A Circuit Theory Approach to Recurrent Neural Network Architectures and Learning Methods

Tesi di Dottorato di

Paolo Campolucci

Coordinatore:
Prof. Ugo Reggiani

Tutore:
Prof. Francesco Piazza

Febbraio 1998
Summary

Recurrent neural networks are powerful tools in particular as circuits for signal processing applications. Classical design of circuits for signal processing is based on three basic assumptions: linearity, stationarity and the existence of second order statistics (usually Gaussianity). These basic assumptions are required for the statistical and mathematical development but they are never satisfied on real signals, even if in some cases they can be approximately true. This means that the results of a classical signal processing solution is not optimal. Artificial neural networks can be successfully used with classical signal processing techniques to improve the performance of the systems.

Standard neural networks are static, i.e. their output depends only on the current inputs and not on the past ones and therefore they are unable to process temporal sequences. To overcome this limitation the first technique developed in literature was to implement an external buffer that feeds the network with delayed samples, in this way the network has a memory equal to the length of the buffer. On applications where a long or unknown memory depth is required, buffered networks result to be inadequate.

On the contrary, recurrent neural networks are able to model both short and deep temporal dependencies between samples of the input and target sequences. They implement the concept of implicit memory, i.e. the use of feedback instead of just a delay line buffer. In this case, the memory resolution and depth can be adaptively trade-off by the network itself, during learning, in the optimal way that depends on the task. The design of the architecture is less dependent on the temporal features of the problem.

On the other hand, recurrent neural networks require learning methods much more complex than their static counterparts and their derivation is difficult and strongly architectural dependent. In this work, it is shown that a circuit theory approach, based on Signal Flow Graphs (SFG), can bring to a new theory of learning that makes learning algorithms derivation trivial, even for complex non-linear dynamical circuits, such as recurrent neural networks.

Not only a neural architecture can be rigorously represented as a circuit, but this representation can be used to derive the learning method by circuit transformation. It is shown that two dual learning techniques, named Backward Computation (BC) and Forward Computation (FC), can always be uniquely derived for any Signal Flow Graph, even if delays or feedback are present. Backward Computation is based on the derivation of an adjoint SFG from the original SFG, while Forward Computation is founded on the concept of derivative SFG. For batch learning, BC is the algorithm of choice since it is simpler, but in the more involved on-line case, both are interesting. FC can directly operate on-line while BC requires an history truncation. FC has a complexity that depends only on the circuit dimension (linearly with the square of the number of components) while BC depends also on the history length considered (linearly with it as with the number of components).

These two methods can be combined for optimal performance in the case of special circuit structure, such as locally recurrent neural networks, and two new on-line learning methods are here proposed in such case, named Causal Recursive Back Propagation (CRBP) and Truncated Recursive Back Propagation (TRBP). They were derived before the BC-FC unifying theory and later interpreted by its concepts.

This work provides also an architectural comparison of recurrent neural networks and, in particular, of models with local feedback. Stability issues are also discussed for such models with the derivation of some new stability control techniques that are of interest also in the linear case. The BC-FC theory is derived here for first order learning algorithms, but an extension is possible to second order methods. This extension is left to the reader, but some new techniques to implement second order methods are proposed here, based on conjugate gradient, on which the BC-FC methods can be usefully applied.

Copyright 1998 Paolo Campolucci.
Request of reprints or comments can be addressed: via Matteotti n. 111, 60121 Ancona, Italy.
Alla mia famiglia
Chapter 4

ON-LINE LEARNING FOR LOCALLY RECURRENT NEURAL NETWORKS

4.1 Introduction

4.2 The batch Recursive Back-Propagation (RBP) algorithm
   4.2.1 Notation
   4.2.2 The forward phase
   4.2.3 The learning algorithm (RBP)

4.3 On-line algorithms: Causal Recursive Back-Propagation (CRBP)

4.4 On-line algorithms: Truncated Recursive Back-Propagation (TRBP)

4.5 Experimental results
   4.5.1 First simulation: Back-Tsoi system identification
   4.5.2 Second simulation: 16-PAM system identification

4.6 Analysis of complexity and implementation issues

4.7 Conclusions

4.A Appendix: RBP for Local Feedback Multi-Layered Networks (MLNs) and Auto-Regressive MLP
   4.A.1 Activation feedback MLN
   4.A.2 Output feedback MLN
   4.A.3 Auto Regressive MLP

Chapter 5

SIGNAL-FLOW-GRAPH APPROACHES:

BACKWARD COMPUTATION

5.1 Introduction

5.2 Signal flow graphs: a theorem similar to Tellegen's theorem

5.3 SFG approach to sensitivity calculation by Backward Computation (BC)

5.4 SFG approach to non-linear dynamic systems learning
   5.4.1 Cost functions and parameters updating rules
   5.4.2 Cost function gradient calculation by SFGs
   5.4.3 Detailed steps of the Backward Computation procedure
Chapter 1

INTRODUCTION

1.1 PRELIMINARY REMARKS

1.1.1 Artificial neural networks for signal processing

Classical design of circuits for Signal Processing (SP) applications is based on three basic assumptions: linearity, stationarity and the existence of second order statistics (usually Gaussianity).

These basic assumptions are required for the statistical and mathematical development but they are never satisfied on real signals, even if in some cases they can be approximately true. This means that the results of a classical signal processing solution is not optimal. Artificial Neural Networks (NNs) can be successfully used with classical SP techniques to improve the performance of the systems (Haykin 1996). In the following, the primary features of neural networks, in the context of signal processing, will be described.

Neural networks are systems developed by the connection of many simple units (neurons) each of them with a non-linear modeling capability. It is due to the non-linear activation function present in each neuron. Neurons work in parallel, therefore neural networks are massively parallel processing systems with a strong fault tolerant behavior. Many synapses or neurons must be damaged before the overall NN system stops to work properly.

Neural networks can be adaptive if a suitable on-line learning method is designed. In such case, they can be adapted to follow changes in the working environment by minimizing a time varying cost function. For example, network parameters can be trained on-line to track a certain real system. Of course, classical linear adaptive filters, such as Finite Impulse Response (FIR) or Infinite Impulse Response (IIR) (Shynk 1989), exhibit the same property (on-line
adaptability) but they are limited to a fixed linear and simple structure. As in the linear case, some stability constraints must be guaranteed by the learning method to avoid instabilities.

Another advantage of NNs is that they do not require any knowledge of the underlying probability distribution as, on the contrary, do classical SP statistical methods.

Finally, NNs are universal approximators in the sense that they can approximate any continuous non-linear input-output mapping with any desired accuracy using a suitable number of neurons (Cybenko 1989, Hornik et al. 1989).

1.1.2 Artificial neural networks are non-linear adaptive circuits

Since NNs are based on the idea of connectivity of simple units and each of them can be easily modeled by the proper combination of many simple components, it is intuitive that one of the most effective formalism for NNs can be that provided by the theory of circuits.

The theory of graphs (directed graphs) and in particular of Signal Flow Graphs (SFG) is a fundamental mathematical theory in this field. It not only allows to rigorously describe the model, with no loss of details, but also to do some graphical computation with it, see Chapters 5 and 6.

Signal Flow Graphs were originally developed by Mason (1953, 1956) for linear networks. A Signal-flow-graph is a network of directed branches interconnected at certain points called nodes. Each branch has associated two variables or signals: the initial and the final variables. Each branch has associated a transfer function or transmittance that specifies the way in which the final signal depends on the initial signal.

The signal flows in the direction of the branch indicated by an arrow in the middle of the branch itself. The final variable of the branch is specified by computing the transfer function at the initial variable. The initial variables of all branches coming out from a certain node are the summation of the final variables of all the branches coming into the node.

Neural networks can be defined as directed graphs consisting of nodes with interconnecting synaptic and activation links. The directed graph is complete if all the computation is specified and partially complete if only the connectivity among neurons is specified and not the computation inside each neuron.

Signal-flow-graphs can be applied not only to static NNs but also to dynamic NNs, with or without recursion, by using two basic concepts. Firstly, a delay branch can be introduced as a one-unit memory element that gives as output a one-time step delayed version of the input signal. Secondly, the graph can be
designed to have loops, so that feedback can be implemented, giving an infinite memory to the overall network.

In this work, discrete time circuits will be covered due to their great practical application, even if an extension of the theory and methods developed to the continuous time case is possible, as sometimes explicitly stated.

The theory presented in Chapters 5 and 6, can deal with both the above extensions and allow to automatically derive the learning method suited for a special architecture just by simple graphical transformations.

1.2 THESIS ORGANIZATION

Recurrent neural networks are powerful tools in particular as circuits for signal processing applications and can be successfully used with classical signal processing techniques to improve the performance of the systems.

Standard neural networks are static, i.e. their outputs depend only on the current inputs and not on the past ones and therefore they are unable to process temporal sequences. To overcome this limitation, the first technique developed in literature was to implement an external buffer that feeds the network with delayed samples, in this way the network has a memory equal to the length of the buffer. On applications where a long or unknown memory depth is required, buffered networks result to be inadequate.

On the contrary, recurrent neural networks are able to model both short and deep temporal dependencies between samples of the input and target sequences. They implement the concept of implicit memory, i.e. the use of feedback instead of just a delay line buffer. In this case, the memory resolution and depth can be adaptively trade-off by the network itself, during learning, in the optimal way that depends on the task. The design of the architecture is less dependent on the temporal features of the problem.

On the other hand, recurrent neural networks require learning methods much more complex than their static counterparts and their derivation is difficult and strongly architectural dependent, especially in the on-line case.

In the following of the chapter, some general knowledge will be presented that is used though the whole thesis, i.e. the idea of on-line learning for providing adaptive systems, the concepts of time representation and memory and the description of some fields of applications of dynamic neural networks.

On Chapter 2, the main architectures for dynamic neural networks will be described by a circuit formalism. Basically, they can be grouped in four classes, i.e. buffered MLPs with or without external feedback, time delay neural networks, locally recurrent neural networks and fully recurrent neural networks.
A comparison is made from a representation power and a complexity point of view.

The main traditional approaches for gradient computation and learning of systems will be described on Chapter 3. They are basically, Automatic Differentiation, Linear Sensitivity Computation, Back Propagation Through Time and Real Time Recurrent Learning. They will be introduced by a tutorial on Werbos’ fundamental theory of Ordered Derivatives.

Chapter 4 will detail two new on-line training procedures for locally recurrent neural networks (Causal Recursive Back Propagation, CRBP, and Truncated Recursive Back Propagation, TRBP), that allow to accurately compute the gradient for these complex structures trading accuracy with computational complexity, in a flexible way. They are proposed to unify some known learning methods and overcome their limitations.

The two previous techniques apply only to locally recurrent neural networks. To remarkably increase the generality and applicability, a new Signal Flow Graph (SFG) theory of sensitivity computation and learning for non-linear dynamical circuits will be developed on Chapter 5 and 6. It makes possible to deal with non-linear dynamical circuits with delays and/or feedback and allows to derive gradient information of a given circuit by the graphical construction of another SFG, the adjoint SFG or the derivative SFG. The calculation of the adjoint SFG (Chapter 5) correspond to a Backward Computation (BC) of the gradient. On Chapter 6, the dual method is developed for deriving a Forward Computation (FC) of the gradient by the derivative SFG.

Recurrent neural networks, as any system with feedback, can exhibit instabilities, i.e. the output or an internal state of the system can grow without limit even with a bounded input. This behavior is not desirable in practical applications and should be avoided. Even if stability control techniques are known and well developed for linear adaptive recursive filters (such as IIR filters), no such technique is usually applied to neural networks neither specific methods are developed in the neural networks context. On Chapter 7, some new stability control techniques, applicable to the linear and non-linear case, will be discussed and compared to the classical ones for the linear case.

The BC-FC theory is derived here for first order learning algorithms, but an extension is possible to second order methods. This extension is left to the reader, but some new techniques to implement second order methods are proposed on Chapter 8, based on conjugate gradient, on which the BC-FC methods can be usefully applied.
1.3 ADAPTIVE TRAINING OF MODELS

With “supervised learning or training” we mean the adaptation process of the parameters of a model in a way that its behavior will be closer to that described by a set of given input-output pairs (Rumelhart et al. 1986). It is usually assumed that the structure of the model is a-priori chosen depending on the required task and only some parameters must be adapted. Nevertheless, in some cases, the learning method can select some structural variables such as the number of hidden units.

The set of given input-output pairs is a subset of all the possible values for the problem considered and it must obey to some rules that can be summarized in the following. The samples (or pairs) of the training subset must be Independent Identically Distributed (IID) on the total set defining the task. Only with such conditions satisfied, the training can be successful in the sense that the trained model will generalize on input values not used for training.

The generalization property is of great importance since the error on samples not used for training (“test error”) is closer than the “training error” to a real evaluation of the performance of the trained system.

On the contrary, the capacity of a model is the capability to represent input-output pairs: more pairs represented (adapting the model parameters) means an higher capacity of the model. Models based on the idea of connectivity exhibit an high capacity choosing a sufficient number of hidden units. In the limit of an infinite number of hidden units, two layers neural networks can model any input-output mapping with an arbitrary accuracy. This property is usually known as universal approximation theorem (Cybenko 1989, Hornik et al. 1989).

For Recurrent Neural Networks (RNNs) a similar theorem can be proved. Funahashi and Nakamura (1993) proved that each trajectory with finite horizon of a dynamical system can be approximated by a recurrent neural network with enough units. Moreover, it can be proved that RNNs are equivalent to Turing machines for symbolic manipulations. All these results state that RNNs are models with an high capacity.

Learning of dynamical systems (by models such as RNNs) can be implemented in two ways, i.e. batch mode and on-line mode. Batch mode means to train the dynamical model on an entire sequence adapting the parameters only once. In other words, the system operation and learning phase are employed at different time. This means that for a circuit operating in a transmission system, such as an equalizer, the learning phase can only be implemented after an interruption of the operation of the system and with the use of a finite length training sequence. On the contrary, on-line learning means that learning can be performed at the
same time as the normal operation of the system, adapting the model at each
time step.

Batch learning is usually implemented by Back Propagation Through Time
(Werbos 1990), (Section 3.5), i.e. by a gradient descent of a cost function
defined usually as the Mean Squared Error (MSE) over the entire sequence.
This method can be correctly applied only for finite sequences (batch mode) due
to the definition of the cost function.

Nevertheless, an on-line method can be derived by choosing for example the
squared error at the current sample as cost function (e.g. Williams and Zipser
1994). This approach is known as stochastic gradient since the cost being used
is “randomly” varying from sample to sample and obviously it is not as smooth
as the MSE. Stochastic gradient performs a descent on a direction not as
accurate as the method based on the true gradient and so it is potentially slower,
but it must be considered that in the second case the adaptation is performed
just once for each epoch. Moreover the stochastic component can be useful for
escaping from local minima; therefore, it is usually the preferred method.

1.4 REPRESENTATIONS OF TIME

The difference between the pattern of a static problem of pattern recognition
and the sample of a problem of sequence processing is that in the first case the
order of the different patterns is completely irrelevant (e.g. they can be
randomly shuffled to improve the learning), while for a sequence the order of its
samples is absolutely significant.

To model this relation among samples the system must have memory, i.e. its
output must depend not only on the current sample but also on the previous
ones.

One of the best known representation of time is the state space representation.
It assumes that for each time instant all the past information can be summarized
by the value of the state vector variable. The state transition function is
sufficient to model all the dynamics of the system while an output function can
be necessary to obtain the output variable from the state.

The previous representation is an implicit one. An explicit one is obtained using
a function of all the previous samples. In this case, the past information is
explicitly stored and processed with no “compression”, as in the state variable
case. An explicit representation of a dynamical system is the Volterra’s series, a
series expansion similar to the Taylor series for a static function. The problem
with Volterra’s series is that there is no general criterion to approximate it to a
finite order of expansion (Frasconi 1994).
From the explicit and implicit representations of time, two possible implementations of memory in a system are possible (Mozer 1993), (Chapter 2), i.e. the Tapped Delay Line memory (explicit) and the Exponential trace memory, implemented by feedback connections (implicit). Buffered MLPs are examples of an explicit representation of memory, while RNNs of an implicit one.

Other implementations of memory are possible other than the two mentioned. In particular a kind of memory in some ways between the previous ones is the “Gamma memory” model (De Vries and Principe 1992). In Gamma memory, the delay operator, used in conventional Tapped Delay Lines, is replaced by a single pole discrete time filter. Gamma Memory is a dispersive delay line with dispersion regulated by an adaptable parameter, so that it is possible to adaptively trade off “temporal depth” with “temporal resolution” (De Vries and Principe 1992). Refer also to (Schmidhuber 1992a).

What makes the Gamma memory and its special cases particularly useful is that they can be computed by an incremental update procedure, whereas forms such as the Gaussian one require evaluating the convolution of the kernel with the input sequence at each time step. Such convolutions, while occasionally used (Bodenhausen and Waibel 1991, Tank and Hopfield 1987, Unnikrishnan, et. al. 1991), are not practical because of the computational and storage requirements.

1.5 DYNAMIC NEURAL NETWORKS
APPLICATIONS

1.5.1 System identification and control

RNNs have been applied with success to a wide class of problems in which a long memory mixed with a strong non-linearity is present and has to be modeled (IEEE Trans. on Neural Networks, March 1994). Of course, identification and control of non-linear dynamic systems is a problem belonging to such class.

For control problems, even if the system to be controlled is almost static, the overall system controller-plant is dynamic due to the feedback control connection, therefore the techniques developed for RNNs are very useful in such cases. In the following, five basic neural network controller designs will be shortly presented (Miller et al. 1990).

In supervised control, an Artificial Neural Network (ANN) learns to imitate a human or a computer program which already knows how to perform a task.

In direct inverse control, an ANN learns the mapping from the position of a robot arm, or something similar, back to the actuator signals which would move
the arm to that position. The ANN is then used to make the arm follow a desired trajectory or reach a desired target point.

In neural adaptive control, ANNs are substituted for the (linear) mappings used in conventional adaptive control. Conventional adaptive control includes designs like the Self-Tuning Regulator (STR) and Model-Reference Adaptive Control (MRAC). These designs, like inverse control, try to achieve a pre-specified target.

In Back Propagation Through Time (BPTT), the user specifies a utility function or performance measure to be maximized and a model of the external environment. Backpropagation is used to calculate the derivative of utility summed across all future times with respect to current actions. These derivatives are then used to adapt the ANN which outputs the actions, or to adapt a schedule of actions.

In adaptive critics methods, the user again supplies a function or measure to be maximized. The long-term optimization problem is solved by adapting an additional ANN, called a critic network, which evaluates the progress that the system is making. In other words, the output of the critic may be seen as a kind of secondary utility function (or its derivatives), that somehow represents the sum of the original utility function across all future time. The network which outputs the actions is adapted to maximize this secondary utility function in the immediate future. Many different adaptive critic designs have been proposed.

Real applications of neural networks to identification and control problems include chemical plants, automotive subsystems, space station stabilization, supersonic airplane stabilization, echo cancellation, transmission channel pre-distortion and equalization.

1.5.2 Time series prediction

While in system identification the problem is described by a sequence of input-output pairs, in time series prediction the problem is to predict future values of a signal given its past values. This prediction can be performed by a temporal model such as a dynamic neural network, by modeling the system underlying the time series (Weigend and Gershenfeld 1993, Refenes 1995). Assuming that the system is stationary and that its output depends only on past output values, i.e. no external input influences the system, then a temporal model able to identify the system will correctly predict the series.

In real applications, these two hypotheses are usually not satisfied so the problem becomes more involved. For taking into account a slow variation of the system, an on-line adaptation of the model to the system should be performed. For taking into account external inputs, they should be selected on the basis of
relevance to the output and the most important ones, depending on the desired prediction accuracy, should be introduced into the model. The problem is that usually such external inputs are not known; sometimes not only their values but also the nature of some external inputs. Therefore the time series is noisy and the random component should be modeled by the predictor.

Traditional models for time series prediction are statistical in nature and essentially estimate the probability that a particular event will occur given a set of past information. Typical neural network models are closely related to statistical regression, and can be shown to estimate Bayesian a-posteriori probabilities given an appropriately formulated problem.

Real fields of application of neural networks to time series prediction include signal processing and compression, economic series prediction (exchange rate prices, stock values, currency exchange rates…), electric load prediction and others.

1.5.3 Signal classification and speech recognition

This application field is the temporal counterpart of classical pattern recognition, where patterns are classified, not only on the base of their shape, but also on the order of presentation.

Speech recognition is a clear example of signal classification even if many others exist, e.g. (Haykin 1996, Revel et al. 1996, Filippetti et al. 1996, Frasconi 1994). In its simplest version, i.e. isolated word recognition, the problem is to classify a sequence of samples of an utterance given a set of possible words. Apparently, this can look as a standard pattern recognition problem if we assume that the entire sequence is a single pattern, but the problem is that the length of the utterance can easily be different from time to time. Therefore, the structure of the pattern can change. For this reason is not correct to approach this problem simply as a pattern recognition one.

The correct recognition process is a chain of many blocks of operations. They are: pre-emphasis (spectrum equalization), frame blocking (divide the utterance in time stationary blocks), windowing and frequency features extraction (extraction of relevant information for the recognition, from each frame). After these signal processing steps, a sequence of features vector (one vector for each frame) is obtained. At this point, the problem is that different utterances of the same word can easily have a different number of frames. To extract information from such sequence of data, a dynamical model such as a RNN can be used.

On the contrary, a standard approach is to perform a kind of normalization on the length of the utterance. The problem is that the compression-expansion in
the process of speaking is highly non-linear, e.g. usually vocal sounds can vary in length, while certain consonant tracts such as ‘p’, ‘t’, ‘b’ stay constant.

Nowadays, one of the best solutions for speech recognition problems is an hybrid technique based on Hidden Markov Model (HMM) and Neural networks (Haykin 1996).

Other real application fields of signal classification are continuous speech recognition, radar target classification, Electro-Encephalo-Graphic (EEG) and Electro-Cardio-Graphic (ECG) signal classification.
Chapter 2

ARCHITECTURES FOR DYNAMIC NEURAL NETWORKS

2.1 INTRODUCTION

Recently dynamic recurrent neural networks have been attracting much attention from the scientific community (see the special issues of IEEE Trans. on Neural Networks, March 1994 and of Neurocomputing, June 1997) because they can be very powerful on temporal processing, e.g. Digital Signal Processing (DSP), system identification and control and temporal pattern recognition (e.g. speech recognition, signals classification), refer to Chapter 1.

Since dynamic neural networks are much more complex than static neural networks on both architectures and learning algorithms, they were first studied and implemented as extension of static models.

Nowadays, it is clear that dynamic NNs cannot be limited to extension of static neural architectures such as buffered versions of such models, since they can not model complex dynamics. They are simple, easy to train and do not require much specific knowledge on non-linear dynamics, but they can just show a finite memory horizon. The insertion of a feedback buffer, such as in the so called NARX models or Elman’s network, does not solve the problem if the learning does not exploit the recurrent nature of the network, i.e. if standard backpropagation instead of Back Propagation Through Time is employed.

The choice for a standard learning method such as backpropagation is motivated by a conceptual and implementation simplicity and by a lower
computational complexity. The price for this choice is a much slower and less stable learning.

Two main methods exist to provide a static neural network with dynamic behavior: the insertion of a buffer somewhere in the network, i.e. implementing an explicit memory of the past inputs, or the use of feedback. In both approaches, an arbitrary input \((x[t])\) may influence a future output \((y[t+h])\), so that \(\frac{\partial y[t+h]}{\partial x[t]}\) is not equal to zero for some \(h\). In the case of asymptotic stability this derivative goes to zero when \(h\) goes to infinity. The value of \(h\) for which the derivative becomes negligible is called temporal depth, whereas the number of adaptable parameters divided by the temporal depth is called temporal resolution (De Vries and Principe 1992).

In the following, first buffered static networks architectures will be described and motivations for the use of feedback will be given, then fully recurrent networks will be shown. They are general and powerful but for specific tasks other simpler architectures can be preferred, such as locally recurrent neural networks. These models will be deeply discussed on the last section.

