably require calculation of the gradient vector $\partial J / \partial W(k)$. At the architectural level, a variable weight w_{ij} may be isolated between two points in a network with corresponding signals $a_i(k)$ and $a_j(k)$ (*i.e.*, $a_j(k) = w_{ij}\ a_i(k)$). Using the chain rule, we get

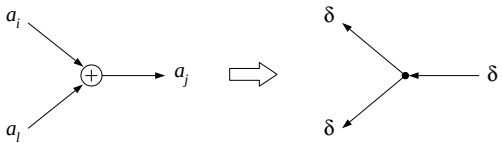
$$\frac{\partial J}{\partial w_{ij}(k)} = \frac{\partial J}{\partial a_j(k)} \frac{\partial a_j(k)}{\partial w_{ij}(k)} = \delta_j(k)\ a_i(k), \tag{1}$$

where we define the error gradient $\delta_j(k) \triangleq \partial J / \partial a_j(k)$. The error gradient $\delta_j(k)$ depends on the entire topology of the network. Specifying the gradients necessitates finding 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 diagrammatic method for deriving the delta terms associated with any network architecture.

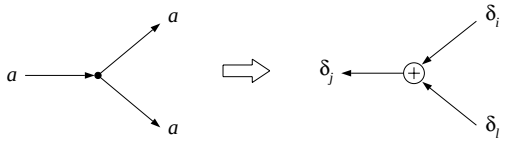
2 Networks and Reciprocal Construction Rules

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. Only discrete-time systems are considered. A *reciprocal network* is constructed 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.*



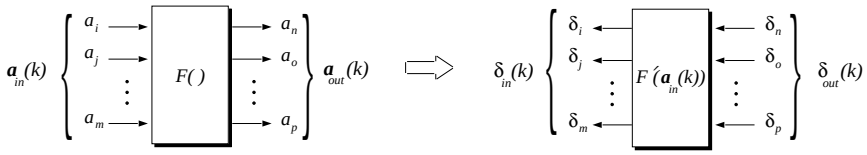
2. *Branching points are replaced with summing junctions.*



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

   Explicitly, $\delta_i(k) = f'(a_i(k))\ \delta_j(k)$, where $f'(a_i(k)) \triangleq \partial a_j(k) / \partial a_i(k)$. We have included the time index k to emphasize the *linear time-dependent* transmittance. Synaptic weights are a special case for which $a_j = w_{ij}\ a_i$, and the rule yields $\delta_i = w_{ij}\ \delta_j$. For activation functions, $a_n(k) = \tanh(a_j(k))$, and $f'(a_j(k)) = 1 - a_n^2(k)$.

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

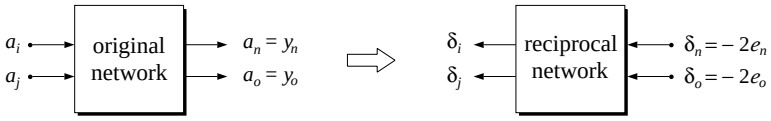


$\boldsymbol{\delta}_{in}(k) = F'(\mathbf{a}_{in}(k))\ \boldsymbol{\delta}_{out}(k)$, where $F'(\mathbf{a}_{in}(k)) \triangleq \partial \mathbf{a}_{out}(k)/\partial \mathbf{a}_{in}(k)$ corresponds to a matrix of partial derivatives. For shorthand, $F'(\mathbf{a}_{in}(k))$ will be written simply as $F'(k)$. Clearly both summing junctions and univariate functions are special cases of multivariate functions. A multivariate function may also represent a product junction (for sigma-pi units) or even another multi-layer network.

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

The unit delay, $a_j(k) = q^{-1}a_i(k) = a_i(k-1)$, is transformed into 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 reciprocal network in a causal manner is addressed in specific examples.

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 reciprocal network. These inputs are then set at each time step to $-2e_n(k)$. (For cost functions other than squared error, the input should be set to $\partial L_k/\partial y_n(k)$.)

These 6 rules allow direct construction of the reciprocal network from the original network. Note that there is a topological equivalence between the two networks. The order of computations in the reciprocal 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 *reciprocal network* is a linear time-dependent system. The signals $\delta_j(k)$ that propagate through the reciprocal 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 reciprocal network, completing the derivation. A formal proof of the validity and generality of this method is presented in Wan and Beaufays 1994.[1]

[1] The method presented here is similar in spirit to *Automatic Differentiation* (Griewank and Corliss, 1991). Automatic Differentiation is a simple method for finding derivative of functions and algorithms that can be represented by acyclic graphs. Our approach, however, applies to discrete-time systems with the possibility of feedback. In addition, we are concerned with diagrammatic derivations rather than computational rule-based implementations.

3 Examples

3.1 Backpropagation

We start by rederiving standard backpropagation (Rumelhart *et al.* 1986). Figure 1 shows a hidden neuron feeding other neurons and an output neuron in a multilayer network. (Superscripts are added to denote the layer. Also the time index k is omitted since multilayer networks are static structures.) The reciprocal network also shown in Figure 1 is found by applying the construction rules of the previous section.

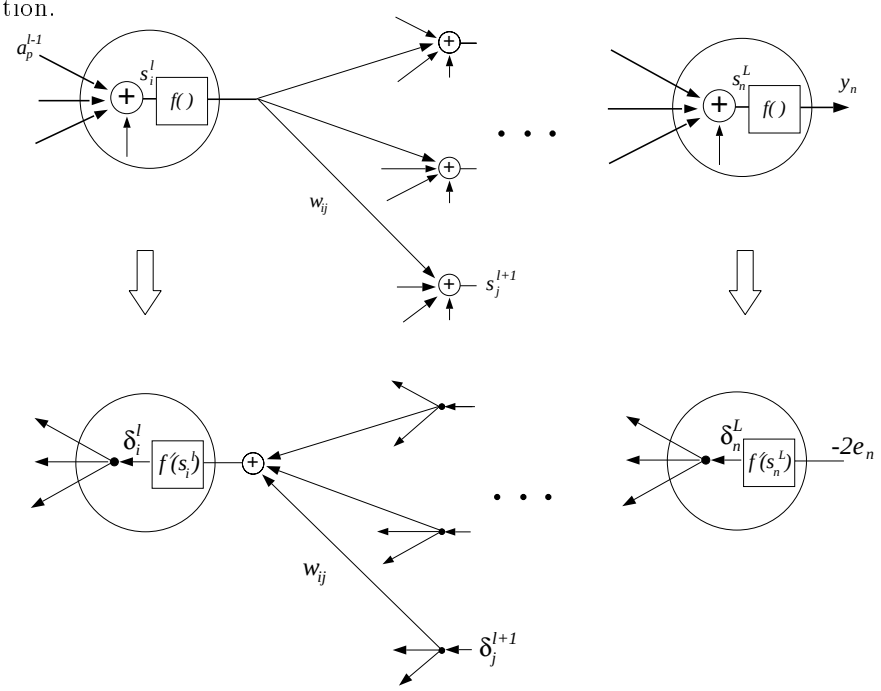


Figure 1: Feedforward network (top) and reciprocal counterpart (bottom).

From this figure, we may immediately write down the equations for calculating the delta terms:

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

These are precisely the equations describing standard backpropagation. In this case, there are no delay operators and $\delta_j = \delta_j(k) \triangleq \partial J / \partial s_j(k) = (\partial \mathbf{e}^T(k)\mathbf{e}(k))/\partial s_j(k)$ corresponds to an *instantaneous* gradient. Readers familiar with neural networks have undoubtedly seen these diagrams before. What is new is the concept that the diagrams themselves may be used directly to specify the delta's, completely circumventing all intermediate steps involving tedious algebra. It should further be emphasized that this approach still constitutes a *formal* derivation.

3.2 Backpropagation-Through-Time

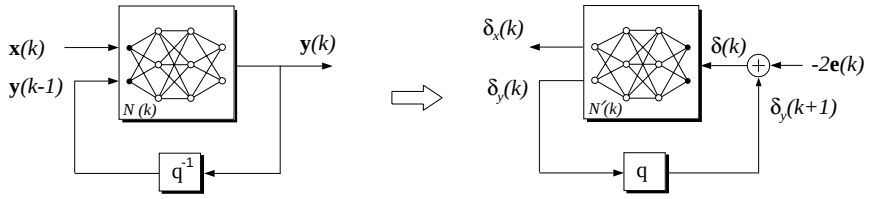


Figure 2: Recurrent network and backpropagation-through-time.