### 2.2 BUFFERED MULTI-LAYER-PERCEPTRONS AND TIME DELAY NEURAL NETWORKS

The simplest dynamic network is the buffered Multi Layer Perceptron (MLP) in which Tapped Delay Lines (TDLs) of the inputs are used. The buffer can be applied at the network inputs only, keeping the network internally static as in buffered MLPs (Haykin 1994) (see Fig. 2.2.1), or at the input of each neuron as in MLP with Finite Impulse Response (FIR) filter synapses (FIR-MLP, see Fig. 2.2.2) (Back and Tsoi 1991, Wan 1990, Haykin 1994, Back et al. 1994, Benvenuto et al. 1994) often called Time Delay Neural Network (TDNN) (Waibel et al. 1989, Lang et al. 1990) and in adaptive time delay neural networks (Day and Davenport 1993, Lin et al. 1995).

It is well known that buffered MLP and FIR-MLP can be shown to be theoretically equivalent (Wan 1990) since internal buffers can be implemented as an external one. The problem with this approach is that first layers sub-networks must be replicated (with shared weights) (Wan 1990) and so the complexity is much higher than considering the buffer internal. Therefore, buffered MLP and FIR-MLP are different architectures with regard to a real implementation.

The main disadvantage of the buffer approach is the limited past history horizon thereby preventing modeling of arbitrary long time dependencies (Frasconi et al. 1992, Frasconi 1994, Hochreiter and Schmidhuber 1995) between inputs and desired outputs. It is also difficult to set the length of the buffer, given a certain
application. Moreover, to have sufficient temporal depth, a long buffer, i.e. a large number of inputs weights, could be required, usually with a decrease in generalization performance and an increase in the overall computational complexity. In other words, the buffer approach with no feedback has the maximum temporal resolution, at the cost of a low temporal depth.

$$y[t]$$

$$x[t]$$  $$x[t-1]$$  ......  $$x[t-n]$$

Multi Layer Perceptron

$$x[t]$$  $$x[t-1]$$  ......  $$x[t-n]$$

Tapped Delay Line

**Fig. 2.2.1**  Buffered Multi Layer Perceptron with input buffer.

**Fig. 2.2.2**  Model of the neuron for the FIR-MLP. Sgm is the activation function (usually a sigmoid). The variables are defined in §4.2.1.

Recently a new buffer type called Gamma Memory has been proposed by J.C. Principe, B. de Vries and M.A. Motter (De Vries and Principe 1992, Motter and Principe 1994) for which the delay operator, used in conventional TDLs, is replaced by a single pole discrete time filter. Gamma Memory is a dispersive delay line with dispersion regulated by an adaptable parameter, so that it is possible to adaptively trade off temporal depth with temporal resolution. In addition to these advantages of temporal depth and resolution characteristics, it is known that neural networks with feedback have useful dynamic modeling behavior (Elman 1990, Mozer 1988).

In the past few years, a growing interest has been devoted to methods which allow introduction of temporal dynamics into the multilayer neural model. In
fact the related architectures are less complex and easier to train with respect to
the fully recurrent networks. The major difference among these methods lies in
how the feedback are included in the network:

- **externally**: as in the Narendra-Parthasarathy MLP (Narendra and
  Parthasarathy 1991) also known as NARX network, where TDLs are used for
  the outputs that feedback to the input of the network (see Fig. 2.2.3), and in the
  Elman’s network (Elman 1990);
- **internally**: as in locally recurrent neural networks.

![Buffered Multi Layer Perceptron with input and output buffers, sometimes called Narendra-Parthasarathy or NARX neural network.](image)

2.3 FULLY RECURRENT NEURAL NETWORKS

The main example of implementation of feedback is the classical fully recurrent
neural network, i.e. a single layer of neurons fully interconnected with each
other (Rumelhart et al. 1986, Pineda 1987, Almeida 1987, Pearlmutter 1995,
and Zipser 1989a and 1989b, Santini et al. 1995, Haykin 1994), or several such
layers (Puskorius and Feldkamp 1994). Such recurrent networks however
exhibit some well known disadvantages: a large structural complexity \( O(n^2) \)
weights are necessary for \( n \) neurons) (Ku and Lee 1995) and a slow and difficult
training, e.g. (Tsoi and Back 1994). In fact they are very general architectures
which can model a large class of dynamical systems, but on specific problems
simpler dynamic neural networks which make use of available prior knowledge
can be better (Frasconi 1994, Mozer 1988).

Let’s consider a fully recurrent neural network with \( n \) neurons and \( m \) inputs.
Let’s define \( y(t) \) the \( n \times 1 \) vector of the neuron outputs at time \( t \), \( x^{net}(t) \) the
\( m \times 1 \) vector of the external inputs at time \( t \), \( x(t) \) the \((m+n) \times 1 \) vector
obtained from the two previous ones by concatenation as $x(t) = \Delta \left[ \frac{x^{net}(t)}{y(t)} \right]$. Let’s define $U$ the set of indexes of $x(t)$ for which, $x_k$ is the output of a neuron and $I$ the one for external inputs, so that $x_k(t) = \begin{cases} x^{net}_k(t) & \text{if } k \in I \\ y_k(t) & \text{if } k \in U \end{cases}$

Let’s define $W$ the $n \times (m+n)$ weights matrix of elements $w_{ij}$ that is the weight of the connection from neuron $j$, if $j \in U$, or external input $j$, if $j \in I$, to neuron $i$. To implement biases, one of the $m$ external inputs can be taken to be equal to 1 for each $t$. For an example of fully recurrent neural network see Fig. 2.3.1.

![Fig. 2.3.1](image)

**Fig. 2.3.1** Two equivalent representations for a fully recurrent neural network with 3 neurons and 2 external inputs. Any sub-set of the neuron outputs can be defined to be the network outputs. The second representation shows the $3 \times 5$ weights matrix. In each feedback loop a one-unit delay is present.

The forward equations for the network are:

$$s_k(t+1) = \sum_{l \in U} w_{kl} y_l(t) + \sum_{l \in I} w_{kl} x^{net}_l(t) = \sum_{l \in U \cup I} w_{kl} x_l(t)$$

$$y_k(t+1) = f_k[s_k(t+1)]$$

where $f_k$ is the non-linear activation function of the $k$-th unit, and $k \in U$.

A common choice is the unsigned sigmoid: $f_k[s_k(t)] = \frac{1}{1+e^{-s_k(t)}}$, but in general the activation function can be any squashing and first order differentiable function.
Let’s define the instantaneous error as

\[
e_k(t) = \begin{cases} 
  d_k(t) - y_k(t) & \text{if } k \in T(t) \\
  0 & \text{otherwise}
\end{cases}
\]

where \(d_k(t)\) is the desired output for the \(k\)-th network output, at time \(t\), and \(T(t)\) the set of index \(k \in U\) for which a target is specified at time \(t\).

The instantaneous global squared error is

\[
e^2(t) = \sum_{k \in U} \left[ e_k(t) \right]^2
\]

while the global error over a \([t_0, t_1]\) learning epoch is

\[
E(t_0, t_1) = \sum_{\tau = t_0}^{t_1} e^2(\tau).
\]

### 2.4 Locally Recurrent Neural Networks

The newest approach to the temporal processing by neural networks is realized by Locally Recurrent Neural Networks (LRNNs) or Local Feedback Multi-Layer Networks (LF-MLN) (Tsoi and Back 1994, Frasconi et al. 1992, Mozer 1993). In these structures, classical Infinite Impulse Response (IIR) linear filters (Shynk 1989), here called also Auto Regressive Moving Average (ARMA) models, are used directly or with some modifications. Different architectures arise depending on how the ARMA model is included in the network.

The first architecture is the IIR-MLP proposed by A.D. Back and A.C. Tsoi (Back and Tsoi 1991, Tsoi and Back 1994) where static synapses are substituted by conventional IIR adaptive filters (see Fig. 2.4.1).

![Fig. 2.4.1 Model of the neuron for the IIR-MLP.](image)

The second architecture is the activation feedback locally recurrent multilayer network studied by P. Frasconi, M. Gori, G. Soda et al. (Frasconi et al. 1992,
Frasconi 1994, Bengio et al. 1992). The output of the neuron summing node is filtered by an Auto Regressive (AR) adaptive filter (all poles transfer function) before feeding the activation function; in the most general case the synapses are FIR adaptive filters (see Fig. 2.4.2).

![Fig. 2.4.2 Model of the neuron for the locally recurrent activation feedback MLN.](image)

The activation feedback multilayer network is a particular case of the IIR-MLP, when all the synaptic transfer functions of the same neuron have the same denominator.

The third structure is the output feedback locally recurrent network proposed by Gori et al. (Frasconi et al. 1992, Frasconi 1994, Bengio et al. 1992). In this architecture the IIR filter is not simply placed in the classical neuron model but is modified to make the feedback loop pass through the non-linearity, i.e. the one time step delayed output of the neuron is filtered by a FIR filter whose output is added to the inputs contributions, providing the activation. Again in the general model the synapses can be FIR filters (see Fig. 2.4.3).

![Fig. 2.4.3 Model of the neuron for the locally recurrent output feedback MLN.](image)

The work of Gori et al. has its foundation in the work of M. Mozer (1988) in which the main idea was the introduction of context units to include memory in a network, substituting the spatial metaphor of the external buffer (common at
that time) with the recurrent context approach, as also suggested by J.L. Elman 
(1990). Context units are dynamic recurrent neurons placed in the first layer to 
process the input signals while the following layers are supposed to be static. 
This architectural constraint also used in the works of Gori et al. has been 
chosen basically to simplify the learning phase.

At last, there is another architecture (see Fig. 2.4.4) that has not been studied as 
the previous ones: it was proposed by Mozer (1988) (with one delay feedback 
dynamic units in the first layer only) and by R.R. Leighton and B.C. Conrath in 
(Leighton and Conrath 1991) (multiple delays and no restriction on the position 
of dynamic units).

It is again a multilayer network where each neuron has FIR filter synapses and 
an AR filter after the activation function (AR-MLP). It is easy to see that this 
network is a particular case of the IIR-MLP, followed by linear all-pole filters.

Recently Diagonal Recurrent Neural Networks (DRNN) (Ku and Lee 1995), 
have been proposed for dynamic systems control, claiming relevant results. This 
arichitecture is also a particular case of output feedback MLN since DRNN is a 
two layer network with static linear output neurons and dynamic hidden neurons 
with static synapses but with one delay feedback from the output. Again the 
position of dynamic units is restricted to the first layer only.

Another version of locally recurrent neural network was presented in (Uchiyama 
et al. 1989) with a biological motivation: a multilayer connection of perceptrons 
with low pass temporal filtering of the activation.

The major advantages (Tsoi and Back 1994, Mozer 1993, Ku and Lee 1995, 
1994) of locally recurrent neural networks with respect to buffered MLPs or 
fully recurrent networks can be summarized as follows:

1) well known neuron interconnection topology, i.e. the efficient and 
hierarchic multilayer;
2) small number of neurons required for a given problem, due to the use of powerful dynamic neuron models;

3) generalization of the popular FIR-MLP (or TDNN) to the infinite memory case;

4) prewired forgetting behavior (Frasconi et al. 1992), needed in applications such as DSP, system identification and control;

5) simpler training than fully recurrent networks;

6) many training algorithms could be derived from filter theory and recursive identification.
Chapter 3

GRADIENT CALCULATION
AND LEARNING:
TRADITIONAL APPROACHES

3.1 INTRODUCTION

Even if the calculation of the gradient of a given function with respect to some of its independent variables looks apparently as a simple problem it can be highly complicated by many factors, primary the presence of memory and the time varying nature of the function (e.g. for on-line learning). Moreover, since this function can be very complex, in some fields of interest, such as neural networks, with thousand of variables, parameters and elementary sub-functions and the gradient can be required on-line, the computational complexity is an important issue.

Internally static networks can be trained by the simplest algorithms: for the buffered MLP (Section 2.2) with only input buffer (i.e. with no recursion) the standard BP should be used, while for the Narendra-Parthasarathy network (Section 2.2) the so called ‘open loop’ approximation (Bauer and Geisel 1990, Jordan 1986) of the standard BP is usually employed. It consists in opening the loop during the backward phase, feeding the network with the desired outputs instead of the true network outputs. In IIR adaptive filter theory, this is the equation error approximation of the true output error approach (Shynk 1989), in neural network theory, this is the teacher forced technique (Williams and Zipser 1989).

Extension of Back Propagation to recurrent networks was firstly proposed in (Rumelhart et al. 1986, Pineda 1987, Almeida 1987). F. Pineda and L.B.

BPTT is a family of algorithms which extends the BP paradigm to dynamic networks. There are two main points of view to understand what BPTT is. The first is an intuitive one: time unfolding of the recurrent network, i.e., for single layer single feedback delay fully recurrent networks one can think at the network state at time \( t \) as if it was obtained from the \( t \)-th layer output of a multilayer network with \( T \) layers (Rumelhart et al. 1986), where \( T \) is the length of the sequence. The other point of view is a mathematical one and it is based on the Werbos’ theory of ordered derivatives (Werbos 1974). P.J. Werbos provided a mathematical tool to rigorously compute the derivatives of a certain variable with respect to another one in complex structures described by ordered mathematical relations (for example a neural network).

Actually, with ordered derivatives it is possible to derive both BPTT and RTRL algorithms in the same framework (Beaufays and Wan 1994, Srinivasan et al. 1994). The difference between BPTT and RTRL is in how the chain rule derivative expansion is applied. More specifically, during the learning phase, in BPTT the neural network is computed backward both in the layer and time dimensions, whereas in RTRL it is calculated forward. Reversing the signal flow graph provides the great efficiency of BPTT but the necessary reversion of time makes it non-causal even if only one delay is present inside the network (i.e. after an adaptable parameter in the signal flow graph) (Wan and Beaufays 1996).

Therefore BPTT is local in space but not in time, is computationally simple but is non-causal; therefore it can be implemented only in batch mode. For on-line adaptation some approximations are needed, namely causalization and truncation of past history, as explained in (Williams and Peng 1990, Elman 1990, Williams and Zipser 1994) for fully recurrent neural networks.

RTRL instead is local in time but not in space, computationally complex but intrinsically on-line. However even RTRL implements an approximated calculation of the gradient if the parameters are continually adapted since its
true derivation would require constant weights (Williams and Peng 1990, Williams and Zipser 1994).

R.J. Williams and D. Zipser (1994) report better performance and convergence rate for truncated BPTT than RTRL and explain this result stating that the history truncation approximation can be better than the approximation implemented in RTRL.

Most of these methods were studied for general fully recurrent networks. Recently E. Wan and F. Beaufays (1996) proposed a simple method to derive BPTT for discrete time dynamic neural networks composed of a general interconnection of weights, delays, additive units, differentiable non-linearities. This derivation can be carried out with simple transformations of the signal flow graph of the network itself; however the algorithm derived in this way is always non-causal and the authors did not address the question of on-line learning for networks with feedback. For a more general approach solving this and some other limitations, see Chapter 5. An analogous theory for the forward approach can be derived, see Chapter 6.

In the following Section the theory of Ordered Derivatives will be shortly presented to show how the BPTT and RTRL algorithms are traditionally derived then the theory of Automatic Differentiation will be outlined to shows its similarity with Werbos’ theory. Section 3.4 describe the classical theory of sensitivity for linear network since it is strongly connected to the RTRL forward algorithm, then the BPTT and RTRL approaches are respectively developed.

### 3.2 ORDERED PARTIAL DERIVATIVES


The idea underlying this theory is to describe a complex system by a set of ordered equations that link a set of ordered variables. Let \( \{z_1, z_2, \ldots, z_n\} \) be a set of \( n \) variables related by a set of \( n \) equations. Such set of equations is ordered if each variable \( z_i \) depend just on \( \{z_1, \ldots, z_{i-1}\} \). Therefore, each equation of an ordered set can be expressed as \( z_i = f_i(z_1, \ldots, z_{i-1}) \). In order to compute the value of \( z_i \), \( \{z_1, \ldots, z_{i-1}\} \) must be computed first.

An example of ordered set of equations is the following:
For computing a partial derivative it must be specified which variable to hold constant and which varying. By the usual notation all the variables are held constant with the exception of the one written in the denominator, e.g. the derivative $\frac{\partial z_3}{\partial z_1}$ means that $z_1$ is variable and $z_2$ constant, therefore its value is 3, in this example.

Let’s define partial ordered derivative a partial derivative in which the variables involved are determined by an ordered set of equations. In the ordered partial derivative of $z_j$ with respect to $z_i$, i.e. $\frac{\partial^+ z_j}{\partial z_i}$ the constant terms are $\{z_{i+1},\ldots,z_{j-1}\}$, the varying ones are $\{z_{i},\ldots,z_{j-1}\}$.

By the usual mathematical notation:

$$\frac{\partial^+ z_j}{\partial z_i} = \frac{\partial z_j}{\partial z_i} \bigg|_{\{z_{i+1},\ldots,z_{j-1}\} \text{ held constant}}$$

By the definition of ordered partial derivative the following properties can be proved:

$$\frac{\partial^+ z_j}{\partial z_i} = \begin{cases} 
\frac{\partial z_j}{\partial z_i} & \text{if } j = i + 1 \\
1 & \text{if } j = i \\
0 & \text{if } j < i
\end{cases}$$

(3.2.3)

If $j > i + 1$, the ordered partial derivative can be computed by one of the following chain rule expansions:

$$\frac{\partial^+ z_j}{\partial z_i} = \frac{\partial z_j}{\partial z_i} + \sum_{k=i+1}^{j-1} \frac{\partial z_j}{\partial z_k} \frac{\partial^+ z_k}{\partial z_i}$$

(3.2.4)

or

$$\frac{\partial^+ z_j}{\partial z_i} = \frac{\partial z_j}{\partial z_i} + \sum_{k=i+1}^{j-1} \frac{\partial^+ z_j}{\partial z_k} \frac{\partial z_k}{\partial z_i}$$

(3.2.5)
Application of one of the last two equations to the ordered equations set (3.2.1) results in \( \partial^+ z_3 / \partial z_1 = 27 \). Note that instead \( \partial z_3 / \partial z_1 = 3 \). Since the first expansion computes \( \partial^+ z_j / \partial z_i \) using \( \partial^+ z_k / \partial z_i \) with \( k < j \), i.e. \( k \) is increasing during the computation, it is called a forward expansion (Williams and Zipser 1989).

On the contrary, the second equation calculates \( \partial^+ z_j / \partial z_i \) using \( \partial^+ z_k / \partial z_k \) with \( k > i \), i.e. \( k \) is decreasing during the computation, therefore it is called a backward expansion (Werbos 1990).

Both forward and backward equations are started by null initial conditions due to (3.2.3).

### 3.3 Automatic Differentiation

The theory of automatic differentiation, refer to (Griewank 1988) for a survey, was born to solve an interesting problem in scientific simulations: how to derive a software code that computes the gradient of a function defined by a piece of code. The gradient information can be useful in many applications such as sensitivity computation or optimization of generic functions. By automatic differentiation the exact numerical value of the gradient of the given function can be found.

Let’s consider a function \( y = f(x) : \mathbb{R}^n \rightarrow \mathbb{R} \) defined by the following code:

**Original program**

For \( i = n + 1, n + 2, \ldots, m \)

\[
x_i = f_i(\{x_j\})_{j \in I_i}
\]

\( y = x_m \)

The basic functions \( f_i \) depend on the variables \( x_j \) already computed, with \( j \) belonging to the index sets

\[
I_i \subseteq \{1,2,\ldots,i-1\} \quad \text{for } i = n + 1, n + 2, \ldots, m
\]

So \( f \) is the composition of \((m-n)\) elementary or library functions \( f_i \), whose gradients

\[
\nabla f_i = \{ \partial f_i / \partial x_j \}_{j \in I_i}
\]

are assumed to be computable at all arguments of interest.
For example, this is clearly the case when all \( f_i \) represent either elementary arithmetic operations, i.e. \(+,-,\times,\div\), or non-linear system functions of a single argument, e.g. logarithms, exponential and trigonometric functions. Almost all scalar functions of practical interest can be represented in this factorable form. Any number of arguments \( n_i \equiv |I_i| < i \), where \( |I_i| \) denotes cardinality, can be allowed.

The original code can be thought as a computational graph with the vertex set \( \{x_i\}_{i=1}^m \). If \( j \) belongs to \( I_i \) then an arc runs from \( x_j \) to \( x_i \). Because of the restriction on \( I_i \) only a-cyclic graph can be obtained.

### 3.3.1 Forward Accumulation

The gradient of \( x_i \) with respect to the independent variables \( \{x_j\}_{j=1,\ldots,n} \) can be denoted by \( \nabla x_i \); from the original code by chain rule:

**Forward Extension**

For \( i = 1,2,\ldots,n \)

\[
\nabla x_i = e_i
\]

For \( i = n+1, n+2,\ldots,m \)

\[
x_i = f_i(x_j)_{j \in I_i}
\]

\[
\nabla x_i = \sum_{j \in I_i} \frac{\partial f_j}{\partial x_j} \nabla x_j
\]

\[
y = x_m
\]

\[
g = \nabla x_m
\]

where \( e_i \) denotes the \( i \)-th vector of the Cartesian basis in \( \mathbb{R}^n \) and \( g \) is the gradient of \( y \) with respect to the independent variables, i.e. \( \nabla f \).

The chain rule expansion implemented for the forward accumulation method is Werbos’ forward expansion (3.2.4).

### 3.3.2 Reverse Accumulation

Let’s associate with each intermediate variable \( x_i \) the scalar derivative

\[
\overline{x_i} = \frac{\partial}{\partial x_i}
\]

rather than the gradient vector \( \nabla x_i \).
By definition it holds \( x_m = 1 \) and for \( i = 1, \ldots, n \)

\[
\frac{\partial}{\partial x_i} f(x) = \bar{x}_i \tag{3.3.6}
\]

By chain rule:

\[
\bar{x}_j = \sum_{i \in \Omega_j} \frac{\partial f_i}{\partial x_i} \bar{x}_i \tag{3.3.7}
\]

where

\[
\Omega_j = \{ i : i \leq m \text{ and } j \in I_i \} \tag{3.3.8}
\]

\( \bar{x}_j \) can be computed once all \( \bar{x}_i \) with \( i > j \) are known.

This is equivalent to the following code:

**Reverse Extension**

For \( i = n+1, n+2, \ldots, m \)

\[
x_i = f_i \langle x_j \rangle_{j \in I_i} \tag{Forward Sweep}
\]

\[
\bar{x}_i = 0
\]

\( y = x_m \)

\( \bar{x}_m = \gamma \)

\( \langle \bar{x}_i \rangle_{i=1}^n = \bar{g} \)

For \( i = m, m-1, \ldots, n+1 \)

\[
\bar{x}_j = \bar{x}_j + \frac{\partial f_i}{\partial x_j} \bar{x}_i \quad \text{for all } j \in I_i \tag{Reverse Sweep}
\]

\( g = \langle \bar{x}_i \rangle_{i=1}^n \)

When the initial vector \( \bar{g} \) is set to zero and \( \gamma \) equals one then the resulting vector \( g \) is the gradient \( \nabla f \).

The chain rule expansion implemented for the reverse accumulation method is Werbos’ backward expansion (3.2.5).

### 3.4 LINEAR CASE: FORWARD COMPUTATION

A general expression for the sensitivity of the transfer function of a discrete-time linear network with respect to its parameters can be derived by a version of
the Tellegen’s theorem for Signal Flow Graph (Oppenheim and Schafer 1975). It implements a forward chain rule expansion of derivatives (3.2.4), but it is valid only in the linear case.

Let’s denote $T_{ab}(z)$ the transfer function between nodes $a$ and $b$ of a linear network, where each source node but the one connected with $a$ is equal to zero. The variable associated to node $b$, i.e. $W_b(z)$ is given by

$$W_b(z) = T_{ab}(z)X_a(z)$$  

(3.4.1)

The sensitivity of $T_{ab}(z)$ with respect to variation of a parameter $F_{nm}$ placed between nodes $n$ and $m$ (Fig. 3.4.1) is defined as

$$\frac{\partial T_{ab}(z)}{\partial F_{nm}}$$  

(3.4.2)

By the mentioned Tellegen’s theorem it holds (Oppenheim and Schafer 1975)

$$\frac{\partial T_{ab}}{\partial F_{nm}} = T_{an}T_{mb}$$  

(3.4.3)

where the dependence on $z$ has been omitted. (3.4.3) is called network sensitivity formula. By (3.4.1) it holds

$$\frac{\partial W_b}{\partial F_{nm}} = \frac{\partial T_{ab}}{\partial F_{nm}}X_a = T_{an}T_{mb}X_a = W_aT_{mb}$$  

(3.4.4)

This formula show how to compute the derivative of a linear network output with respect to one of its parameter.

This equation seems complex but the graph interpretation is simple (Forssén 1990), see Fig. 3.4.2. Given a linear dynamic time-invariant circuit, e.g. a FIR or
IIR filter, the derivative of the output with respect to one of its parameters can be computed. By this method, the derivative is obtained by a system with the same structure as the original one (Fig. 3.4.2).

\[ T(z^{-1}, \alpha) \]

Fig. 3.4.2 System output gradient computation for linear systems.

3.5 BACK-PROPAGATION-THROUGH-TIME

In this Section, the BPTT algorithm will be derived for a general fully recurrent neural network, please refer to Section 2.3 for definitions and notations.

The chain rule expansion implemented for the BPTT method is Werbos’ backward expansion (3.2.5).

3.5.1 Epochwise Back-Propagation-Through-Time

The Back-Propagation-Through-Time (BPTT) algorithm was presented for the first time in (Rumelhart et al. 1986, Pearlmutter 1989, Werbos 1990). It was developed as a batch method based on learning epochs and therefore named epochwise. The cost function to be minimized in this case is the error over the all epoch, e.g. the overall squared error \( E(t_0, t_e) \).

In the following, the BPTT method will be derived considering a fully recurrent neural network architecture.

The updating rule is simply the steepest descend method:

\[ \Delta w_{ij} = -\mu \frac{\partial E(t_0, t_e)}{\partial w_{ij}} \]  

(3.5.1)

The gradient can be derived by a backward chain rule expansion
\[
\frac{\partial E(t_0, t_1)}{\partial w_{ij}} = \sum_{\tau=t_0}^{t_1} \frac{\partial E(t_0, t_1)}{\partial w_{ij}(\tau)} = \sum_{\tau=t_0}^{t_1} \frac{\partial E(t_0, t_1)}{\partial s_i(\tau)} \frac{\partial s_i(\tau)}{\partial w_{ij}(\tau)} = \sum_{\tau=t_0}^{t_1} \frac{\partial E(t_0, t_1)}{\partial s_i(\tau)} x_j(\tau - 1)
\]

(3.5.2)

where

\[
\frac{\partial E(t_0, t_1)}{\partial s_i(\tau)} = \frac{\partial E(t_0, t_1)}{\partial y_j(\tau)} \frac{\partial y_j(\tau)}{\partial s_i(\tau)} = \frac{\partial E(t_0, t_1)}{\partial y_j(\tau)} f_i'(s_i(\tau))
\]

(3.5.3)

if \( \tau = t_1 \) then

\[
\frac{\partial E(t_0, t_1)}{\partial y_j(\tau)} = -2e_i(\tau)
\]

(3.5.4)

if \( \tau < t_1 \) then

\[
\frac{\partial E(t_0, t_1)}{\partial y_j(\tau)} = \frac{\partial E(\tau, t_1)}{\partial y_j(\tau)} = \frac{\partial e^2(\tau)}{\partial y_j(\tau)} + \frac{\partial e^2(\tau + 1, t_1)}{\partial y_j(\tau)} = -2e_i(\tau) + \sum_{l \in U} \frac{\partial E(t_0, t_1)}{\partial s_j(\tau + 1)} w_{li}
\]

\[
= -2e_i(\tau) + \sum_{l \in U} \frac{\partial E(t_0, t_1)}{\partial s_j(\tau + 1)} w_{li}
\]

(3.5.5)

For each \( i \in U \) and \( \tau \in [t_0, t_1] \) let's define

\[
\delta_j(\tau) = -\frac{\partial E(t_0, t_1)}{\partial s_i(\tau)}
\]

(3.5.6)

By (3.5.3), (3.5.4), (3.5.5) it holds

\[
\delta_j(\tau) = \begin{cases} 
  f_i'(s_i(\tau)) \cdot 2e_i(\tau) & \text{if } \tau = t_1 \\
  f_i'(s_i(\tau)) \left[ 2e_i(\tau) + \sum_{l \in U} \delta_j(\tau + 1) w_{li} \right] & \text{if } t_0 \leq \tau < t_1
\end{cases}
\]

(3.5.7)

By (3.5.2) the updating rule can be computed as

\[
\Delta w_{ij} = -\mu \frac{\partial E(t_0, t_1)}{\partial w_{ij}} = \mu \sum_{\tau=t_0}^{t_1} \delta_i(\tau) x_j(\tau - 1)
\]

(3.5.8)

The algorithm defined in this way is the epochwise BPTT. It has the smallest complexity for a batch gradient computation. For a fully recurrent neural network with \( n \) neurons and therefore \( O(n^2) \) weights, naming \( h \) the length of the learning epoch, this algorithm requires \( O(nh) \) real cells memory and
$O(n^2h)$ arithmetic operations (for computing the $\delta_i(\tau)$ variables and for the parameters updating).

### 3.5.2 Truncated Back-Propagation-Through-Time

The on-line version of the BPTT learning algorithm is named Truncated BPTT, and was presented with variations in (Williams and Peng 1990) for fully recurrent neural networks. In this case, the cost function to be minimized is the instantaneous squared error at the current time step, i.e. $e^2(t)$.

The updating rule now is

$$\Delta w_{ij} = -\mu \sum_{\tau=t-h+1}^{t} \frac{\partial e^2(t)}{\partial w_{ij}(\tau)}$$

where $h$ is the past history considered in the calculation. For an exact calculation $h$ should be equal to the number of time step occurred from the beginning but, since it is increasing with time, this method cannot be implemented on-line. Therefore $h$ must be fixed and chosen higher for a better approximation at the cost of higher complexity.

It holds

$$\frac{\partial e^2(t)}{\partial w_{ij}(\tau)} = \frac{\partial e^2(t)}{\partial s_i(\tau)} \frac{\partial s_i(\tau)}{\partial w_{ij}(\tau)} = \frac{\partial e^2(t)}{\partial s_i(\tau)} x_j(\tau - 1)$$

where

$$\frac{\partial e^2(t)}{\partial s_i(\tau)} = \frac{\partial e^2(t)}{\partial y_i(\tau)} \frac{\partial y_i(\tau)}{\partial s_i(\tau)} = \frac{\partial e^2(t)}{\partial y_i(\tau)} f'_i(s_i(\tau))$$

if $\tau = t$ then

$$\frac{\partial e^2(t)}{\partial y_i(\tau)} = -2e_i(\tau)$$

if $\tau < t$ then

$$\frac{\partial e^2(t)}{\partial y_i(\tau)} = \sum_{i \in U} \frac{\partial e^2(t)}{\partial s_i} \frac{\partial s_i(\tau+1)}{\partial y_i(\tau)} = \sum_{i \in U} \frac{\partial e^2(t)}{\partial s_i} w_{hi}(\tau+1)$$

For each $i \in U$ and $\tau \in [t_0, t]$ let’s define

$$\delta_i(\tau) = -\frac{\partial e^2(t)}{\partial s_i(\tau)}$$

By (3.5.11), (3.5.12) and (3.5.13)
\[ \delta_i(\tau) = \begin{cases} f_i'(s_i(\tau)) \cdot 2e_i(\tau) & \text{if } \tau = t \\ f_i'(s_i(\tau)) \sum_{l \in U} \delta_j(\tau + 1)w_{ij}(\tau + 1) & \text{if } \tau < t \end{cases} \] (3.5.15)

By (3.5.10) the updating rule becomes

\[ \Delta w_{ij} = -\mu \sum_{\tau=t-n+1}^{t} \frac{\partial e^2(t)}{\partial w_{ij}(\tau)} = \mu \sum_{\tau=t-n+1}^{t} \delta_i(\tau)x_j(\tau - 1) \] (3.5.16)

For the sake of simplicity, often the weights are not memorized in practice but just the current weights value is used with similar performance (Williams and Peng 1990), i.e. (3.5.15) becomes

\[ \delta_i(\tau) = \begin{cases} f_i'(s_i(\tau)) \cdot 2e_i(\tau) & \text{if } \tau = t \\ f_i'(s_i(\tau)) \sum_{l \in U} \delta_j(\tau + 1)w_{ij} & \text{if } \tau < t \end{cases} \] (3.5.17)

Epochwise BPTT and Truncated BPTT differ in the updating rule and in the calculation of the \( \delta_i(\tau) \) variables, compare (3.5.8) with (3.5.16) and (3.5.7) with (3.5.15). Since in the on-line case the parameters are updated at each time step, the error injection, i.e. the \( 2e_i(\tau) \) term in (3.5.7), must be performed only at the current time step, therefore is not present in the second line of (3.5.15).

For a fully recurrent neural network with \( n \) neurons and therefore \( O(n^2) \) weights, naming \( h \) the length of the past history considered, this algorithm requires \( O(nh) \) real memory cells and \( O(n^2h) \) arithmetic operations per time step (for computing the \( \delta_i(\tau) \) variables and for the parameters updating).

### 3.6 REAL-TIME-RECURRENT-LEARNING

In this Section, the RTRL algorithm will be derived for a general fully recurrent neural network, please refer to Section 2.3 for definitions and notations.

The Real Time Recurrent Learning algorithm was presented in (Williams and Zipser 1989) for fully recurrent neural networks. The chain rule expansion implemented for RTRL is Werbos’ forward expansion (3.2.4).

The parameters updating rule is

\[ \Delta w_{ij} = -\mu \frac{\partial e^2(t)}{\partial w_{ij}} \] (3.6.1)

where
\[
\frac{\partial e^2(t)}{\partial w_{ij}} = - \sum_{k \in U} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}}
\]  
(3.6.2)

where \( \frac{\partial y_k(t)}{\partial w_{ij}} \) is derived differentiating the forward pass equations

\[
\frac{\partial y_k(t+1)}{\partial w_{ij}} = \frac{\partial y_k(t+1)}{\partial s_k(t+1)} \frac{\partial s_k(t+1)}{\partial w_{ij}} = f'_k(s_k(t+1)) \frac{\partial s_k(t+1)}{\partial w_{ij}} =
\]

\[
= f'_k(s_k(t+1)) \left[ \sum_{l \in U} w_{kl} \frac{\partial y_l(t)}{\partial w_{ij}} + \delta_{ik} x_j(t) \right]
\]  
(3.6.3)

where \( \delta_{ik} \) is the Kronecker’s delta (\( \delta_{ik} = 1 \) if \( i = k \); \( \delta_{ik} = 0 \) if \( i \neq k \)). Since weights do not affect initial conditions

\[
\frac{\partial y_k(t_0)}{\partial w_{ij}} = 0
\]  
(3.6.4)

All these equations hold \( \forall k \in U, \forall i \in U \cap e \forall j \in U \cup I \), and they allow to compute for each \( t \) the exact gradient of \( e^2(t) \) with respect to the weights with the assumption that they are constant in time. Updating the parameters on-line the previous equations are approximated. The approximation is better if the learning rate is smaller, since this implies slower parameters variations.

For a fully recurrent neural network, with \( n \) neurons and therefore \( O(n^2) \) weights, this algorithm requires \( O(n^3) \) real memory cells and \( O(n^4) \) arithmetic operations per time step.

Therefore, for \( h \) chosen small with respect to \( n^2 \) (i.e. for large networks), that is the usual case, truncated BPTT is much more efficient than RTRL both computationally and for memory requirements.
Chapter 4

ON-LINE LEARNING FOR LOCALLY RECURRENT NEURAL NETWORKS

4.1 INTRODUCTION

The learning paradigms presented in Sections 3.5 and 3.6 were studied for general fully recurrent networks. On the other side, several on-line learning algorithms have been presented for specific dynamic multilayer neural networks, most of the times with no reference to each other and to the general paradigms.

In (Wan 1990) a learning algorithm, named Temporal Back Propagation, was proposed by E. Wan. This is an on-line version of the BPTT approach (Wan and Beaufays 1996). However it can only be applied to the non-recurrent FIR-MLP (see Section 2.2).

Back Propagation for Sequences (BPS) is a learning algorithm proposed by M. Gori et al. (Gori et al. 1989, Bengio et al. 1992, Frasconi 1994, Pearlmutter 1990 and 1995) both for output and activation Local Feedback MLNs (LF-MLNs), (see Section 2.4). It is interesting because it is local both in time and in space, computationally simple and with small memory requirement; in fact it is only slightly more complex than standard BP. However it can be applied only to LF-MLN with no dynamic units in layers other than the first one. BPS basically is the classical backpropagation on the multilayer network with a recursive computation only inside each dynamic neuron. Due to the architectural constraints, this algorithm does not implement backpropagation through a dynamic structure.
The same approach was proposed by M. Mozer (1988) that independently derived a quite similar algorithm named Focused Backpropagation for a particular version of AR-MLP (Section 2.4). BPS was rediscovered in (Ku and Lee 1995) where it was derived for a structure that is a particular case of the output feedback LF-MLN and was applied to control problems with good results.

In (Back-Tsoi 1991 and 1993), a learning algorithm for IIR-MLP (see Section 2.4), was proposed by A.D. Back and A.C. Tsoi. It is similar to BPS, implementing both a backpropagation and a recursive computation, but without any architectural restriction. However, to avoid dynamic backpropagation, they propose using static backpropagation even through a dynamic neuron. This is too strong an approximation, as outlined by this work both theoretically and experimentally.

Analogous learning algorithms are also: Auto Regressive BP proposed by R.R. Leighton and B.C. Conrath (1991) for the AR-MLP, and the algorithm in (Uchiyama et al. 1989). They are equivalent to Back-Tsoi algorithm since they also use instantaneous backpropagation without implementing the full backpropagation through a dynamic unit. So in the following of the chapter we will call Back-Tsoi algorithm the method with instantaneous backpropagation and we will not refer anymore to the works in (Leighton and Conrath 1991, Uchiyama et al. 1989).

From this background, it comes out the necessity for the derivation of accurate and efficient learning methods; the result of the Doctorate work on this matter is presented on this chapter. The on-line algorithms proposed in this chapter, i.e. CRBP, whose basic ideas were presented in (Campolucci 1994, Campolucci et al. 1995, Campolucci, Uncini and Piazza 1997a) and TRBP shortly presented in (Campolucci, Uncini and Piazza 1997b), implement and combine together BPTT and RTRL paradigms for locally recurrent networks. They work with the most general locally recurrent networks and implement a more accurate computation of the gradient than the Back-Tsoi method. While Back-Tsoi algorithm uses an instantaneous error as cost function, the CRBP and TRBP algorithms can minimize the global error; this fact results in an improved stability of the algorithm. The names that will be used, i.e. Causal Recursive Back Propagation and Truncated Recursive Back Propagation (not to be confused with Recurrent Back Propagation (Pineda 1987, Almeida 1987)) were chosen to remember the dual nature of the algorithms: BPTT style formulas are used to backpropagate the error through the neurons and recursive computation of derivatives inside each neuron is implemented to calculate weights variations. It is well known that RTRL and BPTT approaches are equivalent in batch mode operation (Beaufays and Wan 1994): they compute the same weights variations.
using different chain rule expansions. Since CRBP and TRBP use another expansion of the same derivatives, they become equivalent to RTRL and BPTT when working in batch mode (RBP). Since off-line BPTT is computationally simpler than RTRL or RBP, it is the algorithm of choice working in batch mode, unless the memory requirement is an issue. In this case, RTRL can be preferred since for long sequences it requires less memory (see Section 4.6).

However the four methods are not equivalent in on-line mode. In this case, truncated BPTT must be considered instead of BPTT and CRBP or TRBP instead of RBP. It must be stressed that in CRBP or TRBP each local feedback of a certain neuron is taken into account with no history truncation (necessary for truncated BPTT) for the adaptation of the coefficients of the same neuron, using recursive formulas instead of non-causal ones as in the truncated BPTT approach.

In other words, the RBP algorithm computes exact gradient, is not local in time (like BPTT) but has the advantage that can be efficiently implemented on-line (CRBP or TRBP) at approximately the same cost, with a parameter that controls the trade-off between exactness of the gradient and computational time. With respect to RTRL the proposed CRBP or TRBP algorithms have the advantage to be local in space and in time while RTRL is not local in space.

The differences between CRBP and TRBP arise in how the batch RBP algorithm is approximated into an on-line one and are highlighted in Section 4.4.

Of course the gradient calculation techniques developed in this chapter, as in all the Doctorate thesis, can be implemented in second order methods such as conjugate gradient or Kalman filter (Puskorius and Feldkamp 1994) but this is beyond the aim of this chapter, please refer to chapter 8.

In the following the Recursive Back Propagation batch learning algorithm and its on-line versions are derived for the MLP with IIR synapses; the formulas for LF-MLNs and AR-MLP are in the appendix.

4.2 THE BATCH RECURSIVE BACK-PROPAGATION (RBP) ALGORITHM

An IIR-MLP contains in each synapse a linear filter with poles and zeros, which are the Auto Regressive (AR) and Moving Average (MA) part respectively. Due to the complexity of the resulting structure, a rigorous notation is needed, where each index is explicitly written. This notation, which is a generalization of that used in (Widrow and Lehr 1990) for static MLP and in (Wan 1990, Benvenuto et al. 1994) for FIR-MLP, is appropriate in this case where complex architectures of different kinds are defined and compared.
4.2.1 Notation

$M$  
number of layers in the network.

$l$  
layer index. In particular $l=0$ and $l=M$ denote the input and output layer, respectively.

$N_l$  
number of neurons of the $l$-th layer. In particular $N_0$ and $N_M$ denote the number of inputs and outputs respectively.

$n$  
neuron index.

$t$  
time index. $t=1, 2, ..., T$. Where $T$ is the length of the training sequence.

$x_n(l)[t]$  
output of the $n$-th neuron of the $l$-th layer, at time $t$. In particular $n=0$ refers to the bias inputs: $x_0(l)=1$. Note that $\{x_n^{(0)}[t]\}, n=1, ..., N_0$, are the input signals.

$L_{nm}^{(l)}$  
order of the MA part of the synapse of the $n$-th neuron of the $l$-th layer relative to the $m$-th output of the $(l-1)$-th layer. $L_{nm}^{(l)} \geq 1$ and $L_{n0}^{(l)}=1$.

$I_{nm}^{(l)}$  
order of the AR part of the synapse of the $n$-th neuron of the $l$-th layer relative to the $m$-th output of the $(l-1)$-th layer. $I_{nm}^{(l)} \geq 0$ and $I_{n0}^{(l)}=0$.

$w_{nm}^{(l)}(p)$  
coefficient of the MA part of the corresponding synapse. If $L_{nm}^{(l)}=1$, the synapse has no MA part and the weight notation becomes $w_{nm}^{(l)}$. $w_{n0}^{(l)}$ is the bias.

$v_{nm}^{(l)}(p)$  
coefficient of the AR part of the synapse. If $I_{nm}^{(l)}=0$ the synaptic filter is purely MA.

$\text{weight}$  
either a $w$ or $v$ coefficient.

$\text{sgm}(z)$  
activation function.

$\text{sgm}'(z)$  
derivative of $\text{sgm}(z)$.

$y_{nm}^{(l)}[t]$  
synaptic filter output at time $t$ relative to the synapse of $n$-th neuron, $l$-th layer and $m$-th input. $y_{n0}^{(l)}=w_{n0}^{(l)}$ is the bias.

$s_n^{(l)}[t]$  
'net' quantity relative to the $n$-th neuron of the $l$-th layer at time $t$, i.e. the input to the corresponding activation function.

$d_n[t]$  
$(n=1, ..., N_M)$ desired outputs at time $t$.

To further clarify the notation a simple two layer ($M=2$) IIR-MLP, with two inputs ($N_0=2$), one hidden neuron ($N_1=1$) with no MA and AR parts in each
synapse \((L_{1m}^{(1)}=1 \text{ and } I_{1m}^{(1)}=0 \text{ for } m=1, 2)\) and one output neuron \((N_2=1)\) with both MA and AR parts in the synapses \((L_{11}^{(2)}=3 \text{ and } I_{11}^{(2)}=2)\), is shown in Fig. 4.2.1.

![Diagram](image)

**Fig. 4.2.1** A simple example of IIR-MLP network (the bias terms are not shown).

### 4.2.2 The forward phase

The forward phase at time \(t\) can be described by the following equations evaluated for \(l=1,...,M\) and \(n=1,...,N_l\):

\[
\begin{align*}
\sum_p W^{(l)}_{nm(p)} x^{(l-1)}_{m} [t - p] &+ \sum_p V^{(l)}_{nm(p)} y^{(l)}_{nm} [t - p] \\
\sum_m y^{(l)}_{nm} [t] &+ s^{(l)}_n [t] = \text{sgm}(s^{(l)}_n [t])
\end{align*}
\]

For equation (4.2.1), the direct form I of the IIR filter has been used (Shynk 1989), but other structures exist. In particular, direct form II structures allow reduction in the storage complexity as well as in the number of operations, both in forward and backward computation (see Section 4.6). For the sake of clarity the expression corresponding to (4.2.1) in the IIR filter usual notation (Shynk 1989) is reported:

\[
\begin{align*}
y[t] &\sum_p W_p [t] x[t - p] + \sum_p V_p [t] y[t - p] \\
x_n^{(l)} [t] &\sum_m y^{(l)}_{nm} [t]
\end{align*}
\]
\[ y[t] = \left( \frac{B(t, q)}{1 - A(t, q)} \right) x[t] \]  \hspace{1cm} (4.2.4)

where

\[ A(t, q) = \sum_{p=1}^{N-1} v_p[t] q^{-p} \quad \text{and} \quad B(t, q) = \sum_{p=0}^{M-1} w_p[t] q^{-p} \]  \hspace{1cm} (4.2.5)

where \( q^{-1} \) is the delay operator, i.e. \( q^{-1} s[t] = s[t - 1] \).

In this case the dependence of the coefficients upon time is explicitly stated with the index \( t \), since we are considering adaptive filters that are adapted every time step. However, for neural networks in order to reduce the complexity of the notation the explicit indication of time will not be used.

### 4.2.3 The learning algorithm (RBP)

The instantaneous global squared error at time \( t \) is defined as:

\[ e^2_n[t] = \sum_{n=1}^{N_k} e^2_n[t] \quad \text{with} \quad e_n[t] = d_n[t] - x_n^{(M)}[t]. \]  \hspace{1cm} (4.2.6)

So the global squared error over the whole training sequence is:

\[ E^2 = \sum_{t=1}^{T} e^2[t], \]  \hspace{1cm} (4.2.7)

where \( T \) is the duration of the sequence.

In the most general case, the training set of a dynamic neural network is composed of a certain number of training sequences (runs), and so the error to be minimized is the statistical average of the error \( E^2 \) over all the runs. To simplify the notation, we will consider only one run but the extension is trivial.

Let us define the usual quantities 'backpropagating error' and 'delta':

\[ e^{(l)}_n[t] = -\frac{1}{2} \frac{\partial E^2}{\partial x_n^{(l)}[t]} \quad \text{and} \quad s^{(l)}_n[t] = -\frac{1}{2} \frac{\partial E^2}{\partial s_n^{(l)}[t]} \]  \hspace{1cm} (4.2.8)

As in the static case, it holds:

\[ \delta^{(l)}_n[t] = e^{(l)}_n[t] \cdot \text{sgn}(s^{(l)}_n[t]) \]  \hspace{1cm} (4.2.9)

Therefore, using gradient descent method and the chain rule expansion:

\[ \Delta W^{(l)}_{nm(p)} = -\frac{\mu}{2} \frac{\partial E^2}{\partial W^{(l)}_{nm(p)}} = -\frac{\mu}{2} \sum_{t=1}^{T} \frac{\partial E^2}{\partial s^{(l)}_n[t]} \cdot \frac{\partial s^{(l)}_n[t]}{\partial W^{(l)}_{nm(p)}} \]  \hspace{1cm} (4.2.10)
where \( \mu \) is the learning rate.