For the next example, consider a network with output feedback (see Figure 2) described by

$$\mathbf{y}(k) = \mathcal{N}(\mathbf{x}(k), \mathbf{y}(k-1)), \tag{3}$$

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 every other neuron and the structure is referred to as a *fully recurrent network*. 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 using chain rule expansions is extremely complicated. 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 reciprocal contruction rules (see Figure 2) we find immediately:

$$\boldsymbol{\delta}(k) = \boldsymbol{\delta}_y(k+1) - 2\mathbf{e}(k) = \mathcal{N}'(k+1)\boldsymbol{\delta}(k+1) - 2\mathbf{e}(k). \tag{4}$$

These are precisely the equations describing *backpropagation-through-time*, which have been derived in the past using either ordered derivatives or Euler-Lagrange techniques (Werbos, 1990). The diagrammatic approach is by far the simplest and most direct method. Note that in this case, the product $\mathcal{N}'(k)\boldsymbol{\delta}(k)$ may be calculated directly by a standard backpropagation of $\boldsymbol{\delta}(k)$ through the network at time k.

A variety of other recurrent architectures may be considered including radial basis networks with feedback. The system may be configured for neural control, using either full-state feedback or more complicated ARMA (AutoRegressive Moving Average) models. In all cases, the diagrammatic approach provides a direct derivation of the adaptation algorithm.

3.3 Cascaded Neural Networks

Consider two cascaded neural networks as illustrated in Figure 3. 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

$$u(k) = \mathcal{N}_1(W_1, x(k), x(k-1), x(k-2)), \tag{5}$$
$$y(k) = \mathcal{N}_2(W_2, u(k), u(k-1), u(k-2)), \tag{6}$$

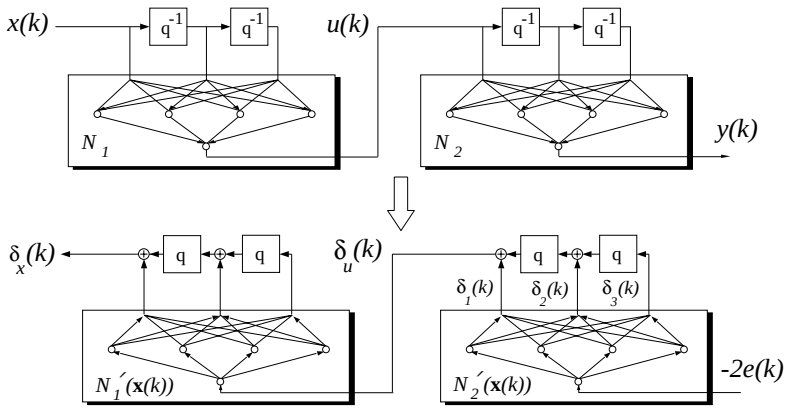


Figure 3: Cascaded networks (top) and reciprocal counterpart (bottom).

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 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 reciprocal network also shown in Figure 3, we simply label the desired signals and write down the gradient relations:

$$\delta_u(k) = \delta_1(k) + \delta_2(k+1) + \delta_3(k+2), \tag{7}$$

with

$$[\delta_1(k)\ \delta_2(k)\ \delta_3(k)] = -2e(k)\,\mathcal{N}_2'(\mathbf{u}(k)), \tag{8}$$

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 gradient for the weights in the first network is thus given by

$$\frac{\partial J}{\partial W_1(k)} = \delta_u(k)\,\frac{\partial u(k)}{\partial W_1(k)}, \tag{9}$$

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.

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:

$$\begin{aligned}
\frac{\partial e^2(k)}{\partial W_1} &= -2e(k)\frac{\partial y(k)}{\partial W_1} &(10)\\
&= -2e(k)\left[\frac{\partial y(k)}{\partial u(k)}\frac{\partial u(k)}{\partial W_1} + \frac{\partial y(k)}{\partial u(k-1)}\frac{\partial u(k-1)}{\partial W_1} + \frac{\partial y(k)}{\partial u(k-2)}\frac{\partial u(k-2)}{\partial W_1}\right]\\
&= \delta_1(k)\frac{\partial u(k)}{\partial W_1} + \delta_2(k)\frac{\partial u(k-1)}{\partial W_1} + \delta_3(k)\frac{\partial u(k-2)}{\partial W_1}, &(11)
\end{aligned}$$

where we define

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

The δ_i terms are found simultaneously by a single backpropagation of the error through the second network. Each product $\delta_i(k)(\partial u(k-i-1)/\partial W_1)$ is then found by backpropagation applied to the first network with $\delta_{i+1}(k)$ acting as an error. However, since the derivatives used in backpropagation are time-dependent, *separate* backpropagations are necessary for each $\delta_{i+1}(k)$. These equations, in fact, imply backpropagation through an unfolded structure and is equivalent to *weight sharing*. 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 reciprocal networks. Similar arguments can be used to derive an efficient on-line algorithm for adapting time-delay neural networks.