The above equation can be rewritten as

\[
\Delta w_{nm(p)}^{(t)} = \sum_{t=1}^{T} \Delta w_{nm(p)}^{(t)} [t + 1],
\]

(4.2.11)

where

\[
\Delta w_{nm(p)}^{(t)} [t + 1] = -\frac{\mu}{2} \frac{\partial E^2}{\partial s_n^{(t)}[t]} \frac{\partial s_n^{(t)}[t]}{\partial w_{nm(p)}^{(t)}} = \mu \delta_n^{(t)}[t] \frac{\partial s_n^{(t)}[t]}{\partial w_{nm(p)}^{(t)}}.
\]

(4.2.12)

Similarly for the \( v \) weights:

\[
\Delta v_{nm(p)}^{(t)} = \sum_{t=1}^{T} \Delta v_{nm(p)}^{(t)} [t + 1],
\]

(4.2.13)

where

\[
\Delta v_{nm(p)}^{(t)} [t + 1] = \mu \delta_n^{(t)}[t] \frac{\partial s_n^{(t)}[t]}{\partial v_{nm(p)}^{(t)}}.
\]

(4.2.14)

Expressions to compute \( \delta_n^{(t)}[t] \) and the derivatives in (4.2.12) and (4.2.14) must be provided.

Differentiating equation (4.2.1) and considering that \( \frac{\partial s_n^{(t)}[t]}{\partial weight_{nm(p)}^{(t)}} = \frac{\partial y_{nm}^{(t)}[t]}{\partial weight_{nm(p)}^{(t)}} \), where 'weight' indicates either \( w \) or \( v \), it follows:

\[
\frac{\partial s_n^{(t)}[t]}{\partial w_{nm(p)}^{(t)}} = x_m^{(l-1)}[t - p] + \sum_{r=1}^{l_p} y_{nm(r)}^{(l)} \frac{\partial s_n^{(t)}[t - r]}{\partial w_{nm(p)}^{(t)}}
\]

(4.2.15)

\[
\frac{\partial s_n^{(t)}[t]}{\partial v_{nm(p)}^{(t)}} = y_{nm}^{(l)}[t - p] + \sum_{r=1}^{l_p} y_{nm(r)}^{(l)} \frac{\partial s_n^{(t)}[t - r]}{\partial v_{nm(p)}^{(t)}}
\]

(4.2.16)

Note that such expressions are the same found in the IIR linear adaptive filter theory (Shynk 1989, equations 16a, 16b):

\[
\frac{\partial y[t]}{\partial w_p} = x[t - p] + \sum_{r=1}^{N-1} v_r \frac{\partial y[t - r]}{\partial w_p} \quad \text{or} \quad \frac{\partial y[t]}{\partial w_p} = \left( \frac{1}{1 - A(t, q)} \right) x[t - p]
\]

(4.2.17)

\[
\frac{\partial y[t]}{\partial v_p} = y[t - p] + \sum_{r=1}^{N-1} v_r \frac{\partial y[t - r]}{\partial v_p} \quad \text{or} \quad \frac{\partial y[t]}{\partial v_p} = \left( \frac{1}{1 - A(t, q)} \right) y[t - p]
\]

(4.2.18)
where the weights time index is not explicitly written, because it can be confusing.

The formulas (4.2.15) and (4.2.16), as in the IIR linear adaptive filter context, are exactly true only if the weights \( w \) or \( v \) are not time-dependent, because the derivatives evaluation point is fixed, or approximately true if they adapt slowly, i.e. the learning rate is sufficiently small (Williams and Zipser 1989, Shynk 1989). In batch RBP the weights update is performed only at the end of the learning epoch using the accumulated weight variations computed at every time instant, so that the above expressions are exact and can be computed recursively starting with null values of the initial derivatives.

Now we want to derive an expression for \( \delta_n^{(l)}[t] \): by (4.2.9) we need to compute \( e_n^{(l)}[t] \); using the chain rule it is possible to obtain:

\[
e_n^{(l)}[t] = \sum_{q=1}^{N_{ls}} \sum_{p=0}^{L_q^{(l+1)}} \delta_q^{(l+1)}[t + p] \frac{\partial y_{qn}^{(l+1)}[t + p]}{\partial x_n^{(l)}[t]} \quad \text{for } l < M
\]  \hspace{1cm} (4.2.19)

By the last expression, under the hypothesis of IIR synaptic filter causality, the internal summation can start from \( k=t \). Then, changing the variables as \( k - t \to p \), using the definition of \( \delta_n^{(l)}[t] \) and considering that for \( l = M \) the derivative can be directly computed, the backpropagation through the layers can be derived:

\[
e_n^{(l)}[t] = \begin{cases} e_q[t] & \text{for } l = M \\ \sum_{q=1}^{N_{ls}} \sum_{p=0}^{L_q^{(l+1)}} \delta_q^{(l+1)}[t + p] \frac{\partial y_{qn}^{(l+1)}[t + p]}{\partial x_n^{(l)}[t]} & \text{for } l = (M-1), \ldots, 1 \end{cases}
\]  \hspace{1cm} (4.2.20)

where the partial derivatives are computed deriving expression (4.2.1):

\[
\frac{\partial y_{qn}^{(l+1)}[t + p]}{\partial x_n^{(l)}[t]} = \left\{ \begin{array}{ll} \frac{w_{qni}^{(l+1)}}{v_{qni}^{(l+1)}} & \text{if } 0 \leq p \leq L_q^{(l+1)} - 1 \\ 0 & \text{otherwise} \end{array} \right\} + \sum_{r=1}^{\min(l^{(l+1)},L_q^{(l+1)})} \frac{\partial y_{qn}^{(l+1)}[t + p - r]}{\partial x_n^{(l)}[t]}
\]  \hspace{1cm} (4.2.21)

These derivatives have a very interesting interpretation. Consider the expression of a generic causal linear filter output as the convolution of the input \( x[\tau] \) with an impulse response \( h[t, \tau] \) (in general time variant):

\[
y[t] = \sum_{\tau=0}^{t} x[\tau] h[t, \tau]
\]  \hspace{1cm} (4.2.22)
Chapter 4 ON-LINE LEARNING FOR LOCALLY RECURRENT NEURAL NETWORKS

where \( t_0 \) is the initial time instant. Differentiating we get:

\[
\frac{\partial y[t+p]}{\partial x[t]} = h[t+p,t]. \tag{4.2.23}
\]

If the learning algorithm updates the coefficients only at the end of the epoch (batch mode adaptation), then the IIR filter is time invariant and

\[
\left( \frac{1}{1 - A(t,q)} \right) \begin{cases} 
  w_p & \text{if } 0 \leq p \leq M - 1 \\
  0 & \text{otherwise}
\end{cases} = \frac{\partial y[t+p]}{\partial x[t]} = h[t+p-t] = h[p]. \tag{4.2.24}
\]

where the operator \( q^{-1} \) now is delaying the \( p \) index and not the \( t \) index, \( h[p] \) is the impulse response of the filter and \( A(t,q) \), previously defined, does not depend on \( t \). Obviously, if the filter is time invariant, the derivative in (4.2.21) does not depend on \( t \). This means that the derivative is obtained through auto-regressive filtering of the sequence of the coefficients of the MA part with the AR part of the corresponding IIR synaptic filter. This is true since for the causal filter the derivative inside the summation in (4.2.21) is zero if \( r > p \), allowing the upper limit of the summation to be written also as \( I_{qn}^{(l+1)} \). If the learning rate is small enough, also when on-line adaptation is performed the derivative is slowly changing in time, i.e. with the \( t \) index in (4.2.21).

Therefore, for MLP with IIR synapses, expression (4.2.20) means that each back propagating error at layer \( l \) is a summation of all the \( \delta \)'s at the following layer filtered by the non-causal version of the respective IIR filter, i.e. filtering by the time reverted impulse response of the synaptic filter.

The expressions (4.2.1), (4.2.2), (4.2.9), (4.2.11), (4.2.12), (4.2.13), (4.2.14), (4.2.15), (4.2.16), (4.2.20), (4.2.21) constitute the RBP algorithm for IIR-MLP. Note that, if all the synapses contain only the MA part \( (I_{nm}^{(l)} = 0 \) for each \( n,m \) and \( l \)\), the architecture reduces to FIR-MLP and this algorithm reduces to the Temporal Back Propagation (TBP; batch mode) as in (Wan 1990, Benvenuto et al. 1994, Back et al. 1994, Haykin 1994). Obviously, if all the synaptic filters have no memory \( (I_{nm}^{(l)} = 0 \) and \( L_{nm}^{(l)} = 1 \) for each \( n,m \) and \( l \)\), this algorithm gives standard Back Propagation (batch adaptation) for the MLP. Moreover the on-line versions of TBP and BP are obtained as particular cases of CRBP.

Fig. 4.2.2 shows the diagram of the RBP applied to the simple IIR-MLP example of Fig. 4.2.1, with a simplification of the recursive computation of derivatives, as explained in Section 4.6. These are the steps of the algorithm for each learning epoch:

- perform forward pass for the entire input sequence saving the states of the network at all times, using expressions (4.2.1), (4.2.2);
Chapter 4 ON-LINE LEARNING FOR LOCALLY RECURRENT NEURAL NETWORKS

- start the backward pass computing the error for all the outputs and time instants;
- compute the derivatives in (4.2.21) recursively with null initial conditions;
- for \( l = M \) to 1
  - compute \( e_n^{(l)}[t] \) by (4.2.20) \( \forall t \in [1, T] \);
  - compute \( \delta_n^{(l)}[t] \) by (4.2.9) \( \forall t \in [1, T] \);
  - compute the weights variations using expressions (4.2.11÷4.2.16);
  - update weights.

Since the RBP recursive expressions (4.2.15), (4.2.16), and (4.2.21) have the same feedback coefficients as the corresponding IIR filter in the forward expression (4.2.1), the learning algorithm calculation will be stable if all the IIR filters are stable.

\[
\begin{align*}
\Delta w_{11}^{(1)}[t+1] &= sgm(x_1[t] + B(t,q_{10})y_1[t]) - y_1[t] \\
\Delta w_{12}^{(1)}[t+1] &= sgm(x_1[t] + B(t,q_{10})y_1[t]) - y_1[t] \\
\Delta w_{11}^{(2)}[t+1] &= sgm(x_1[t] + B(t,q_{10})y_1[t]) - y_1[t] \\
\Delta w_{12}^{(2)}[t+1] &= sgm(x_1[t] + B(t,q_{10})y_1[t]) - y_1[t] \\
\Delta w_{11}^{(0)}[t+1] &= sgm(x_1[t] + B(t,q_{10})y_1[t]) - y_1[t] \\
\Delta w_{12}^{(0)}[t+1] &= sgm(x_1[t] + B(t,q_{10})y_1[t]) - y_1[t] \\
\Delta w_{11}^{(2)}[t+1] &= sgm(x_1[t] + B(t,q_{10})y_1[t]) - y_1[t] \\
\Delta w_{12}^{(2)}[t+1] &= sgm(x_1[t] + B(t,q_{10})y_1[t]) - y_1[t] \\
\end{align*}
\]

**Fig. 4.2.2** The RBP applied to the IIR-MLP example of Fig. 4.2.1 (the bias terms are not shown). It is obtained assuming batch mode (no causalization or truncation) and simplified recursive computation of derivatives.
4.3 ON-LINE ALGORITHMS: CAUSAL RECURSIVE BACK-PROPAGATION (CRBP)

As previously stated, the RBP algorithm is used only as an intermediate step in the derivation of CRBP. In fact, as shown by expression (4.2.20), the exact RBP algorithm is non-causal, since the \( e^{(l)}_n \) at time \( t \) depends on the \( \delta^{(l+n)}_n \) quantities, taken at future time instants. Therefore the weights update can only be performed in batch mode.

However the RBP algorithm, due to the recursive structure of expressions (4.2.15) and (4.2.16), can be easily approximated to obtain a very efficient on-line learning algorithm. The on-line approximation consists in three steps: (a) incremental instead of cumulative adaptation, (b) future convolution truncation, (c) causalization. In this section, the IIR-MLP case is fully considered. The extension to the other LF-MLN architectures is easy, and so we only will explain some non obvious differences.

(a) Incremental instead of cumulative adaptation can be implemented using:

\[
weight_{\text{variation}}^{(l)}_{nm(p)}[t + 1] = \Delta weight^{(l)}_{nm(p)}[t + 1] \tag{4.3.1}
\]

instead of (4.2.11) and (4.2.13), at each time step, where 'weight' indicates either \( w \) or \( v \). This is the approximation that is usually proposed in literature for dynamic networks. However a less obvious choice could be done:

\[
\Delta weight^{(l)}_{nm(p)} = \mu \delta^{(l)}_n[t] \frac{\partial s^{(l)}_n[t]}{\partial weight^{(l)}_{nm(p)}} + \sum_{\tau=1}^{t} \Delta weight^{(l)}_{nm(p)}[\tau + 1]. \tag{4.3.2}
\]

This expression can be computed recursively (with an approximation) that is equivalent to using a momentum term with momentum parameter equal to one. In fact the momentum formula is:

\[
weight_{\text{variation}}^{(l)}_{nm(p)}[t + 1] = \alpha weight_{\text{variation}}^{(l)}_{nm(p)}[t] + \mu \delta^{(l)}_n[t] \frac{\partial s^{(l)}_n[t]}{\partial weight^{(l)}_{nm(p)}} \tag{4.3.3}
\]

where \( \alpha \) is the momentum parameter that in general can be chosen in the range: \( 0 \leq \alpha \leq 1 \). Here we will consider only the incremental adaptation without momentum.

(b) If a causalization is desired, a truncation of the future convolution is necessary, due to the infinite memory of the IIR synapses. The truncated formula is therefore:
where $Q_{l+1}$ is appropriately chosen. In the particular case when in a given layer $I_{nm}^{(l)}=0$ for each $n,m$ (i.e. the synapses have finite memory) it is useful to choose $Q_l = \max(L_{nm}^{(l)} - 1)$. In this way, $Q_l$ is set to the maximum memory of the synaptic filters of the layer and no real truncation of the filter response is implemented as in Wan’s Temporal Back Propagation (Wan 1990, Haykin 1994).

c) Then we have to introduce a suitable number of delays in the weight adaptation formulas in order to remove the non-causality. In other words:

$$weight^{(l)}[t+1] = weight^{(l)}[t] + weight \_ variation^{(l)}[t+1-D_l]$$

(4.3.5)

where $D_l$ is a suitable integer number. It follows that

$$D_l = \begin{cases} 0 & \text{if } l = M \\ \sum_{i=1}^{M} Q_i & \text{if } 1 \leq l < M \end{cases}$$

(4.3.6)

The causalized formula can be obtained from (4.3.4) reversing the order of the internal summation and issuing the variable change $t + Q_{l+1} \rightarrow \tau$, where $\tau$ is the current time instant (present):

$$e_n^{(l)}[\tau - Q_{l+1}] = \sum_{q=1}^{N_{l+1}} \sum_{p=0}^{Q_{l+1}} \delta^{(l+1)}_{q}[\tau-p] \frac{\partial y_{qn}^{(l+1)}[\tau-p]}{\partial x_n^{(l)}[\tau]}$$

for $l = (M-1),...,1$ (4.3.7)

In expressions (4.3.4) and (4.3.7), the trivial hypothesis $\delta[t] = 0$ for $t \notin [1,T]$ has been done. Expression (4.3.7) is now causal since it is evaluated at time $\tau$ from delta’s up to time $\tau$. The impulse response computed by (4.2.21) is used reversing the time scale since $\frac{\partial y_{qn}^{(l+1)}[t-p]}{\partial x_n^{(l)}[t-Q_{l+1}]} = h[Q_{l+1} - p]$ in the time invariance hypothesis (or in that approximation), whereas backpropagating errors and delta’s are used in the normal time scale, as for standard convolution.

The result is not assigned to the backpropagating error at time $\tau$ but at time $\tau - Q_{l+1}$, as dictated by expression (4.3.7). For the sake of clarity, a diagram of CRBP is shown in Fig. 4.3.1.
Fig. 4.3.1 The CRBP applied to the IIR-MLP example of Fig. 4.2.1 (the bias terms are not shown). It is obtained assuming on-line mode (with truncation and causalization) and simplified recursive computation of derivatives. The Back-Tsoi approximation uses the same flow diagram but with a multiplication by $w_{11}(0)$ instead of the truncated IIR filtering and $Q_0=0$.

The causalization and the on-line update, compared to the batch mode case, is not a strong approximation if the learning rate is small enough, because in this case the weights variation is small in the time interval of $D_l$ instants. Instead the truncation approximation can be justified by the following property:

• If a linear time invariant IIR filter is asymptotically stable (i.e. all the poles are inside the unit circle) then $\frac{\partial y[t+p]}{\partial x[t]} \rightarrow 0$ if $p \rightarrow \infty$ where $y[t]$ is the output of the filter and $x[t]$ the input at time $t$. 
The proof can be done in two ways that are both interesting. The first is just considering that the derivative is the impulse response of the filter that must go to zero in the stable case. The second is considering that in (4.2.21) and (4.2.24) the recursion coefficients are the same of the corresponding IIR filter therefore their poles must lie inside the unit circle for stability, i.e. the derivative goes to zero as $p \to \infty$. The second reasoning is more general and it is interesting because it shows a way to verify the validity of the truncation hypothesis for any locally recurrent network architecture, e.g. output feedback MLN. For the derivative to go to zero it is necessary and sufficient that the feedback coefficients in the calculation of (4.2.21) or the corresponding expressions in the appendix, give poles inside the unit circle.

Moreover it is well known that impulse responses of stable rational transfer functions have an exponentially decaying behavior. This means that the truncation parameter can be chosen quite small. This fact was confirmed by the simulations even if they show that setting the truncation parameter to zero is too strong an approximation and should be avoided.

The previous property can be used to automatically select the desired truncation parameter $Q_{\tau+1}$ by taking into account the impulse response explicitly computed by the algorithm. In the subsequent discussions this possibility is not further investigated, since for the selected problems a good choice of the truncation parameter was within a very small range.

The condition

$$\lim_{p \to \infty} \frac{\partial y[t+p]}{\partial x[t]} = 0$$

and $x[t]$ an input of a recurrent neuron (Frasconi et al. 1992, Hochreiter and Schmidhuber 1995) holds by definition for each neuron in IIR-MLP, activation-output feedback MLNs and AR-MLP in case each neuron exhibits forgetting behavior. Instead in case of latching behavior (possible only for output feedback MLN), that derivative does not go to zero and the corresponding linear system is unstable (Frasconi et al. 1992). In this case, the truncation of the internal summation in (4.A.20') (see the Appendix) can be too strong an approximation. However, it should be considered that the advantages of networks with local feedback over fully connected ones is especially in modeling a forgetting behavior. Latching behavior is outside normal working conditions for locally recurrent networks.

The algorithm proposed by Back and Tsoi (1991) that is the only on-line learning algorithm proposed for locally recurrent networks with no architectural restriction, can be seen as a particular case of the approximation here proposed if a strong truncation of the summation is assumed: $Q_{\tau+1} = 0$ for each $\tau$. The simplified formula is:
In this way the backpropagation is considering only the instantaneous influence of the IIR filter input to the output (the coefficient \( w_{nm}(0)^{(l+1)} \)). Hence in the scheme of Fig. 4.3.1, the truncated IIR filter should be replaced by a simple multiplication for \( w_{11}(0)^{(2)} \). No causalization is needed (being \( D_l=0 \) for each \( l \)) and the algorithm is very simple. However we will show that with the inclusion of only few additional memory terms in the backpropagation \((Q_{l+1}>0)\) it is possible to reach much better stability and speed of convergence.

The BPS algorithm also (on-line mode) (Gori et al. 1989) can be obtained as particular case of CRBP under the architectural restriction that the dynamic units can only be placed in the first layer. In this case, CRBP implements no real truncation of past history as BPS does. The CRBP applied to AR-MLP can be also viewed as a generalization of Leighton-Conrath’s work (1991), although their formulas do not exactly match the expression corresponding to (4.3.8) for the AR-MLP.

### 4.4 On-line Algorithms: Truncated Recursive Back-Propagation (TRBP)

For the derivation of the TRBP algorithm, first the RBP algorithm must be causalized. This is provided by choosing the squared error computed at the current time instant \((\tau)\) in the following and not over all the sequence, so that the gradient descent can be expressed by:

\[
\Delta \text{weight}_{nm(p)}^{(l)} = \frac{\mu}{2} \left( \frac{dE(\tau, \tau)}{d \text{weight}_{nm(p)}^{(l)}} \right) = \sum_{t=1}^{\tau} \Delta \text{weight}_{nm(p)}^{(l)}[t+1]
\]

(4.4.1)

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

In other words, causalization can be obtained simply substituting the final instant \( T \) of the sequence with the current instant \( \tau \) in (4.2.7), (4.2.10), (4.2.13) and in (4.2.20). In this way, the upper index of the inner summation in (4.2.20) is the current time minus the index \( t \) of the summation in (4.4.1). So every time step, expression (4.2.20) is evaluated for \( t=1 \) to \( \tau \) using \( \delta_q^{(l+1)}[.] \) up to time \( \tau \) for each time instant, so that the calculation is causal.
Since this implementation would require a memory and computational complexity, for each iteration, that grows with time, a forgetting mechanism must be implemented for on-line training. Forgetting the old past history is very reasonable and recommended by many authors, e.g. (Williams and Peng 1990). The obtained performance are competitive (Williams and Peng 1990) even with the RTRL algorithm that computes a slightly approximated gradient of the instantaneous error (Williams and Zipser 1989). The only expression that must be changed to transform RBP into TRBP is (4.4.1) (that substitutes (4.2.11) and (4.2.13)) that becomes:

$$\Delta \text{weight}^{(l)}_{m(p)} = \sum_{t=-h+1}^{\tau} \Delta \text{weight}^{(l)}_{m(p)} [t+1]$$  \hspace{1cm} (4.4.2)

where $h$ is the length of the considered history buffer.

Now the algorithm can be implemented on-line with parameters changes computed by (4.4.2), performed every time step. It can be shown that since we are considering an instantaneous error cost function, the error injection (Williams and Peng 1990) should be performed only for the current time step so that in this case, for the last layer, $e_{n}^{(l)}[t]$ is zero for $t<\tau$; this simplifies the calculation of expression (4.4.2) for the last layer since the corresponding delta is also zero and so only one term remains in (4.4.2) when $l=M$. For the same reason, expression (4.2.20) can be easily computed for $l=M-1$ since again only one term remains in the inner summation.

To simplify the algorithm a modification can be implemented: updating the coefficients every $h'$ time steps instead of every step. This reduces the computational complexity basically by a factor of $h'$; if $h-h'$ is large enough ($h \geq h'$ always) the approximation gives good performance (Williams and Peng 1990). However, when applications that require adaptation every time instant are involved, (e.g. on-line tracking of fast varying systems), $h'$ must be chosen equal to one, of course.

For the general method with $h'>1$, the error injection must be performed for the time indexes $t$ from $\tau-h'+1$ to $\tau$ so that more terms are present in the summations in (4.2.20) and (4.4.2) and not only one as previously explained. This corresponds to compute the gradient of $E(\tau-h'+1,\tau)$.

Even when $h'=1$, if a cost function smoother than the instantaneous error is desired such as $E(\tau-N_c+1,\tau)$ (with $N_c$ appropriately chosen and constrained to be $h' \leq N_c \leq h$), the error injection can be performed for all $t \in [\tau-N_c+1,\tau]$. In this section is being assumed $N_c=1$, but the extension is trivial.

The algorithm defined in this way will be named Truncated Recursive Back Propagation (TRBP).
To improve the accuracy, when IIR-MLP or locally recurrent layered networks are considered, it can be useful to make $h$ depending on the layer $l$ in (4.4.2) and choose it approximately equal to one plus the summation of the maximum memory of the synaptic filters (roughly estimated) from layer $l+1$ to the last one, i.e.:

$$h_l = \begin{cases} 
  h' & \text{if } l = M \\
  1 + \sum_{i=l+1}^{M} Q_i & \text{if } 1 \leq l < M 
\end{cases} \quad (4.4.3)$$

where $Q_i$ is an estimate of the maximum memory for the filters in layer $i$. This choice is reasonable since the truncation of (4.4.1) depends on the value $\delta_n^{(l)}[\tau]$ varying $\tau$ that in turn depends on the memory of the synaptic filters of layers from $l+1$ to $M$.

It is possible to give a formal condition under which the truncation of (4.4.1) is feasible:

$$\lim_{h \to \infty} \delta_n^{(l)}[\tau - h] = \lim_{h \to \infty} -\frac{1}{2} \frac{\partial E(\tau, \tau)}{\partial \delta_n^{(l)}[\tau - h]} = 0 \quad \text{for each } n, \tau \quad (4.4.4)$$

or equivalently

$$\lim_{h \to \infty} \frac{\partial x_k^{(M)}[\tau]}{\partial \delta_n^{(l)}[\tau - h]} = 0 \quad \text{for each } k, n, \tau \quad (4.4.5)$$

This condition is satisfied for IIR-MLP if all the IIR filters of an arbitrary chosen layer from the $l$-th to the last one are stable or more in general for locally recurrent multilayered networks if all the neurons in an arbitrary layer after the $l$-th exhibit forgetting behavior (Frasconi et al. 1992) (the proof can be given by the chain rule). This condition is different from that which allows the truncation needed by CRBP (in formula (4.2.20)) since in that case the stability of the filters of all layers after the first was needed. Moreover, only for the sake of truncation of (4.4.1), the previous condition can be relaxed to the following: at least one branch in every path from one arbitrary neuron of layer $l$-th to a neuron of the last layer must be a stable IIR filter. Anyway, for the overall TRBP algorithm all the synaptic IIR filters in the network are required to be stable; in fact the previous analysis seem to indicate a smaller sensitivity to truncation of TRBP than CRBP if $M>2$.

A difference with the approximation proposed on CRBP is that more terms than the current time coefficient variations are computed and accumulated for each iteration, potentially with a better approximation of the true gradient.
The new TRBP method is simpler than the previous one (CRBP) if \( h/h' \) is chosen reasonably close to one (with the limitation that \( h-h' \) should be large enough for accuracy). A good choice can be \( h=2h' \) with \( h' \) large enough.

It is interesting to note that TRBP with the previous choice for \( h \) using \( Q_i \) equal to the maximum FIR filter order of layer \( i \)-th and \( h'=1 \) gives as particular case the algorithm proposed by Waibel et al. (1989) for MLP with FIR synapses (or TDNN). It has a nice geometric interpretation since it is obtained applying static BP to the network unfolded in time replicating network substructure for each delay. The resulting network has no internal memory but the inputs are the original ones and their delayed versions. The present formulation is much more general since it allows feedback in the network.

Moreover TRBP with \( h=h'=1 \) gives the Back and Tsoi algorithm (1991) for IIR-MLP. Of course, if all the synaptic filters have no memory (the IIR-MLP becomes a standard MLP) and \( h'=1 \) TRBP particularizes to standard on-line BP (\( h=1 \) for each \( l \) according to (4.4.3)). Even BPS for LF-MLN (Frasconi et al. 1992) is a particular case of TRBP.

### 4.5 EXPERIMENTAL RESULTS

The simulation results reported here are of two kinds. The first is a comparison of different locally recurrent architectures with more traditional dynamic MLP using a fixed training method for each one. The second and more important simulation is the comparison of runs of CRBP with different values of the truncation parameter and of CRBP with other training methods, keeping fixed the architecture. Each of these two simulations was run on two different system identification tests. The TRBP algorithm showed similar performance to CRBP if the truncation parameter is the same \( h = Q_i \forall i, h'=1 \).

Many simulations were performed on three locally recurrent architectures, i.e. IIR-MLP, activation and output feedback MLN (shown in Section 2.4) while the AR-MLP was not implemented. For comparison purposes, also two traditional neural networks were tested, namely the static MLP with input and possibly output buffer, (shown in Section 2.2), and the FIR-MLP (Section 2.2). The results reported here refer to two problems of identification of non-linear dynamical systems.

The number of delays for the five architectures (i.e. buffers lengths for the buffered MLPs, MA orders for the FIR-MLP, MA and AR orders for the locally recurrent networks) was chosen in order to obtain the best performance (approximately) for each network, while the total number of free parameter was fixed (40 parameters, bias included), as shown in Table 4.5.1.
### Table 4.5.1

<table>
<thead>
<tr>
<th>Neural architecture used</th>
<th>Experiment 1: Back and Tsoi system</th>
<th>Experiment 2: PAM system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffered MLP (STAT)</td>
<td>5 input delays; 4 feedback delays (tot. 11 inputs)</td>
<td>10 input delays; no feedback (tot. 11 inputs)</td>
</tr>
<tr>
<td>FIR-MLP (FIR)</td>
<td>hidden MA-AR: 5-0; output MA-AR: 5-0</td>
<td>hidden MA-AR: 5-0; output MA-AR: 5-0</td>
</tr>
<tr>
<td>IIR-MLP (IIR)</td>
<td>hidden MA-AR: 2-3; output MA-AR: 2-3</td>
<td>hidden MA-AR: 3-2; output MA-AR: 3-2</td>
</tr>
<tr>
<td>Activation Feedback MLN</td>
<td>hidden MA-AR: 2-3; output MA-AR: 2-3</td>
<td>hidden MA-AR: 4-3; output MA-AR: 2-3</td>
</tr>
<tr>
<td>(ACT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output Feedback MLN (OUT)</td>
<td>hidden MA-AR: 4-3; output MA-AR: 2-3</td>
<td>hidden MA-AR: 4-3; output MA-AR: 2-3</td>
</tr>
</tbody>
</table>

Table 4.5.1 *Number of delays for the different buffers used for all the neural networks in both the system identification experiments. For the three locally recurrent architectures the number of delays for the Moving Average part (input buffer) and the Auto Regressive part (feedback buffer) for the hidden and output layers are specified. All the neural networks have the same number of adaptable parameters, i.e. 40 bias included, with the only exception of the Activation Feedback MLN used for the Back and Tsoi system that has 34 parameters. Inside brackets the short names used in Table 4.5.2 and Fig.s 4.5.1 and 4.5.4.*

All the used networks had two layers, three hidden neurons with hyperbolic tangent activation function, and one linear output neuron. Three different learning algorithms were used: standard static backpropagation for buffered MLP (with open loop approximation if feedback is present), temporal backpropagation for FIR-MLP (Wan 1990) and the proposed CRBP algorithm for the locally recurrent networks. Momentum term and adaptive learning rate were not used. The results are given in terms of Mean-Square-Error (MSE), expressed in dB, computed on the learning set after each epoch (after all the input-output samples were presented) and averaged over 10 runs, each with a different weights initialization. Obviously, after each iteration, the state of the network was reinitialized.

### 4.5.1 First simulation: Back-Tsoi system identification

The first set of experiments consists in identifying the non-linear system with memory presented by A.D. Back and A.C. Tsoi in (Back and Tsoi 1993). This system is described by the following input-output relationship:
\[
\begin{align*}
\{ z(t) &= 0.0154x(t) + 0.0462x(t-1) + 0.0462x(t-2) + 0.0154x(t-3) + \\
&\quad + 1.99z(t-1) - 1.572z(t-2) + 0.4583z(t-3) \\
y(t) &= \sin[z(t)]
\end{align*}
\] (4.5.1)

where \(x[t]\) and \(y[t]\) are the input and output signals at time \(t\) respectively. A sequence \(x[t]\) of 1000 points of white random noise, with a uniform probability density function between -1 and +1, was generated. This sequence and the corresponding output sequence \(y[t]\) constituted the training set for the various neural networks.

From Fig. 4.5.1, it is evident that the locally recurrent MLPs exhibit much better modeling capabilities than the static MLP or FIR-MLP, and much better accuracy (asymptotic MSE) too.

The learning was stopped at 200 iterations chosen as a reasonable number of iterations, for each architecture. This simulation (and the corresponding one in the second set of experiments) is included basically to justify the choice of this test problem for comparing learning algorithms for locally recurrent neural networks. Since locally recurrent networks perform better than traditional ones, on this problem, it follows that it is a reasonable choice to test the CRBP learning algorithm; the same is true for the second problem. Fig. 4.5.2 shows
the performance of the CRBP learning algorithm with different values of the truncation parameters $Q_2$ ($Q_2=0$ is the Back-Tsoi algorithm). It is clear that the CRBP performs much better than the previous algorithm even with small $Q_2$, i.e. with a small number of recursive terms. A very small truncation parameter is required to obtain good performance, while increasing it over a certain small range does not change the MSE appreciably. The BPS algorithm was not included in the comparison because it is not applicable to the architecture selected, since the output neuron is dynamic.
4.5.2 Second simulation: 16-PAM system identification

The second set of experiments was carried out on the more realistic problem of identifying a base-band equivalent PAM transmission system in the presence of a non-linearity (Proakis 1989). The pulse shaping circuit transforms the discrete-time symbols stream $a[n]$ into a continuous-time signal $v(t)$ (PAM) by a filter with a raised-cosine shape and roll-off factor $\alpha$. The signal $v(t)$ is then processed by the High-Power-Amplifier (HPA) which is modeled here by the following input-output relationship:

$$w[t] = \frac{2 v[t]}{1 + v^2[t]}.$$  \hfill (4.5.2)

The peak power of the input signal $v[t]$ is set to the value of $\beta$ dB (back-off factor), with $\beta=0$ dB being the normalized unit power. The HPA output $w[t]$ is corrupted by an additive white Gaussian noise $z[t]$, producing the final signal $y[t]$ with a given Signal-To-Noise ratio (SNR), see Fig 4.5.3. The overall system is clearly dynamic and non-linear.
A neural network approach to equalize this system has already been proposed in the technical literature (see for example (Chen et al. 1990)). In our experiments, a neural network was used instead to identify a sampled version of the system. For this purpose, \{a[n]\} was chosen to be a random sequence of 512 symbols drawn from a 16-symbols alphabet. The pulse shaping filter had a roll-off factor \(\alpha = 0.3\), and the HPA back-off \(\beta\) was set to -2 dB. The noise level was very low: SNR=80 dB.

By using an over-sampling ratio of four at the output with respect to the symbol rate, the sequences \{a[n]\} of 512 symbols and \{y[t]\} of 2048 samples were used as the learning set and again the MSE was computed after all the 512 input symbols (epoch) were presented.

Fig. 4.5.4 shows the performance of the five neural architectures: again the locally recurrent MLPs perform much better than the two conventional MLPs in modeling the system.

Simulations reported in Fig. 4.5.5 show that CRBP is much faster, stable and also more accurate than the Back-Tsoi algorithm that sometimes does not converge. Again, a very small truncation parameter of CRBP is required to obtain good performance, while increasing it over a certain, small, range does not change the MSE appreciably.
Fig. 4.5.4 Convergence performance of various neural network architectures on identifying the 16-PAM transmission system. STAT is buffered MLP, FIR is FIR-MLP, IIR is IIR-MLP, ACT is activation feedback MLN, OUT is output feedback MLN. Learning rate $\mu=0.01$. Truncation parameter $Q_2=10$. Plots are averaged over ten runs with different weight initializations.
Fig. 4.5.5 Convergence performance of locally recurrent networks trained by CRBP with various values of the truncation parameter ($Q_2$) on identifying the 16-PAM transmission system. ($Q_2=0$ gives the Back-Tsoi algorithm). Learning rate $\mu=0.003$. Results for IIR-MLP in (a), activation feedback MLN in (b), output feedback MLN in (c). The MA and AR orders were chosen respectively as 2 and 3 for both the hidden and output layers of all the networks. Plots are averaged over ten runs with different weight initializations.

Generalization tests for the identification of the 16-PAM system were made using an input symbol sequence different from the one used for learning. Results are given in Table 4.5.2 and 4.5.3; they show a MSE on the test set very close to the one obtained using the learning set, for all the dynamic architectures and for various choices of the truncation parameter.
### Table 4.5.2

<table>
<thead>
<tr>
<th>MLP type</th>
<th>Asymptotic Learning MSE [dB]</th>
<th>Testing MSE [dB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAT</td>
<td>-20.04</td>
<td>-20.09</td>
</tr>
<tr>
<td>FIR</td>
<td>-19.27</td>
<td>-19.43</td>
</tr>
<tr>
<td>IIR</td>
<td>-23.85</td>
<td>-24.39</td>
</tr>
<tr>
<td>ACT</td>
<td>-24.39</td>
<td>-24.5</td>
</tr>
<tr>
<td>OUT</td>
<td>-23.18</td>
<td>-23.32</td>
</tr>
</tbody>
</table>

Table 4.5.2 Generalization performance of various locally recurrent neural networks on identifying the 16-PAM transmission system. STAT is buffered MLP, FIR is FIR-MLP, IIR is IIR-MLP, ACT is activation feedback MLN, OUT is output feedback MLN. The architectures and parameter setting are the same as for the results reported in Fig. 4.5.4. Results are averaged over 10 runs with different weights initializations.

### Table 4.5.3

<table>
<thead>
<tr>
<th>Truncation Parameter (Q2)</th>
<th>Asymptotic Learning MSE [dB]</th>
<th>Testing MSE [dB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Back-Tsoi alg.)</td>
<td>-18.0</td>
<td>-18.0</td>
</tr>
<tr>
<td>2</td>
<td>-19.0</td>
<td>-19.5</td>
</tr>
<tr>
<td>3</td>
<td>-19.2</td>
<td>-19.1</td>
</tr>
<tr>
<td>4</td>
<td>-20.7</td>
<td>-19.4</td>
</tr>
<tr>
<td>10</td>
<td>-20.8</td>
<td>-19.7</td>
</tr>
<tr>
<td>20</td>
<td>-20.9</td>
<td>-19.6</td>
</tr>
</tbody>
</table>

Table 4.5.3 Generalization performances on the identification of the 16 PAM system of CRBP with different values of the truncation term Q2, for a specific IIR-MLP, having the same architecture as for the results reported in Fig. 4.5.5a. Results are averaged over 10 runs with different weights initializations.

For the sake of completeness, the learning performances and the computational complexities have been studied including truncated BPTT and RTRL derived as in Chapter 5 and 6 respectively (here named BC and FC respectively). Fig. 6.4.2 shows that CRBP, RTRL and truncated BPTT are substantially equivalent and the Back-Tsoi method is much worst. With this regard it should be considered that for implementation reasons the truncated BPTT is implemented with weights values at the proper time step for each time in the past history
considered (Williams and Peng 1990), while in CRBP they are set as the ones at the current time step, saving further complexity. Moreover, RTRL and truncated BPTT are more complex than CRBP with regard to arithmetic operations, as explained in the following section, and moreover CRBP compared to RTRL has the advantage to be local in space.

4.6 ANALYSIS OF COMPLEXITY AND IMPLEMENTATION ISSUES

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 epochwise BPTT or \( O(n^2 h) \) for truncated BPTT (Williams and Peng 1990) (where \( h \) is the past temporal depth) compared with \( O(n^4) \) for RTRL. The memory requirement is \( O(nh) \) for epochwise BPTT (in this case \( h \) is the epoch length) or truncated BPTT, and \( O(n^3) \) for RTRL. Therefore, as far as computational complexity is concerned, in batch mode BPTT is significantly simpler than RTRL, whereas in on-line mode the complexity and also the memory requirement ratios depend on \( n^2/h \). Therefore if \( n^2 \) is large enough compared to \( h \), that is usually the case, then RTRL will be more complex than truncated BPTT.

The recursion implemented in RBP is less efficient with respect to the calculation in BPTT, so for batch mode, BPTT should be preferred, unless the memory requirement is an issue. In this case, RTRL can be preferred since for long sequences it requires less memory.

The advantage of RBP is well evident in on-line mode. In this case the proposed CRBP algorithm is more efficient than truncated BPTT, since it has a better scaling of the number of operations increasing the respective truncation parameter, i.e. \( Q_2 \) and \( h \).

The explanation of this fact is intuitive and detailed by the analysis of complexity. While in truncated BPTT for each parameter adaptation a summation of the products between \( \text{delta} \) and the parameter inputs must be computed over the considered \( h \) time steps past history, in CRBP the analogous of this summation is computed to calculate the \( \text{delta} \) (exp. (4.3.7)) that is used to adapt many weights, saving computation. For each parameter adaptation no summation in time is needed but just the product between \( \text{delta} \) and the derivative of the \text{net} with respect to the parameter (exp. (4.2.12) and (4.2.14)). The calculation scheme used is also motivated as a generalization to the recursive case of that implemented in Wan's Temporal Backpropagation (Wan 1990), which is the main training method for TDNN, and also of that used in
adjoint Least Mean Square (Wan 1996) a well known training procedure in the Signal Processing community.

The increase in computational complexity between the Back-Tsoi algorithm (Back and Tsoi 1991) and CRBP is only in formula (4.3.7): since, due to the recursion, the truncation parameter \((Q_{rec})\) can be usually chosen quite small (verified in simulations and theoretically motivated by the exponential decaying of impulse responses of stable rational transfer functions), this increase is fairly low. This holds, of course, in the case of forgetting behavior and not in the case of long term dependencies, but the first is the normal working condition of networks with local feedback. In simulations, we observed that the ratio between the execution times of CRBP and Back-Tsoi algorithm (one iteration) is less than 1.5, for usual architectures and parameter settings.

A mathematical evaluation of complexity can be carried out computing the number of multiplications and additions for one iteration (i.e. one sample here) of the learning phase (on-line learning). In Table 4.6.1, results for CRBP, Back-Tsoi, RTRL and truncated BPTT are reported in the significant special case of two layers IIR-MLP with bias and with MA-AR orders depending only on the layer index: \(M=2\), \(L^{(i)}=L^{(i)}\), \(I^{(i)}=I^{(i)}\).

The numbers in Table 4.6.1 must be added to the number of operations of the forward phase, always done before the backward phase.

<table>
<thead>
<tr>
<th>Learning algorithm</th>
<th># ADDITIONS</th>
<th># MULTIPLICATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRBP</td>
<td>(2N_i + N_i N_i [L^{(i)} + I^{(i)} + (I^{(i)})^2] + N_i N_i [L^{(i)} + L^{(i)} + (I^{(i)})^2] + (I^{(i)})^2 + Q_i + 1 + \sum_{j=1}^{\min(i,j)} (I^{(j)})^2 - \begin{cases} Q_i - L^{(i)} + 1 &amp; \text{if } Q_i &gt; L^{(i)} - 1 \ 0 &amp; \text{otherwise} \end{cases} )</td>
<td>(2(N_i + N_i) + N_i N_i [L^{(i)} + L^{(i)} + (I^{(i)})^2] + (I^{(i)})^2 + Q_i + 1 + \sum_{j=1}^{\min(i,j)} (I^{(j)})^2 )</td>
</tr>
<tr>
<td>Back-Tsoi</td>
<td>(2N_i + N_i N_i [L^{(i)} + E^{(i)} + (I^{(i)})^2 + (I^{(i)}) + 1] + N_i N_i [L^{(i)} + L^{(i)} + (I^{(i)})^2 + 1] )</td>
<td>(2(N_i + N_i) + N_i N_i [L^{(i)} + L^{(i)} + (I^{(i)})^2 + 1] )</td>
</tr>
<tr>
<td>RTRL</td>
<td>(2N_i + N_i N_i [L^{(i)} + E^{(i)} + (I^{(i)})^2 + (I^{(i)}) + 1] + N_i N_i [L^{(i)} + L^{(i)} + (I^{(i)}) + 1] + (I^{(i)})^2 + 2 )</td>
<td>(3(N_i + N_i) + N_i N_i [L^{(i)} + L^{(i)} + (I^{(i)})^2 + 1] + N_i N_i [L^{(i)} + L^{(i)} + (I^{(i)}) + 2] )</td>
</tr>
<tr>
<td>T-BPTT</td>
<td>(N_i + [N_i N_i (L^{(i)} + E^{(i)} + L^{(i)}) + 2N_i N_i (I^{(i)} + E^{(i)}) - N_i (N_i - 1) + N_i - N_i] )</td>
<td>(N_i + [N_i N_i (L^{(i)} + E^{(i)} + 1) + 2N_i N_i (I^{(i)} + E^{(i)}) - N_i (N_i - 1) + N_i - N_i] )</td>
</tr>
</tbody>
</table>

**Table 4.6.1** Number of operations for one iteration of the on-line learning phase for the different algorithms: general expressions.

The number of multiplications or additions of the forward phase (one iteration) is:

\[
M_{\text{forward}} = A_{\text{forward}} = N_1 N_0 \left( L^{(1)} + I^{(1)} \right) + N_2 N_1 \left( L^{(2)} + I^{(2)} \right). \tag{4.6.1}
\]

To provide an easy evaluation of complexity, Table 4.6.2 and 4.6.3 report the numbers of operations obtained from Table 4.6.1 for the parameters choice of
some simulations reported here. These values should be added to the number of forward operations before computing complexity ratios between different algorithms.

<table>
<thead>
<tr>
<th>Learning Algorithm</th>
<th># ADDITIONS</th>
<th># MULTIPLICATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRBP</td>
<td>140</td>
<td>188</td>
</tr>
<tr>
<td>Back-Tsoi</td>
<td>83</td>
<td>101</td>
</tr>
<tr>
<td>RTRL</td>
<td>167</td>
<td>237</td>
</tr>
<tr>
<td>T-BPTT</td>
<td>521</td>
<td>601</td>
</tr>
</tbody>
</table>

Table 4.6.2  Number of operations for one iteration of the on-line learning phase for the different algorithms when: $N_0=1$, $N_1=3$, $N_2=1$, $L^{(1)}=L^{(2)}=3$, $I^{(1)}=I^{(2)}=2$, $Q_2=10$, $h=10$.

<table>
<thead>
<tr>
<th>Learning Algorithm</th>
<th># ADDITIONS</th>
<th># MULTIPLICATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRBP</td>
<td>104</td>
<td>134</td>
</tr>
<tr>
<td>Back-Tsoi</td>
<td>83</td>
<td>101</td>
</tr>
<tr>
<td>RTRL</td>
<td>167</td>
<td>237</td>
</tr>
<tr>
<td>T-BPTT</td>
<td>209</td>
<td>241</td>
</tr>
</tbody>
</table>

Table 4.6.3  Number of operations for one iteration of the on-line learning phase for the different algorithms when: $N_0=1$, $N_1=3$, $N_2=1$, $L^{(1)}=L^{(2)}=3$, $I^{(1)}=I^{(2)}=2$, $Q_2=4$, $h=4$.

By the formulas in Table 4.6.1, substituting the configuration parameters of each of the three IIR-MLP networks used in the simulations reported, and with $Q_2=4$, it is possible to get an averaged ratio $M\{CRBP\}/M\{BackTsoi\}=1.24$ for the total number of multiplications and $A\{CRBP\}/A\{BackTsoi\}=1.18$ for the total number of additions, proving that the increase in complexity is quite small.

Moreover for IIR-MLP and activation feedback MLN it is possible to apply to CRBP the same simplification proposed by A.D. Back and A.C. Tsoi (1993) for their algorithm. The idea is exactly the same used in the adaptive IIR filter context (Hsia 1981, Shynk 1989) and is explained in the following, considering only IIR-MLP for the sake of simplicity. To compute the $y_{nm}^{(l)}$ quantity we have to pass the input of the synapse through the IIR synaptic filter; while to compute $\frac{\partial s_{nm}^{(l)}[t]}{\partial w_{nm}(t)}$ we need to pass the same input through the AR part only of the same filter. So, implementing the IIR filter in direct form II (i.e. filtering the synaptic input during the forward phase separately with the AR and MA parts
respectively) and keeping the intermediate value, the \( \frac{\partial s_n^{(l)}[t]}{\partial w_{nm}(0)} \) quantity is immediately available during the backward phase without further computations.

Moreover it is possible to introduce an approximation (Back and Tsoi 1993) to get an additional decrease on complexity: computing the different values of \( \frac{\partial s_n^{(l)}[t]}{\partial \text{weight}_{nm}(p)} \) (weight is \( w \) or \( v \)) for the different \( p \) as the same quantity obtained for the first \( p \) delayed by a suitable number of time steps, as shown in Fig. 4.3.1. This approximation is already known in the adaptive IIR filter theory (Hsia 1981, Shynk 1989) and is reasonable if the coefficients slowly change in time (a small learning rate is used) since it assumes the time invariance of the filter. The simulations do not show an appreciable loss of performance in the learning process (Back-Tsoi 1993, Hsia 1981, Shynk 1989), while the number of AR filtering operation per synapse is reduced from \( L_{nm}^{(l)}+I_{nm}^{(l)} \) to one. Anyway, in the present simulations exact formulas are used. Another minor simplification of the CRBP formulas was tested, neglecting the recursive part of expressions (4.2.15), (4.2.16) and (4.2.21) (i.e. assuming null past values) as it is sometimes done for IIR linear adaptive filters. However, the obtained performance were so unsatisfactory that we did not use it anymore.