3.4 Temporal Backpropagation

For the last example, we return to Figure 1 for the feedforward network, but we now imagine replacing all scalar weights w_{ij} with discrete time linear filters to provide dynamic interconnectivity between neurons. Possible forms for the synaptic filters $\mathbf{w}_{ij}(q^{-1})$ are:

$$\mathbf{w}(q^{-1}) = \begin{cases} w & \text{Case I} \\ \sum_{m=0}^{M} w(m)q^{-m} & \text{Case II} \\ \dfrac{\sum_{m=0}^{M} a(m)q^{-m}}{1-\sum_{m=1}^{M} b(m)q^{-m}} & \text{Case III} \end{cases} \tag{12}$$

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. Case III represents the more general Infinite Impulse Response (IIR) filter, in which feedback is permitted.

Deriving the gradient terms for adapting filter coefficients is quite formidable if we use a direct chain rule approach. However, using the reciprocal construction rules it is straightforward to verify that the delta associated with each neuron is given by

$$\delta_i^l(k) = f'(s_i^l(k)) \sum_j \mathbf{w}_{ij}^{l+1}(q^{+1})\delta_j^{l+1}(k). \tag{13}$$

These equations define the algorithm known as *temporal backpropagation* (Wan 1993). 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 reciprocal network, backpropagation is achieved through the reciprocal filters $\mathbf{w}(q^{+1})$. Different realizations for the filters dictate how signals flow through the reciprocal structure as illustrated in Figure 4. For FIR filters, simple buffering allows for a causal implementation. For IIR filters, a forward and backward sweep is necessary similar to the implementation of backpropagation-through-time. In all cases, computations remain order N.

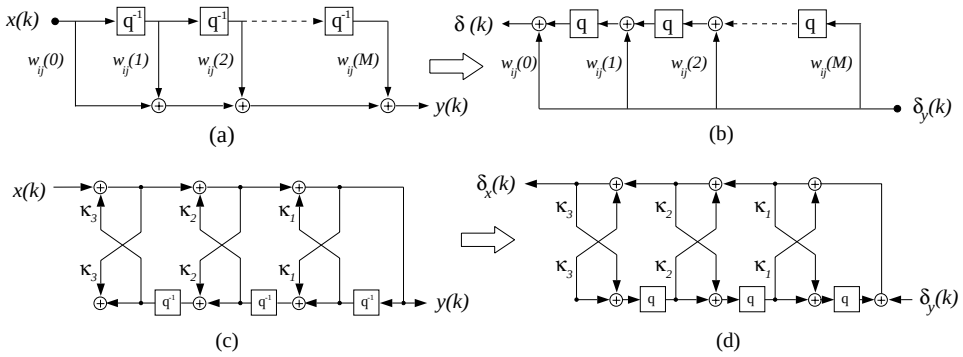


Figure 4: Sample synaptic filter realizations: a) FIR transversal, b) reciprocal FIR c) IIR lattice, d) reciprocal IIR.

4 Summary

The previous examples served to illustrate the ease in which algorithms may be derived using reciprocal construction rules. One starts with a diagrammatic representation of the network of interest. A reciprocal network is then constructed by simply swapping summing junctions for branching points, continuous functions with derivative transmittances, and time delays with time advances. The final algorithm is read directly off the reciprocal network. No chain rules are needed. The diagrammatic approach provides a unified framework for formally deriving gradient algorithms for arbitrary network architectures, network configurations, and systems.

Acknowledgements

Funding in part by EPRI contract RP801013 and NSF grant IRI 91-12531.

References

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

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.

Wan, E. (1993) 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., and Beaufays, F. (1994) Diagrammatic Derivation of Gradient Algorithms for Neural Networks. Submitted to *Neural Computation*, 1994.

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.