About the stability of the CRBP algorithm, in all the simulations we observed that, if the IIR network was initialized with stable synapses, the final result is a stable network as far as a small enough learning rate is used. A similar behavior was exhibited by all the other locally recurrent architectures. For stability test and control methods refer to Chapter 7.

4.7 CONCLUSIONS

In conclusion, the results presented in this chapter show that the locally recurrent MLPs have superior modeling capabilities with respect to more traditional networks, namely MLP with external memory and FIR-MLP (or TDNN) for identification of the systems tested; see (Horne and Giles 1995) for a comparison on other test problems. A general approach to derive on-line algorithms for locally recurrent networks that are local both in time and in space was described and two new learning methods were proposed, that, include as special cases, several algorithms already known in literature. The proposed algorithms have better stability and higher speed of convergence compared to the Back-Tsoi one, as expected by the theoretical development and confirmed by simulations. Stability and speed of convergence are very important in real on-line applications, e.g. where time varying systems have to be tracked. The only
drawback of the algorithms is a slight increase in complexity with respect to Back-Tsoi method, which however can be easily reduced.

With respect to an application of RTRL and truncated BPTT, the proposed algorithms are computationally simpler and easier to implement.

The two on-line methods, namely CRBP and TRBP have similar performance but different complexity, therefore the choice should be done basically on complexity issues, for each particular implementation.
4.A APPENDIX: RBP FOR LOCAL FEEDBACK MULTI-LAYERED NETWORKS (MLNS) AND AUTO-REGRESSIVE MLP

For uniformity of presentation, the RBP algorithm is covered in this appendix instead of CRBP or TRBP. Of course, the same modifications explained in Section 4.3 and 4.4 can be implemented to derive the CRBP or TRBP from the RBP method.

In all of the following cases if no feedback is present \((I_n(l)=0\) for each \(n\) and \(l\)) each architecture becomes a FIR-MLP and the RBP algorithm gives Wan’s Temporal Back Propagation (TBP) (batch mode). If no memory is present at all \((I_n(l)=0\) and \(L_{nm}(l)=1\) for each \(n\), \(m\) and \(l\)) then each architecture becomes a standard MLP and RBP gives BP (batch mode). The on-line mode versions of TBP and BP are obtained as particular case of CRBP and of CRBP or TRBP respectively.

The forward and backward formulas for LF-MLNs and AR-MLP differ from those already derived for the IIR-MLP because the AR filtering is performed once for every neuron and not for every synapse, lowering the number of free parameters. Therefore the input index of the AR coefficients \((\nu)\) and the AR order \((I)\) is now meaningless and will be dropped in the following.

4.A.1 Activation feedback MLN

Only the formulas which are different from those corresponding to the IIR-MLP are reported here. They can be easily derived in a similar way. The forward formulas are now replaced by:

\[
s_n^{(l)}[t] = \sum_{m=1}^{N} \sum_{p=1}^{L_{fm}} w_{nm(p)}^{(l)} x_m^{(l-1)}[t-p] + \sum_{p=1}^{L_{f}} v_{n(p)}^{(l)} s_n^{(l)}[t-p]
\]

(4.A.1’)

\[
x_n^{(l)}[t] = \text{sgn}(s_n^{(l)}[t] + w_n^{(l)})
\]

(4.A.2’)

It should be noted that now \(s_n^{(l)}[t]\) does not represent the net quantity anymore since it differs from that for the additive bias. Due to this bias term the relation between \(\delta\) and backpropagating error is:

\[
\delta_n^{(l)}[t] = e_n^{(l)}[t] \text{sgn}'(s_n^{(l)}[t] + w_n^{(l)})
\]

(4.A.9’)

The coefficient variations are now computed using the derivatives expressed by:

\[
\frac{\partial s_n^{(l)}[t]}{\partial w_{nm(p)}^{(l)}} = x_m^{(l-1)}[t-p] + \sum_{r=1}^{I} v_{n(r)}^{(l)} \frac{\partial s_n^{(l)}[t-r]}{\partial w_{nm(p)}}
\]

(4.A.15’)

- 64 -
\[
\frac{\partial s^{(l)}_n[t]}{\partial v^{(l)}_{n(p)}} = s^{(l)}_n[t - p] + \sum_{r=1}^{l} \frac{\partial s^{(l)}_m[t - r]}{\partial v^{(l)}_{n(p)}}
\]  
\[(4.A.16')\]

The backpropagation is now:

\[
e^{(l)}_n[t] = \begin{cases} 
\epsilon^{(l)}_n[t] & \text{for } l = M \\
\sum_{q=1}^{N_q} \sum_{p=0}^{T_q^{(l+1)}} \delta^{(l+1)}_q[t + p] \frac{\partial s^{(l+1)}_q[t + p]}{\partial x^{(l)}_n[t]} & \text{for } l = (M - 1), \ldots, 1 
\end{cases}
\]  
\[(4.A.20')\]

where the derivatives are computed as:

\[
\frac{\partial s^{(l+1)}_q[t + p]}{\partial x^{(l)}_n[t]} = \begin{cases} 
w^{(l+1)}_{qn(p)} & \text{if } 0 \leq p \leq T^{(l+1)}_{qn} - 1 \\
0 & \text{otherwise} 
\end{cases} + \sum_{r=1}^{\min(T^{(l+1)}_{qn}, T^{(l)}_{qr})} \delta^{(l+1)}_q[t + p - r] \frac{\partial s^{(l+1)}_q[t + p - r]}{\partial x^{(l)}_n[t]}
\]  
\[(4.A.21')\]

Back Propagation for Sequences (Gori et al. 1989) (batch mode) for activation feedback MLN is obtained as a particular case if the architecture is constrained to have dynamic units only in the first layer. In the present notation this is written: \(I^{(l)}_n = 0\) and \(L^{(l)}_{nm} = 1\) for \(l > 1\) and for each \(n, m\). In this case, standard BP is applied to compute \(\delta\) since (4.A.20') and (4.A.21') become:

\[
e^{(l)}_n[t] = \begin{cases} 
\epsilon^{(l)}_n[t] & \text{for } l = M \\
\sum_{q=1}^{N_q} \delta^{(l+1)}_q[t] w^{(l+1)}_{qn} & \text{for } l = (M - 1), \ldots, 1 
\end{cases}
\]

Since RBP is a batch mode algorithm (cumulative adaptation of weights) the particular case is still a batch mode algorithm. The on-line original version of BPS is obtained from the CRBP or TRBP or simply substituting cumulative with incremental adaptation in RBP.

4.A.2 Output feedback MLN

The important difference of output feedback MLN with respect to IIR-MLP and activation feedback MLN is that in the former the dynamic part and the non-linearity are not separated anymore (see Section 2.4).

Now it is not possible to look at the internal summation of expression (4.2.20) as future convolution, because the derivative is not an impulse response anymore, due to the non-linearity. The differences in the learning formulas with respect to the activation feedback MLN is basically that the AR coefficients are now multiplied with the derivative of the activation function computed on the \(\text{net}\) in a certain time instant, as it is easy to prove by chain rule.
Only the formulas which are different from those corresponding to the
activation feedback MLN are reported here. The forward formulas are now
replaced by:

\[ s_n^{(l)}[t] = \sum_{m=0}^{N_{l-1}} \sum_{p=0}^{L_{mn}^{(l)}} w_{mn(p)}^{(l)} x_{m}^{(l-1)}[t - p] + \sum_{p=1}^{l^{(l)}} v_{n(p)}^{(l)} x_{n}^{(l)}[t - p] \]  

(4.A.1’’)

As for the IIR-MLP it holds:

\[ x_n^{(l)}[t] = \text{sgm}(s_n^{(l)}[t]) \]  

(4.A.2’’)

\[ \delta_n^{(l)}[t] = \epsilon_n^{(l)}[t] \text{sgm}'(s_n^{(l)}[t]) \]  

(4.A.9’’)

The coefficient variations are now computed using the derivatives expressed by:

\[ \frac{\partial s_n^{(l)}[t]}{\partial w_{mn(p)}^{(l)}} = x_m^{(l-1)}[t - p] + \sum_{i=1}^{l^{(l)}} v_{n(i)}^{(l)} \text{sgm}'(s_n^{(l)}[t - r]) \frac{\partial s_n^{(l)}[t - r]}{\partial w_{mn(p)}^{(l)}} \]  

(4.A.15’’)

\[ \frac{\partial s_n^{(l)}[t]}{\partial v_{n(p)}^{(l)}} = x_n^{(l)}[t - p] + \sum_{i=1}^{l^{(l)}} v_{n(i)}^{(l)} \text{sgm}'(s_n^{(l)}[t - r]) \frac{\partial s_n^{(l)}[t - r]}{\partial v_{n(p)}^{(l)}} \]  

(4.A.16’’)

The derivatives used in the backpropagation are now computed as:

\[ \frac{\partial s_q^{(i+1)}[t + p]}{\partial s_q^{(i)}[t]} = \begin{cases} \frac{w_{qp}^{(i+1)}}{L_q^{(i+1)} - 1} & \text{if } 0 \leq p \leq L_q^{(i+1)} - 1 \\ \sum_{i'=1}^{l(q,i)} \text{sgm}'(s_q^{(i+1)}[t + p - r]) \frac{\partial s_q^{(i+1)}[t + p - r]}{\partial s_q^{(i)}[t]} & \text{otherwise} \end{cases} \]  

(4.A.21’’)

As for the previous case, also for output feedback MLN, BPS learning
algorithm is obtained, as particular case, if the architecture is constrained to
have dynamic units only in the first layer.

4.A.3 Auto Regressive MLP

The main difference with all the previous cases is that now the chain rule
expansions are more easily written with respect to the neuron outputs instead of the
net quantities. This is because the AR memory is not included in the net. So
delta's are not useful anymore and they will be replaced by the backpropagating
errors. The forward formulas are now replaced by:

\[ s_n^{(l)}[t] = \sum_{m=0}^{N_{l-1}} \sum_{p=0}^{L_{mn}^{(l)}} w_{mn(p)}^{(l)} x_{m}^{(l-1)}[t - p] \]  

(4.A.1’’)

\[ x_n^{(l)}[t] = \text{sgm}(s_n^{(l)}[t]) + \sum_{p=1}^{l^{(l)}} v_{n(p)}^{(l)} x_{n}^{(l)}[t - p] \]  

(4.A.2’’)

- 66 -
The \( w \) coefficient variations are expressed by:

\[
\Delta w_{nm(p)}^{(l)}[t+1] = -\frac{\mu}{2} \frac{\partial E^2}{\partial x_n^{(l)}[t]} \frac{\partial x_n^{(l)}[t]}{\partial w_{nm}^{(l)}(p)} = \mu e_n^{(l)}[t] \frac{\partial x_n^{(l)}[t]}{\partial w_{nm}^{(l)}(p)}
\]  

(4.A.12‴)

Similarly for the \( v \) weights:

\[
\Delta v_{nm(p)}^{(l)}[t+1] = \mu e_n^{(l)}[t] \frac{\partial x_n^{(l)}[t]}{\partial v_{nm}^{(l)}(p)}
\]  

(4.A.14‴)

The coefficient variations are now computed using the derivatives expressed by:

\[
\frac{\partial x_n^{(l)}[t]}{\partial w_{nm}^{(l)}(p)} = \text{sgm}'(s_n^{(l)}[t])x_m^{(l-1)}[t - p] + \sum_{r=1}^{L} v_{nm}^{(l)} \frac{\partial x_n^{(l)}[t - r]}{\partial w_{nm}^{(l)}(p)}
\]  

(4.A.15‴)

\[
\frac{\partial x_n^{(l)}[t]}{\partial v_{nm}^{(l)}(p)} = x_n^{(l)}[t - p] + \sum_{r=1}^{L} v_{nm}^{(l)} \frac{\partial x_n^{(l)}[t - r]}{\partial v_{nm}^{(l)}(p)}
\]  

(4.A.16‴)

The backpropagation is now:

\[
e_n^{(l)}[t] = \begin{cases} e_q[t] & \text{for } l = M \\ \sum_{q=1}^{N} \sum_{p=0}^{T-1} e_q^{(l+1)}[t + p] \frac{\partial x_n^{(l+1)}[t + p]}{\partial x_n^{(l)}[t]} & \text{for } l = (M-1),...,1 \end{cases}
\]  

(4.A.20‴)

where the derivatives are computed as:

\[
\frac{\partial x_n^{(l+1)}[t + p]}{\partial x_n^{(l)}[t]} = \text{sgm}'(s_q^{(l+1)}[t + p]) \begin{cases} \frac{w_{nq}^{(l+1)}}{\text{sgm}(s_q^{(l+1)}[t + p])} & \text{if } 0 \leq p \leq L_q^{(l+1)} - 1 \\ 0 & \text{otherwise} \end{cases} + \sum_{r=1}^{L} v_{nq}^{(l+1)} \frac{\partial x_n^{(l+1)}[t + p - r]}{\partial x_n^{(l)}[t]}
\]  

(4.A.21‴)
Chapter 5

SIGNAL-FLOW-GRAPH APPROACHES: BACKWARD COMPUTATION

5.1 INTRODUCTION

For many practical problems such as time series forecasting and recognition, identification and control of general non-linear dynamic systems, feedforward Neural Networks (NNs) structures are not adequate, therefore several architectures of Recurrent Neural Networks (RNNs) have been proposed (Chapter 2). Recurrent NNs can be basically divided into three classes: Fully Recurrent NNs (FRNNs) (Williams and Peng 1990, Williams and Zipser 1989, Haykin 1994), (Section 2.3), Locally Recurrent Globally Feedforward NNs (Back and Tsoi 1991, Tsoi and Back 1994, Campolucci et al. 1995, Campolucci, Uncini and Piazza 1997), (Section 2.4), and NARX architectures (Haykin 1994, Narendra and Parthasarathy 1991), (Section 2.2).

In order to train a RNN adaptively, an on-line learning algorithm must be derived for each specific architecture. Considering gradient based learning algorithms, such as the steepest descent or conjugate gradient, the gradient of a cost function must be estimated and then used in the chosen parameters updating scheme, depending on the application. By gradient we mean the vector of derivatives of a certain 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 the circuit components).

When recurrent neural networks (or systems with feedback in general) are involved, the calculation of the gradient is a problem definitely richer and more
difficult than in the case of feedforward neural networks (or systems without feedback). In fact, the present output depends upon the past outputs of the RNN, so the present error depends not only on the present parameters but also on the past parameter values, and this dependence has to be considered in the calculation of the gradient. Quite often in the neural network literature this dependence has been neglected totally or partially to simplify the derivation of the gradient equations (for Back-Tsoi and other such methods refer to Sections 4.1, 4.3 and 4.4).

There are two different schemes to calculate the gradient: the “forward computation” approach and the “backward computation” approach (Section 3.2). Real Time Recurrent Learning (RTRL) (Williams and Zipser 1989) and truncated Back Propagation Through Time (truncated BPTT) (Williams and 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 and Peng 1990, Williams and Zipser 1994, Srinivasan et al. 1994, Nerrand et al. 1993). However, the use of the backward technique as available in literature, in the case of recurrent NNs with an arbitrary architecture, e.g. locally recurrent NNs or more general dynamic systems, is not straightforward; the mathematics of any particular architecture have to be worked out. For systems with feedback or with internal time delays, the chain rule derivation can easily become very complex and intractable (Werbos 1990, Williams and Zipser 1994, Back and Tsoi 1991, Pearlmutter 1995, Campolucci, Uncini and Piazza 1997), (Sections 3.2, 3.5, 3.6 and 4.2).

Using the Signal Flow Graph (SFG) representation theory and its known properties (Mason 1953 and 1956, Lee 1974, Oppenheim and Schafer 1975), a new general method is derived in this chapter for both on-line and batch backward gradient computation of a function (e.g. a cost function) of the system output with respect to the system parameters. This system can be any causal, in general non-linear and time-variant, dynamic system represented by a SFG, in particular any feedforward (static), time delay or recurrent NN. In the following, we shall usually consider the NN framework but the method is completely general. In this chapter, we use discrete time notation, however the same theory holds for the continuous time case.

The new method has been developed mainly for on-line learning (Section 1.3). 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 (Section 1.5), 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).
Chapter 5 SIGNAL-FLOW-GRAPH APPROACHES: BACKWARD COMPUTATION

This new learning approach generalizes the concept of truncated BPTT (backward computation) (Section 3.5) from fully recurrent networks, to any system which can be represented by a SFG and can be easily implemented for any specific architecture. It has therefore been named the Backward Computation (BC) method.

It can also be used for off-line learning. In this case, this approach gives the batch BPTT algorithm and the method by Wan and Beaufays (1996) that is not suitable for on-line training of recurrent networks or systems with feedback.

The authors in (Beaufays and Wan 1994) propose a proof of the equivalence of RTRL and BPTT in batch mode using the Interreciprocity property of transposed flow graphs to show that the computed weight variations are the same, though the complexity of the two algorithms is different. The paper (Beaufays and Wan 1994) is mainly of theoretical interest and does not address the problem of graphical derivation of RTRL, as in (Wan and Beaufays 1996) for BPTT; nevertheless, the two approaches can be combined for that purpose. This would allow for a diagrammatic derivation of an on-line learning method, but the resulting RTRL algorithm would have a very high computational complexity, i.e. $O(n^2)$ as opposed to $O(n)$ of the on-line backward method presented here (where $n$ is the number of graph components, as defined in Section 5.2. While we provide a proof based on a theorem, proved by Lee (1974), similar to Tellegen’s theorem for analog network, the proof by Wan and Beaufays for their diagrammatic derivation is based on the interreciprocity property of transposed graphs, that is a consequence of the theorem used here.

This work also generalizes a previous one by Osowski (1994) that proposed a SFG approach to neural networks 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 also 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 Multi-Layer-Perceptron (MLP) showing that a circuit implementing backpropagation arises. 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 for the method presented here.

Lastly, our formulation allows us to deal both with the learning problem and with the sensitivity calculation problem, i.e. 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 chapter were developed and briefly presented in (Campolucci 1996a, Campolucci, Marchegiani et al. 1997, Campolucci et al. 1998).

5.2 SIGNAL FLOW GRAPHS: A THEOREM SIMILAR TO TELLEGEN’S THEOREM

The aim of this chapter is to show how gradient computation in arbitrary complex architectures can be performed with simple graphical transformations of the SFG which describes the system, completely avoiding any complex mathematical development. This can be done by moving from a mathematical description of the model, such as a neural network, to a graph based description (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 introduced by Oppenheim and Schafer (1975) but we find Lee’s approach more straightforward for this purpose. A SFG theory was first developed in 1953, by Mason (1953, 1956).

A large class of complex systems, such as RNNs, can be represented by SFGs. A SFG is a set of nodes and oriented branches. A branch is oriented from node $i$ to node $j$ if node $i$ is the initial node and node $j$ is the final node of the branch. An input node is one which has only one outgoing branch, the input branch, and no incoming branch. Associated with the input branch is an input variable $u$. An output node is one which has only one incoming branch, the output branch. Associated with the output branch is an output variable $y$. 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 $x$, at the tail of the branch, and the final variable $v$, 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 $f_j[.]$ is a general function describing the operator (circuit component) of branch $j$-th. By definition, for each $n$-node, the value of the $x$ variables associated with its outcoming branches is the sum of all the $v$ variables associated with its incoming branches, refer to Fig. 5.2.1. 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). Please note that in the following mathematical development only branches must be indexed (and not nodes), therefore the reported indices are to be considered as branch indices.
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

\[
x_j = \sum_{i \in I_A} v_i \quad \forall j \in Z_A.
\]

In the case of discrete time systems, the functional relationship between the variables \( x_j \) and \( v_j \) of the \( f \)-branch \( j \)-th (expressed as \( v_j = f_j(x_j) \)) can usually be detailed in this way:

\[
\begin{align*}
    v_j(t) &= g_j[x_j(t), \alpha_j(t), t] & \text{for a static branch (no memory)} \\
    v_j(t) &= q^{-1} x_j(t) = x_j(t-1) & \text{for a delay branch (one unit memory)}
\end{align*}
\]

where \( q^{-1} \) is the delay operator and \( g_j \) is a general differentiable function, depending on the adaptable parameter (or vector of parameters) \( \alpha_j \). This function \( g_j \) can change on time, i.e. it directly depends on time \( t \).

For a static branch the relationship between \( x_j \) and \( v_j \) is often specified as

\[
\begin{align*}
    v_j(t) &= w_j(t) x_j(t) & \text{for a weight branch} \\
    v_j(t) &= f_j(x_j(t)) & \text{for a non-linear branch}
\end{align*}
\]

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. In the neural networks context, \( w_j(t) \) is a synaptic weight and \( f_j \) a sigmoidal activation function.

Equation (5.2.1) can represent a more general non-linear adaptable function controlled by a parameter (or vector of parameters) \( \alpha_j \), such as a spline based activation function (Vecci et al. 1997).

The generic connection (also with feedback) of the branches described by (5.2.2), (5.2.3) and possibly (5.2.1), to allow shape adaptation or variation in time of the non-linearity, gives the overall SFG \( (G_N) \) that implements the system, see Fig. 5.2.2. 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.

![SFG representation of a generic system.](image)

Fig. 5.2.2  *SFG representation of a generic system.*

A reversed SFG \( \hat{G}_N \) of a given SFG \( G_N \) is obtained by reversing the orientation of all the branches in \( G_N \), i.e. by replacing the summing junctions with branching points and viceversa. Let \( \hat{u}_k, \hat{v}_k, \hat{x}_k \) and \( \hat{v}_k \) be the variables associated with the branches in \( \hat{G}_N \) corresponding to \( u_k, v_k, x_k \) and \( v_k \) in \( G_N \).

In \( \hat{G}_N \) the input variables are \( \hat{v}_k \) \( k = 1, \ldots, r \), the output variables are \( \hat{u}_k \) \( k = 1, \ldots, m \), \( \hat{v}_k \) and \( \hat{x}_k \) are the initial and the final variable of the \( k \)-th \( f \)-branch respectively. Let the indices assigned to the branches of \( \hat{G}_N \) be the same as the indices assigned to the corresponding branches of \( G_N \). As previously stated, node indices are not required by this development. The functional relationships between the variables \( \hat{x}_k \) and \( \hat{v}_k \) in \( \hat{G}_N \) are generic, therefore the graph \( \hat{G}_N \) effectively represents a set of possible SFGs.

The transposed SFG is defined as a reversed SFG when the relationships of the branches of \( G_N \) and \( \hat{G}_N \) remain the same. The adjoint SFG is defined as a reversed SFG specifying a particular choice (to be explained later) for the branch relationships of \( \hat{G}_N \).

For an example of SFG and reversed SFG, see Fig. 5.2.3 (a,b,c).
Fig. 5.2.3  (a) A simple IIR-MLP neural network with 2 layers, 2 inputs, 1 output, 1 neuron in the first layer with 2 IIR filter synapses (1 zero, 1 pole) and 1 neuron in the second layer with 1 IIR filter synapse (2 zeros, 2 poles); (b) a section of its SFG $G_N$; (c) its reversed SFG $\hat{G}_N$; (d) its adjoint SFG $\hat{G}_N^\phi$. 
For a generic SFG, the following theorem, similar to Tellegen’s theorem for electrical network (Tellegen 1952, Penfield et al. 1970), was derived by Lee (1974). It relies solely on the topological properties of the original and reversed graphs and not on the functional relationships between the branch variables, that can therefore be of any kind.

**Theorem (Lee 1974):** Consider a causal dynamic system. Let $G_N$ be the corresponding signal flow graph and $\hat{G}_N$ a generic reversed signal flow graph. If $\hat{u}_k, \hat{y}_k, \hat{x}_k$ and $\hat{v}_k$ are the variables in $\hat{G}_N$ corresponding to $u_k, y_k, x_k$ and $v_k$ in $G_N$, respectively, then

$$
\sum_{i=1}^{m} \hat{y}_i(t) * y_i(t) + \sum_{i=1}^{q} \hat{x}_i(t) * x_i(t) = \sum_{i=1}^{m} \hat{u}_i(t) * u_i(t) + \sum_{i=1}^{q} \hat{v}_i(t) * v_i(t)
$$

(5.2.4)

holds true, where “*” denotes the convolution operator, i.e.

$$
y(t) * x(t) = \sum_{\tau = t_0}^{t} y(t - \tau) x(\tau)
$$

(5.2.5)

where $t_0$ is the initial time.

For a static system the following equation holds

$$
\sum_{i=1}^{m} \hat{y}_i y_i + \sum_{i=1}^{q} \hat{x}_i x_i = \sum_{i=1}^{m} \hat{u}_i u_i + \sum_{i=1}^{q} \hat{v}_i v_i
$$

(5.2.6)

### 5.3 SFG APPROACH TO SENSITIVITY CALCULATION BY BACKWARD COMPUTATION (BC)

A general method to compute sensitivity, i.e. the derivative of an output with respect to a parameter of a dynamic continuous time system represented by a signal flow graph was derived by Lee (1974) for constant parameters. Similarly, for a discrete time system, an analogous result can be found. However, such a method has two disadvantages for our purposes: the first is that it is formulated for sensitivity computation only and therefore merely in an off-line mode, and the second is that the mathematical meaning of the variables of the adjoint graph (Lee 1974) in the equation for the gradient calculation is not directly expressed in terms of derivatives. Here, we have derived a new method to on-line compute the derivative of a system output with respect to past (or present) parameters of the system. By such a method, both the above problems are solved and adaptive systems with time varying parameters can be considered. The parameters we are referring to are the weights $w_i$ and the non-linearity 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 provided by this method and not considered in the literature, e.g. (Lee 1974) and (Wan and Beaufays 1996).

The derivative of an output \( y_k(t) \) of the system with respect to a parameter \( w_i(t-\tau) \), where \( t \) is the time of the original SFG and \( \tau \) a positive integer and \( t \geq 0 \) and \( 0 \leq \tau \leq t \), by the first equation in (5.2.3) is

\[
\frac{\partial y_k(t)}{\partial w_i(t-\tau)} = \frac{\partial y_k(t)}{\partial v_i(t-\tau)} \frac{\partial v_i(t-\tau)}{\partial w_i(t-\tau)} = \frac{\partial y_k(t)}{\partial v_i(t-\tau)} x_i(t-\tau) \tag{5.3.1}
\]

where \( k=1,\ldots,r \) and \( i \) spans over the set of indices of the weight branches. For the parameters of the non-linear function, by (5.2.1), the following more general expression holds:

\[
\frac{\partial y_k(t)}{\partial \alpha_i(t-\tau)} = \frac{\partial y_k(t)}{\partial v_i(t-\tau)} \frac{\partial v_i(t-\tau)}{\partial \alpha_i(t-\tau)} = \frac{\partial y_k(t)}{\partial v_i(t-\tau)} \frac{\partial g_i}{\partial \alpha_i}\bigg|_{i,(t-\tau),\alpha_i(t-\tau),j,j-t-\tau} \Delta \frac{\partial y_k(t)}{\partial v_i(t-\tau)} \frac{\partial g_i}{\partial \alpha_i(t-\tau)} \tag{5.3.2}
\]

where \( k=1,\ldots,r \) and \( i \) spans over the set of indices of the non-linear parametric branches, and a simplified notation for the derivative of \( g \) is implicitly defined.

Thus, in both cases, we need to compute \( \frac{\partial y_k(t)}{\partial v_i(t-\tau)} \), which is the derivative of the \( k \)-th output variable at time \( t \) with respect to a signal in the system, i.e. the final variable of the \( i \)-th \( f \)-branch at time \( t-\tau \).

As for the computation of the sensitivity in an electrical circuit, also in this case an adjoint network can be introduced to perform this task. The main idea is to apply Lee’s theorem (equation 5.2.4) to the SFG \( G_N \) of the original system and to a particular reversed SFG \( \hat{G}_N \), here called adjoint SFG or \( \hat{G}_N^{(a)} \). Since the functional relationships of the branches of \( \hat{G}_N^{(a)} \) can be freely chosen, as equation (5.2.4) does not depend on them, these relationships are chosen in order to obtain an estimate of \( \frac{\partial y_k(t)}{\partial v_i(t-\tau)} \) from equation (5.2.4) itself.

The adjoint network \( \hat{G}_N^{(a)} \) is thus defined as the 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 5.3.1 and briefly described in the following.
### Table 5.3.1 Adjoint SFG construction rules. 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.

For each delay branch in the original SFG, the corresponding branch of the adjoint SFG is a delay branch (with zero initial condition). The non-linearity without parameters \( f_j \) in the original SFG corresponds to a gain equal to the derivative of \( f_j \) computed at 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 non-linearity with control parameters. The outputs of the original SFG correspond to the inputs of the adjoint SFG.

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

\[
\hat{y}_j(\tau) = \begin{cases} 
0 & \forall \tau \text{ if } j \neq k \\
1, \tau = 0 & j=1,\ldots,r \\
0, \tau > 0 & j = k 
\end{cases}
\]

(5.3.3)

For an example of adjoint SFG construction for a simple MLP with Infinite Impulse Response (IIR) filter synapses see Fig. 5.2.3.

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

\[
\frac{\partial y_k(t)}{\partial v_i(t-\tau)} = \hat{v}_i(\tau) 
\]

where \( k=1,\ldots,r \) and \( i=1,\ldots,q \).

Using this result in (5.3.1) we get

\[
\frac{\partial y_k(t)}{\partial w_i(t-\tau)} = \hat{v}_i(\tau)x_i(t-\tau) 
\]

(5.3.5)

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

\[
\frac{\partial y_k(t)}{\partial \alpha_i(t-\tau)} = \hat{v}_i(\tau)\frac{\partial g_i(t-\tau)}{\partial \alpha_i} 
\]

(5.3.6)
5.4 SFG APPROACH TO NON-LINEAR DYNAMIC SYSTEMS LEARNING

In the previous section, we have shown how the SFG of the original network \( G_N \) and the SFG of the adjoint network \( \hat{G}_N^{(w)} \) can be used to compute the sensitivity of an output with respect to a (past or present) system parameter (equations 5.3.5 and 5.3.6).

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 whole system has the inputs of the original network and the desired outputs, as inputs, and the value of the cost function as its unique output. The gradient of this cost function with respect to the system parameters (i.e. weights, etc.) now corresponds to the sensitivity of this output, which can be computed by again using equation (5.3.5) and (5.3.6), applied however, to the cascade system.

5.4.1 Cost functions and parameters updating rules

Let us consider a discrete time non-linear dynamic system with inputs \( u_i \), \( i = 1,\ldots,m \), outputs \( y_i \), \( i = 1,\ldots,r \), and parameters \( w_j \) and \( \alpha_x \) which have to be adapted with respect to an output error. Using gradient based learning algorithms following the steepest-descent method updating rules, it holds, e.g. for the weight \( w_i \)

\[
\Delta w_i = -\mu \frac{\partial J}{\partial w_i}, \quad \mu > 0 \tag{5.4.1}
\]

where \( J \) is a 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 the parameters can change at each time instant, following the BPTT approach:

\[
\frac{\partial J}{\partial w_i} = \sum_{\tau=-h+1}^{i-1} \frac{\partial J}{\partial w_i(\tau)} = \sum_{\tau=0}^{h-1} \frac{\partial J}{\partial w_i(t-\tau)} \tag{5.4.2}
\]

As the length of the summation increases linearly with the current time step \( t \), for on-line learning the summation in (5.4.2) must be truncated in order to implement the algorithm

\[
\Delta w_i(t) = -\mu \sum_{\tau=-h+1}^{i-1} \frac{\partial J}{\partial w_i(\tau)} = -\mu \sum_{\tau=0}^{h-1} \frac{\partial J}{\partial w_i(t-\tau)} \tag{5.4.3}
\]
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, e.g. recurrent NNs, while for feedforward networks with delays (e.g. Time Delay NNs or TDNNs) 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; for layered TDNN $h$ should depend on the layer and should be increased parting from the last to the first layer, since more memory is involved.

The above updating rule (equations 5.4.2 and 5.4.3) holds theoretically in the hypothesis of constant weights but practically it is only a good approximation. Equations (5.3.5) and (5.3.6) do not require that hypothesis, and can be used in a more general context. Equations analogous to (5.4.1), (5.4.2) and (5.4.3) 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

$$e^2(t) = \sum_{k=1}^{r} e_k^2(t) \quad \text{with} \quad e_k(t) = d_k(t) - y_k(t) \quad (5.4.4)$$

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

$$E(t_0,t_1) = \sum_{t=t_0}^{t_1} e^2(t) \quad (5.4.5)$$

In the case of non-adaptive (or batch) training, the cost function can be chosen as the global squared error 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 and Parthasaraty 1991). Therefore it holds:

$$\begin{cases}
E(0,T) = \sum_{t'=0}^{T} e^2(t') & \text{batch training} \\
E(t-N_c+1,t) = \sum_{t'=t-N_c+1}^{t} e^2(t') & \text{on-line training}
\end{cases}$$

Assuming $J = E(t_0,t_1)$, it follows
\[ \Delta w_i(t) = -\mu \sum_{\tau=0}^{h-1} \frac{\partial E(t - N_e + 1, t)}{\partial w_i(t - \tau)} \] 

for on-line training \hspace{1cm} (5.4.6)

and

\[ \Delta w_i = -\mu \sum_{\tau=0}^{T} \frac{\partial E(0,T)}{\partial w_i(T - \tau)} \] 

for batch training

5.4.2 Cost function gradient calculation by SFGs

Let us consider a new SFG (named \(G_S\) in the following) obtained by connecting the SFG \(G_N\) of the system to be adapted and the SFG which implements the cost function \(J\) (named \(G_J\) in the following), see Fig. 5.4.1 for an example. \(G_J\) has the outputs of the adaptable system \(y_i\) and the targets \(d_i\) as inputs and the value of \(J\) as output. Obviously \(G_J\) can contain the same kind of operators as \(G_N\), i.e. delays, non-linearities, parameters to be adapted (e.g. for regularization purposes) and feedback. The class of cost functions allowed by this approach is therefore enlarged with respect to Wan and Beaufays’ work (1996), 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 $\hat{G}_N^{(a)}$ of the new SFG $G_N$, it is easy to calculate the sensitivities of the output with respect to the system parameters (equations 5.3.5 and 5.3.6). Since the output of $G_S$ is the cost function $J$, these sensitivities will result:

$$\frac{\partial J(t)}{\partial W_i} = \sum_{\tau=0}^{h-1} \hat{y}_i(\tau) x_i(t-\tau) \quad (5.4.7)$$

$$\frac{\partial J(t)}{\partial \alpha_i} = \sum_{\tau=0}^{h-1} \hat{y}_i(\tau) \frac{\partial g_i}{\partial \alpha_i}(t-\tau) \quad (5.4.8)$$

which are the quantities needed for the learning process.

When $J$ is chosen to be a standard MSE cost function, we now show that the explicit implementation of $G_j$ can be avoided. Using, for example, the instantaneous squared error, i.e. $J = e^2(t)$, the input-output relationship of $G_j$ is given by (5.4.4), thus by differentiation

$$\frac{\partial e^2(t)}{\partial y_k(t-\tau)} = \begin{cases} -2e_k(t), & \tau = 0 \\ 0, & \tau > 0 \end{cases}, \quad k=1,\ldots,r \quad (5.4.9)$$
In $G_S$ by (5.3.4) and (5.4.9) we get

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

(5.4.10)

where $\hat{y}_k$ is the $k$-th input of $\hat{G}_N^{(a)}$. Therefore the signal (5.4.10) can be used as input of $\hat{G}_N^{(a)}$ instead of (5.3.3) avoiding the explicit use of $\hat{G}_J^{(a)}$.

If, instead $J = E(t - N_c + 1, t)$ then the following signal must be injected into $\hat{G}_N^{(a)}$ (see Fig. 5.4.1):

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

(5.4.11)

Applying this method to a batch context, i.e. when $J = E(0, T)$, the following equation has to be used instead of (5.4.10) or (5.4.11)

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

(5.4.12)

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 and Beaufays 1996), see Fig. 5.4.2.
Fig. 5.4.2  
(a) SFG $G_{\mathcal{S}}$, i.e. the system SFG $G_N$ cascaded by the error calculation SFG $G_J$ with $J=E(0,t)$, $\text{sqr}(x)$ means $x^2$; (b) its adjoint $\hat{G}_{\mathcal{J}}^{(a)}$, i.e. $\hat{G}_{\mathcal{J}}^{(a)}$ cascaded by $\hat{G}_{\mathcal{N}}^{(a)}$. The feedback in (a) sums the error terms from the initial instant. The feedback in (b) allows us to inject into the SFG’s left-side a constant value equal to 1 for all $\tau$.

Thus, in these three cases, the derivative calculation can be performed considering $G_N$ and $\hat{G}_{\mathcal{N}}^{(a)}$ only, but now using either (5.4.10), (5.4.11) or (5.4.12) as inputs for $\hat{G}_{\mathcal{N}}^{(a)}$ instead of (5.3.3).

In (Srinivasan et al. 1994), a theorem which states substantially what is expressed in (5.4.7) when $J=e^2(t)$, is derived but 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 our method 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 for the batch case.

### 5.4.3 Detailed steps of the Backward Computation procedure

To derive the BC algorithm for training an arbitrary SFG $G_N$, using a MSE cost function, first the adjoint SFG $\hat{G}_{\mathcal{N}}^{(a)}$ must be drawn by reversing the graph and applying the transformation of Table 5.3.1. If the desired cost function is not a conventional MSE, then the cost function SFG $G_J$ must be designed and
cascaded to the network SFG, giving an overall \( G_S \) SFG, then the adjoint SFG \( \hat{G}^{(a)}_S \) must be drawn.

If on-line learning is desired, i.e. choosing for example \( J = E(t - N_c + 1, t) \), then the following steps have to be performed for each time \( t \):

a) the system SFG, \( G_N \) is computed one time step, keeping in memory the states for the last \( h \) time steps;

b) the adjoint SFG \( \hat{G}^{(a)}_N \) is reset (setting null initial conditions for the delays);

c) \( \hat{G}^{(a)}_N \) is computed for \( \tau = 0, 1, \ldots, h-1 \) with the input given in (5.4.11) computing the terms in summation (5.4.7) and (5.4.8);

d) the parameters \( w_j \) and \( \alpha_j \) are adapted by (5.4.1).

However, not all the adjoint SFG has to be computed, but only the initial variables of the branches containing adaptable parameters must be calculated to get (5.4.7) and (5.4.8). It must be stressed that the delay operator of the adjoint SFG delays the \( \tau \) and not the \( t \) index.

In the case of a non MSE cost function, \( G_S \) must be considered instead of \( G_N \) and \( \hat{G}^{(a)}_S \) instead of \( \hat{G}^{(a)}_N \) and the input of the adjoint SFG, \( \hat{G}^{(a)}_S \), is given by equation (5.3.3).

If batch learning is desired, i.e. choosing for example \( J = \sum_{t=0}^{T} e^2 (t) \), then the procedure is simpler. For each epoch:

a) the system SFG, \( G_N \), is computed, storing its internal states from time \( t=0 \) up to time \( T \);

b) the adjoint SFG \( \hat{G}^{(a)}_N \) is reset;

c) \( \hat{G}^{(a)}_N \) is evaluated for \( \tau = 0 \) up to \( T \), with \( t=T \), accumulating the terms for (5.4.7) and (5.4.8), (in those equations \( h-1 \) must be substituted by \( T \));

d) the parameters \( w_j \) and \( \alpha_j \) are adapted by (5.4.1).

Similar comments on implementation optimization, rule of the delay operators in adjoint SFGs and non MSE cost functions hold for the batch case as for the on-line.

The batch mode BC learning method corresponds to batch BPTT and Wan and Beaufays’ method (1996).
The on-line mode BC procedure corresponds to truncated BPTT (Williams and Peng 1990).

5.4.4 Example of application of the BC procedure

For an example of such a procedure, a very simple SFG, i.e. a recurrent neuron with two adaptable parameters is trained (see Fig. 5.4.3a). The equations of the BC algorithm will be detailed to make clear the method and to show the relationship with truncated BPTT\((h)\) (Williams and Peng 1990), in on-line mode, and with BPTT in batch mode.

\[
\begin{align*}
\Delta s(t) &= x_2(t) = w_1(t)u(t) + w_4(t)y(t-1) \\
y(t) &= f_2(s(t))
\end{align*}
\]

(a) On-line case:

For the simplest case \( J = e^2(t) = (d(t) - y(t))^2 \), the input of the corresponding adjoint SFG (Fig. 5.4.3b) must be:
\[ \hat{y}(\tau) = \begin{cases} -2e(t), & \tau = 0 \\ 0, & \tau > 0 \end{cases} \]  
(5.4.15)

The BC equations can be simply read out from the adjoint SFG:

\[ \dot{v}_4(\tau) = f'(s(t - \tau)) \left( \hat{y}(\tau) + \begin{cases} \hat{w}_4(\tau - 1) \hat{v}_4(\tau - 1), & \tau > 0 \\ 0, & \tau = 0 \end{cases} \right) \]  
(5.4.16)

where (see Table 5.3.1)

\[ \hat{w}_4(\tau - 1) = w_4(t - \tau + 1) \]  
(5.4.17)

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

\[ \Delta w_1(t) = \mu \sum_{t=0}^{h-1} \delta(t - \tau) u(t - \tau) \]  
(5.4.18)

\[ \Delta w_4(t) = \mu \sum_{t=0}^{h-1} \delta(t - \tau) y(t - \tau - 1) \]  
(5.4.19)

where

\[ \delta(t - \tau) = \frac{\Delta e^2(t)}{\partial s(t - \tau)} = \frac{\partial e^2(t)}{\partial \hat{v}_1(t - \tau)} = -\dot{v}_1(\tau) \]  
(5.4.20)

It can easily be seen that these equations do indeed correspond to truncated BPTT(h), since the introduced \( \delta \) is simply the usual \( \delta \) of truncated BPTT. A weight buffer is used since it is theoretically correct, although sometimes it is neglected and only the current weights are used in practice to obtain a simpler approximated implementation (Williams and Peng 1990).

It must be noted though, that in spite of the simplicity of the system, i.e. 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.

**b) Batch case:**

Forward equations are computed for \( t=0, \ldots, T \) saving the internal states, without any evaluation of the adjoint SFG.

The weight time index can be neglected since the weights are constant.

In this case the global squared error over the entire epoch is taken,

\[ J = \sum_{t=0}^{T} e^2(t), \]  
therefore the input of the adjoint SFG (Fig. 5.4.3b) must be:
The learning equations of this SFG can be simply read out from the adjoint SFG, remembering that \( t = T \):

\[
\hat{v}_1(\tau) = \hat{v}_4(\tau) = f'(s(T - \tau)) \left( \hat{y}(\tau) + \begin{cases} \hat{w}_4 \hat{v}_4(\tau - 1), & \tau > 0 \\ 0, & \tau = 0 \end{cases} \right) \quad (5.4.22)
\]

where \( \hat{w}_4 = w_4 \).

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

\[
\Delta w_i = \mu \sum_{\tau=0}^{T} \delta(T - \tau) u(T - \tau) \quad (5.4.23)
\]

\[
\Delta w_4 = \mu \sum_{\tau=0}^{T} \delta(T - \tau) y(T - \tau - 1) \quad (5.4.24)
\]

where

\[
\delta(T - \tau) = \frac{\partial J}{\partial s(T - \tau)} = -\frac{\partial J}{\partial v_1(T - \tau)} = -\hat{v}_1(\tau) \quad (5.4.25)
\]

It is easy to see that these equations do correspond to BPTT, since the introduced \( \delta \) is simply the usual \( \delta \) of BPTT.

### 5.5 Conclusions

The complexity of the proposed gradient computation is minimal, since it increases linearly with the number of adaptable parameters. In particular, since the adjoint graph has the same topology of the original one, the complexity of its computation is about the same. In practice, it is lower, since not all the adjoint graph has to be computed, as stated in paragraph (5.4.3). As in on-line mode, the adjoint graph must be evaluated \( h \) times for each time step, 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 necessary for (5.4.7) and (5.4.8) (linear with \( h \) and with the number of parameters), for each time step.

For the simple case of a fully recurrent, single layer, single delay neural network (Section 2.3) composed of \( n \) neurons, the computational complexity is \( O(n^2) \) operations per time step for epochwise BC or \( O(n^2h) \) 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 $n^2/h$. Hence, if $n^2$ is large enough compared to $h$, and that is usually the case, then RTRL will be more complex than on-line BC.

For a complex architecture such as a locally recurrent layered network (Section 2.4), a mathematical evaluation of complexity can be carried out by computing the number of multiplications and additions for one iteration (i.e. one sample here) of the learning phase (on-line learning). Results for on-line BC are reported in the significant special case of a two layers MLP with Infinite Impulse Response (IIR) temporal filter synapses (IIR-MLP) (Tsoi and Back 1994, Campolucci, Uncini and Piazza 1997), (Section 2.4) with bias and with Moving Average (MA) and Auto Regressive (AR) orders ($L(l)$ and $I(l)$, respectively), depending only on the layer index $l$.

The number of additions and multiplications are respectively:

\[
\begin{align*}
N_2 + h[N_1N_0(2I^{(1)} + L^{(1)}) + 2N_2N_1(I^{(2)} + L^{(2)}) - N_2(N_1 - 1) + N_1] , \\
N_2 + h[N_1N_0(2I^{(1)} + L^{(1)} + 1) + 2N_2N_1(I^{(2)} + L^{(2)}) + N_1(N_2 + 1)]
\end{align*}
\]

where $N_i$ is the number of neurons of layer $i$-th.

These numbers must be added to the number of operations of the forward phase, which should always be done before the backward phase, i.e. $N_iN_0(L^{(1)} + I^{(1)}) + N_2N_1(L^{(2)} + I^{(2)})$, that is the same for both additions and multiplications.

In this chapter we presented a Signal Flow Graph approach which easily allows us the on-line computation of the gradient terms needed in both sensitivity and learning problems of non-linear dynamic adaptive systems represented by SFG, even if their structure is complicated by feedback (of any kind, even nested) or time delays. This method allows us to avoid 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 complex training algorithms based on the gradient, such as conjugate gradient or other fancy techniques. This method should allow for the development of a Computer Aided Design (CAD) software, by which 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 extends previous results in different fields and notably provides an elegant analogy with the sensitivity analysis for analog networks by the “adjoint
network” obtained applying Tellegen’s theorem that relates voltages and currents of an analog network (Director and Rohrer 1969).

5.A APPENDIX: PROOF OF THE ADJOINT SFG METHOD

Let us suppose we have a non-linear dynamical system described by a SFG with the notation defined in Section 5.2.

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 \( \Delta u' \) to a specific node of the graph (Fig. 5.A.1a). This is equivalent to considering a system with \( m+1 \) inputs, i.e. one more than the original system, where the input \( (m+1) \)-th is \( u_{m+1}(t) = u'(t) \) where \( u'(t) \) is defined as

\[
\begin{align*}
    u'(t - \phi) = & \begin{cases} 
    0 & , \phi \neq \tau \\
    \Delta u' & , \phi = \tau 
    \end{cases}
\end{align*}
\]

![Fig. 5.A.1](image)

(a) the perturbation introduced in the original SFG; (b) the corresponding node in the adjoint SFG.

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: \( \Delta u'(t) = u'(t) , \forall t \).

Applying (5.2.4) it follows

\[
\sum_{j=1}^{m+1} \hat{y}_j(t) * y_j(t) + \sum_{j=1}^{m} \hat{x}_j(t) * x_j(t) = \sum_{j=1}^{m+1} \hat{u}_j(t) * u_j(t) + \sum_{j=1}^{m} \hat{v}_j(t) * v_j(t) \quad (5.1)
\]
and

\[
\sum_{j=1}^{m+1} \hat{y}_j(t) \ast (y_j(t) + \Delta y_j(t)) + \sum_{j=1}^{q} \hat{x}_j(t) \ast (x_j(t) + \Delta x_j(t)) = \\
= \sum_{j=1}^{m+1} \hat{u}_j(t) \ast (u_j(t) + \Delta u_j(t)) + \sum_{j=1}^{q} \hat{v}_j(t) \ast (v_j(t) + \Delta v_j(t))
\]  

(5.A.2)

Since the convolution is a linear operator, subtracting (5.A.1) from (5.A.2) results in

\[
\sum_{j=1}^{m+1} \hat{y}_j(t) \ast \Delta y_j(t) + \sum_{j=1}^{q} \hat{x}_j(t) \ast \Delta x_j(t) = \sum_{j=1}^{m+1} \hat{u}_j(t) \ast \Delta u_j(t) + \sum_{j=1}^{q} \hat{v}_j(t) \ast \Delta v_j(t)
\]  

(5.A.3)

Now we will compute the convolution terms for each kind of \( f \)-branch showing that for each \( f \)-branch in Table 5.3.1 (left column) it holds:

\[
\hat{v}_j(t) \ast \Delta v_j(t) = \hat{x}_j(t) \ast \Delta x_j(t)
\]  

if the corresponding branch in the adjoint graph is defined as in Table 5.3.1 (right column). This equation will be used to simplify (5.A.3).

a) DELAY BRANCHES:

\[
v_j(t) = q^{-1} x_j(t) \quad \Rightarrow \quad \Delta v_j(t) = \Delta x_j(t - 1) = q^{-1} \Delta x_j(t)
\]  

(5.A.4)

Let us define the relationship between the branch variables in \( \hat{G}_N \), in this case as

\[
\hat{x}_j(\tau) = q^{-1} \hat{v}_j(\tau)
\]  

(5.A.5)

where \( \tau \) is the time index for the adjoint SFG and \( t \) the time index for 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 \( \phi > \tau \) and the initial conditions of \( \hat{G}_N \) are set to be zero, that is

\[
\hat{x}_j(0) = 0
\]  

(5.A.6)

it follows

\[
\hat{v}_j(t) \ast \Delta v_j(t) = \sum_{\phi=0}^{t} \hat{v}_j(\phi) \Delta v_j(t - \phi) = \sum_{\phi=0}^{t} \hat{v}_j(\phi) \Delta v_j(t - \phi) = \\
= \sum_{\phi=0}^{t} \hat{x}_j(\phi + 1) \Delta x_j(t - \phi - 1) = \sum_{s=0}^{t} \hat{x}_j(s) \Delta x_j(t - s) =
\]
\[ = \sum_{s=0}^{s+1} \hat{x}_j(s) \Delta x_j(t-s) = \sum_{s=0}^{s} \hat{x}_j(s) \Delta x_j(t-s) = \hat{x}_j(t) \ast \Delta x_j(t) \]  \hspace{1cm} (5.A.7)

b) STATIC BRANCHES:

\[ v_j(t) = g_j(x_j(t), \alpha_j(t), t) \]

Let us define the relationship between the branch variables \( \hat{x}_j \) and \( \hat{v}_j \) in \( \hat{G}_N \), in this case as

\[ \hat{x}_j(\tau) = \hat{v}_j(\tau) \frac{\partial g_j}{\partial x_j} \]

\[ = \Delta \hat{v}_j(\tau) \hat{g}_j(t-\tau) \]  \hspace{1cm} (5.A.8)

where \( \hat{g}_j(.) \) is implicitly defined. By differentiating

\[ \Delta v_j(t') = \begin{cases} (\partial g_j / \partial x_j) \Delta x_j(t') = \hat{g}_j(t') \Delta x_j(t'), & t' \geq t - \tau \\ 0, & t' < t - \tau \end{cases} \]  \hspace{1cm} (5.A.9)

\[ \hat{v}_j(t) \ast \Delta v_j(t) = \sum_{\phi=0}^{t} \hat{v}_j(\phi) \Delta v_j(t-\phi) = \sum_{\phi=0}^{t} \hat{v}_j(\phi) \hat{g}_j(t-\phi) \Delta x_j(t-\phi) = \hat{x}_j(t) \ast \Delta x_j(t) \]  \hspace{1cm} (5.A.10)

Very interesting particular cases of (5.A.8) for non-linear adaptive filters or neural networks are:

\[ \begin{cases} \hat{x}_j(\tau) = w_j(t-\tau) \hat{v}_j(\tau) & \text{for a weight branch} \hspace{1cm} (5.A.11) \\ \hat{x}_j(\tau) = f_j(x_j(t-\tau)) \hat{v}_j(\tau) & \text{for a non-linear branch} \hspace{1cm} (5.A.12) \end{cases} \]

the last two equations correspond to the branches defined in (5.2.3), respectively.

c) \( \hat{y} \) – BRANCHES : with regard to the input branches \( \hat{y}_j \) of the SFG \( \hat{G}_N \), let us define

\[ \hat{y}_j(\tau) = \begin{cases} 0 & \text{if } j \neq k \\ 1, \tau = 0 & \text{if } j = 1, \ldots, r \\ 0, \tau > 0 & \text{if } j = k \end{cases} \]  \hspace{1cm} (5.A.13)

where \( y_k \) is the output of which we shall compute the gradient.
d) $\hat{u}$ – BRANCHES: on the output branches of $\hat{G}_N$ no definition is needed, and since the inputs $u_j$, $j = 1, \ldots, m$ of $G_N$ are obviously not dependent on the perturbation $\Delta u'$, it holds
\[ \Delta u_j(t) = 0 \quad \forall t \quad , \quad j=1,\ldots,m \]  
(5.A.14)

therefore
\[ \hat{u}_j(t) \ast \Delta u_j(t) = 0 \quad , \quad j=1,\ldots,m \]  
(5.A.15)

It holds:
\[ \Delta u_{m+1}(t) = \Delta u'(t) \quad \forall t \]  
(5.A.16)

therefore
\[ \hat{u}_{m+1}(t) \ast \Delta u_{m+1}(t) = \hat{u}'(t) \ast \Delta u'(t) = \sum_{\phi=0}^{1} \hat{u}'(\phi)\Delta u'(t-\phi) = \hat{u}'(\tau)\Delta u'(t-\tau) \]  
(5.A.17)

At this point we have completely defined the reversed signal flow graph $\hat{G}_N$ which is now called Adjoint Signal Flow Graph $\hat{G}_N^{(a)}$. To sum up: the Adjoint SFG $\hat{G}_N^{(a)}$ is defined as a reversed SFG $\hat{G}_N$ with the additional conditions given by (5.A.5) and (5.A.6) for delay branches, (5.A.11) and (5.A.12) (or (5.A.8)) for static branches, and (5.A.13) for the $\hat{y}$-branches. These definitions are summarized in Table 5.3.1.

Combining (5.A.3),(5.A.7),(5.A.10),(5.A.13),(5.A.15) and (5.A.17) results in
\[ \Delta y_k(t) = \hat{u}'(\tau)\Delta u'(t-\tau) \]  

Since $\hat{u}'(\tau)$ obviously does not depend on $\Delta u'$ we get
\[ \frac{\partial y_k(t)}{\partial u'(t-\tau)} = \lim_{\Delta u'\to0} \frac{\Delta y_k(t)}{\Delta u'(t-\tau)} = \hat{u}'(\tau) \]  
(5.A.18)

Now, from Fig. 5.A.1a we observe that
\[ \frac{\partial y_k(t)}{\partial u'(t-\tau)} = \frac{\partial y_k(t)}{\partial v_i(t-\tau)} \]  
(5.A.19)

and from Fig. 5.A.1b that $\hat{v}_i(\tau) = \hat{u}'(\tau)$, therefore
\[ \frac{\partial y_k(t)}{\partial v_i(t-\tau)} = \hat{v}_i(\tau) \]  
(5.A.20)

QED.
Chapter 6

SIGNAL-FLOW-GRAPH APPROACHES: FORWARD COMPUTATION

6.1 INTRODUCTION

In this chapter, a SFG gradient computation method will be outlined that is the dual approach of the Backward Computation method of the previous chapter. It is named Forward Computation method since it reflects a forward chain rule expansion of derivatives (Section 3.2), while the BC approach implements a backward expansion. While the BC method (batch mode) is local in space but not in time, the FC method is local in time but not in space. In this context, a local algorithm in space or in time is defined as one for which the calculation of a certain variable requires other variables that are ‘close’ to the first one, respectively in space or in time.

The computational complexity of FC increases with the square of the number of components instead that of BC has a linear dependence, but it depends also (online mode) on the length of the past history considered (linearly).

First of all, let’s introduce some notation and mathematical properties.

Let’s consider a function

\[ y = f(w) = f(w_1, \ldots, w_w), \]  

(6.1.1)

given in an open set \( A \subset R^w \). It can be expressed by

\[ y = \varphi(u) = \varphi(u_1, \ldots, u_m) \]  

(6.1.2)
where
\[ u_j = \psi_j(w) \quad (j = 1, \ldots, m; \ w \in A). \]  

(6.1.3)

Each vector \( w \) in \( A \) correspond to the vector \( u = (\psi_1(w), \ldots, \psi_m(w)) \).

Let’s assume that \( f(w), \varphi(u) \) and \( \psi_j(w) (j = 1, \ldots, m) \) have continuous first partial derivatives then
\[ \frac{\partial y}{\partial w_i} = \sum_{k=1}^{m} \frac{\partial y}{\partial u_k} \frac{\partial u_k}{\partial w_i} \]  

(6.1.4)

If also the second order derivatives are continuous:
\[ \frac{\partial^2 y}{\partial w_j \partial w_i} = \frac{\partial}{\partial w_j} \left( \frac{\partial y}{\partial w_i} \right) = \sum_{k=1}^{m} \frac{\partial}{\partial w_j} \left( \frac{\partial y}{\partial u_k} \frac{\partial u_k}{\partial w_i} \right) = \]
\[ = \sum_{k=1}^{m} \left[ \sum_{p=1}^{m} \left( \frac{\partial^2 y}{\partial u_p \partial u_k} \frac{\partial u_k}{\partial w_j} \frac{\partial u_k}{\partial w_i} + \frac{\partial y}{\partial u_k} \frac{\partial^2 u_k}{\partial w_j \partial w_i} \right) \right] \]  

(6.1.5)

In this way, the partial first or second derivative of \( y \) is expressed in terms of the dependent variables \( u_k \).

These properties will be useful in the following.

### 6.2 Sensitivity Calculation of Non-linear Dynamical Circuits by Forward Computation (FC)

Let’s consider a non-linear dynamic system represented by a Signal Flow Graph with inputs \( u_i, \ i = 1, \ldots, m \), outputs \( y_j, \ i = 1, \ldots, r \), and parameters \( w_j \) and \( \alpha_j \) of which the system output sensitivity has to be computed. Please, refer to Section 5.2 for a complete definition and notation description of Signal Flow Graphs, that is here only summarized.

A discrete time formulation will be presented, but the theory can be reformulated in the continuous time case.

Let’s assume that the parameters are constant for the moment.

In the case of discrete time systems, the functional relationship between the variables \( x_j \) and \( v_j \) of the \( f \)-branch \( j \) (in general expressed as \( v_j = f_j(x_j) \)) can be detailed in this way:
\[
\begin{align*}
\forall j \in \mathbb{N}, \quad & v_j(t) = g_j(x_j(t), \alpha_j(t), t) \quad \text{for a static branch (no memory)} \quad (6.2.1) \\
\forall j \in \mathbb{N}, \quad & v_j(t) = q^{-1}(x_j(t)) = x_j(t-1) \quad \text{for a delay branch (one unit memory)} \quad (6.2.2)
\end{align*}
\]

where \( q^{-1} \) is the delay operator and \( g_j \) is a general differentiable function, depending on the adaptable parameter (or vector of parameters) \( \alpha_j \). \( g_j \) can change on time, i.e. is directly dependent on time \( t \).

For a static branch the relationship between \( x_j \) and \( v_j \) is often specified as

\[
\begin{align*}
\forall j \in \mathbb{N}, \quad & v_j(t) = w_j(t)x_j(t) \quad \text{for a weight branch} \\
\forall j \in \mathbb{N}, \quad & v_j(t) = f_j(x_j(t)) \quad \text{for a non-linear branch}
\end{align*}
\]

where \( w_j(t) \) is the \( j \)-indexed weight parameter of the system at time \( t \), and \( f_j \) is a differentiable function. In neural networks context, \( w_j(t) \) can be a weight and \( f_j \) a sigmoidal activation function.

More in general (6.2.1) can represent a non-linear adaptable function controlled by a parameter (or vector) \( \alpha_j \), such as a spline (Vecci et al. 1997).

The general connection (feedback allowed) of the components in (6.2.3), (6.2.1) and (6.2.2), gives the overall numerical circuit (SFG) that implements the system model for which training or sensitivity computation is desired, see Fig. 5.2.2. The components described are conceptually equivalent to electrical components for electrical circuit and the relation between the SFG and the implemented system is the same as that between the electrical circuit and the system it realizes.

In the following, a SFG method will be presented for forward gradient computation that generalizes and unifies the ordered derivatives (Section 3.2) and automatic differentiation (Section 3.3) approaches (forward mode), named Forward Computation (FC) (Campolucci 1996b).

To compute the derivative of \( y(t) \) with respect to a parameter \( w_i \) (for \( \alpha_j \) it is analogous), in a forward way, let’s consider the various operators of the SFG.

**Summation node.** \( x_j(t) = v_k(t) + \ldots + v_p(t) \):

\[
\frac{\partial x_j(t)}{\partial w_i} = \frac{\partial v_k(t)}{\partial w_i} + \ldots + \frac{\partial v_p(t)}{\partial w_i} \quad (6.2.4)
\]

**Weight static branch.** \( v_j(t) = w_jx_j(t) \):
\[
\begin{align*}
\frac{\partial v_j(t)}{\partial w_i} &= \frac{\partial v_j(t)}{\partial x_j(t)} \frac{\partial x_j(t)}{\partial w_i} = w_j \frac{\partial x_j(t)}{\partial w_i} \quad j \neq i \\
\frac{\partial v_j(t)}{\partial w_i} &= x_i(t) + w_i \frac{\partial x_j(t)}{\partial w_i} \quad j = i
\end{align*}
\]

(6.2.5)

Non-linear static branch, \( v_j(t) = f_j(x_j(t)) \):

\[
\frac{\partial v_j(t)}{\partial w_i} = \frac{\partial v_j(t)}{\partial x_j(t)} \frac{\partial x_j(t)}{\partial w_i} = f_j'(x_j(t)) \frac{\partial x_j(t)}{\partial w_i}
\]

(6.2.6)

Delay branch, \( v_j(t) = q^{-1} x_j(t) = x_j(t-1) \):

\[
\frac{\partial v_j(t)}{\partial w_i} = \frac{\partial x_j(t-1)}{\partial w_i} = q^{-1} \frac{\partial x_j(t)}{\partial w_i}
\]

(6.2.7)

(6.2.8)

The initial states are not dependent on the parameters \( w_i \), therefore null initial conditions can be used:

\[
\frac{\partial v_j(0)}{\partial w_i} = 0
\]

(6.2.9)

Input branch, \( u_i \):

The input variables are not dependent on the parameters \( w_i \), therefore

\[
\frac{\partial u_j(t)}{\partial w_i} = 0
\]

(6.2.10)

From the SFG \( G \), other SFGs can be derived, called derivative SFGs \( (G_i^D) \), which correspond to the derivative systems \( S_i^D \), by using the transformation rules in Table 6.2.1.

The circuits associated with the derivative SFGs are linear and time-variant. Each of them has only one non null input, i.e. \( x_i \) and gives the desired derivative as output, i.e. \( \frac{\partial y(t)}{\partial w_i} \).

Therefore, to compute \( \frac{\partial y(t)}{\partial w_i} \) the signal that is the input of the \( i \)-th parameter must be injected to the node at the output of the parameter in the derivative SFG, as in Fig. 6.2.1 and 6.2.2.
### Table 6.2.1  Derivative SFG construction rules.

<table>
<thead>
<tr>
<th>Branch classification</th>
<th>Original signal flow graph</th>
<th>Derivative signal flow graph for computing ( \frac{\partial y(t)}{\partial w_i} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>( x_j \xrightarrow{q^{-1}} v_j )</td>
<td>( \frac{\partial x_j(t)}{\partial w_i} \xrightarrow{q^{-1}} \frac{\partial v_j(t)}{\partial w_i} )</td>
</tr>
<tr>
<td>Static</td>
<td>( x_j \xrightarrow{f_j(t)} v_j )  ( v_j(t) = f_j(x_j(t)) )</td>
<td>( \frac{\partial x_j(t)}{\partial w_i} \xrightarrow{f'_j} \frac{\partial v_j(t)}{\partial w_i} ) ( \frac{\partial f_j(x_j(t))}{\partial w_i} = f'_j(x_j(t)) \frac{\partial x_j(t)}{\partial w_i} )</td>
</tr>
<tr>
<td></td>
<td>( x_j \xrightarrow{w_j} v_j )</td>
<td>( \frac{\partial x_j(t)}{\partial w_i} \xrightarrow{w_i} j = i ) ( \frac{\partial x_j(t)}{\partial w_i} \xrightarrow{w_i} \frac{\partial v_j(t)}{\partial w_i} j \neq i )</td>
</tr>
<tr>
<td>Input</td>
<td>( u_j )</td>
<td>0</td>
</tr>
</tbody>
</table>

**Fig. 6.2.1**  Derivative system for the general SFG of Fig. 5.2.2.
Since each of the derivative SFG has about the same complexity of the original one, the complexity of this method is roughly the same as the forward phase for each component of the gradient, so for $W$ weights the complexity for computing the gradient is $W$ times the forward complexity. This is true since each of them has a different temporal evolution (internal state) so $W$ different systems are necessary for $W$ weights.

At this point, we want to relax the hypothesis of constant weights. Let’s consider the following expansion of the gradient:

$$\frac{\partial y(t)}{\partial w_i} = \sum_{\phi=0}^i \frac{\partial y(t)}{\partial w_i(\phi)}$$ (6.2.11)

For computing each term inside the summation, i.e. $\partial y(t)/\partial w_i(\phi)$, the method of the derivative system can be used with some corrections.

The static branch with $j = i$:

$$v_j(\tau) = w_j(\tau)x_j(\tau)$$ (6.2.12)

should be associated with

Fig. 6.2.2 Derivative system for sensitivity computation.
\[
\begin{align*}
\frac{\partial v_i(\tau)}{\partial w_j(\phi)} &= w_i(\tau) \frac{\partial x_i(\tau)}{\partial w_j(\phi)} \quad j = i, \quad \tau > \phi \\
\frac{\partial v_i(\phi)}{\partial w_j(\phi)} &= x_i(\phi) \quad j = i, \quad \tau = \phi
\end{align*}
\]

\tag{6.2.13}

instead of (6.2.5).

Therefore the derivative system can be constructed as before, but now its input must be a sequence with all zero but not at time \( \phi \) when its value is \( x_i(\phi) \). The output will be the derivative \( \frac{\partial y(t)}{\partial w_i(\phi)} \). Therefore for computing each term in (6.2.11) it is possible to do \( t+1 \) forward propagations with \( \phi \) from 0 to \( t \), but since the derivative system is linear the summation in (6.2.11) can be obtained directly, without computing each term, using as input the sequence \( x_i(\phi) \), \( \phi = 0,1,...,t \) (without zeros) and so by the derivative system defined with (6.2.5) and not (6.2.13).

For computing the gradient with respect to \( \alpha_i \) parameters the same theory holds with the only difference that the input to be injected is \( \frac{\partial g(x_i(t),\alpha_i(t),t)}{\partial \alpha_i(t)} \) instead of \( x_i(t) \). This gradient computation method will be named Forward Computation.

**6.3 NON-LINEAR DYNAMICAL SYSTEMS LEARNING BY FORWARD COMPUTATION**

**6.3.1 Cost functions and parameters updating rules**

Consider a discrete time non-linear dynamic system with inputs \( u_i, \quad i = 1,...,m \), outputs \( y_i, \quad i = 1,...,r \), and parameters \( w_j \) and \( \alpha_p \) which have to be adapted. The instantaneous global squared error at time \( t \) is defined as

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

\[(6.3.1)\]

where \( d_k(t), \quad k = 1,...,r \) are the desired outputs. So the global squared error over a time interval \([t_0,t_1]\) is
In the case of non-adaptive (or batch) training, we can choose as cost function, for example the global squared error over the whole learning epoch \( E(0,T) \), where \( T \) is the last instant of the epoch. Whereas if we want on-line training, we can choose to consider only the most recent errors \( e^2(t) \), for example using \( E(t - N_c + 1,t) \) as cost function (Nerrand et al. 1993, Narendra and Parthasaraty 1991). For truncated BPTT and RTRL algorithms (Williams and Peng 1990, Williams and Zipser 1989) this cost function with \( N_c=1 \) is used.

The choice of the MSE as cost function is arbitrary for the SFG method and any cost function that can be represented by a SFG with \( y_k \) and \( d_k \) as inputs and the cost as output can be considered, as explained in the following.

Considering gradient based learning algorithms with the variation of the weights calculated by the steepest-descent method, the parameters updating rule is

\[
\Delta w_i = -\mu \frac{\partial J}{\partial w_i} , \quad \mu > 0
\]

Where \( J \) is a cost function and \( \mu \) is the learning rate.

### 6.3.2 Gradient of a cost function

In the previous section, it was presented a method to compute the gradient of the output of a SFG with respect to its parameters, now we want to show how to extend this result for computing the gradient of a cost function of the outputs. The idea is very simple: to describe the cost as a SFG that follow the SFG of the trainable system in a cascade form. Let’s call the overall system \( S \) and the cost system \( S^E \). \( S \) is a one output system with \( y(t) = J \), and inputs \( u_i \), \( i = 1, \ldots, m \) and \( d_k \), \( k = 1, \ldots, r \). The partial derivative of \( J \) with respect to a parameter \( w_j \) is obtained as output of the derivative system \( S^D_j \), that is constructed from \( S \) by the transformation rules in Table 6.2.1, see Fig. 6.3.1.

It is easy to show that the Forward Computation method applied to Fully Recurrent Neural Networks gives the Real Time Recurrent Learning (RTRL) algorithm (Williams and Zipser 1989). Applied to IIR filters it gives the well known output error Recursive Prediction Error (RPE) method (Shynk 1989). It should be pointed out that, in the neural networks literature, is not known an application of the forward method to multilayer networks neither static nor dynamic (e.g. local feedback multilayer networks) to our best knowledge.
Chapter 6 SIGNAL-FLOW-GRAPH APPROACHES: FORWARD COMPUTATION

6.4 SIMULATION RESULTS

The results reported here refer to two problems of identification of non-linear dynamic systems. Locally recurrent neural networks (Section 2.4) have been chosen as adaptable models, and trained with an instantaneous error cost function.

The locally recurrent architecture used is the IIR-MLP (Section 2.4). The used network has two layers, three hidden neurons with hyperbolic tangent activation function, and one linear output neuron. Four different on-line learning algorithms were used: the Back-Tsoi algorithm (1991), the CRBP algorithm (Section 4.3), the BC and FC algorithms. The results are given in terms of Mean-Square-Error (MSE), expressed in dB, computed on the learning set after each epoch (after all the input-output samples were presented) and averaged over 10 runs, each with a different weight initialization.

The first set of experiments consisted in identifying the non-linear system with memory presented in (Back and Tsoi 1993):
\[ z(t) = 0.0154 x(t) + 0.0462 x(t-1) + 0.0462 x(t-2) + 0.0154 x(t-3) + \\
+ 1.99 z(t-1) - 1.572 z(t-2) + 0.4583 z(t-3) \]  
(6.4.1)

\[ y(t) = \sin[z(t)] \]

It is basically an ARMA model with long memory followed by a sinusoidal non-linearity. The input for the identification was a 1000 samples uniform random noise in the range \([-1,1]\). The \(x(t)\) and \(y(t)\) sequences were the training set for an IIR-MLP with all the synaptic filters in both layers with 2 zeros and 3 poles. Fig. 6.4.1 shows the learning plots for various training methods discussed in this work: FC, BC, CRBP, Back-Tsoi.

![Learning Plot](image)

**Fig. 6.4.1** Back-Tsoi test system identification results by an IIR-MLP network trained by four different learning methods. Learning rate: \(\mu = 0.01\). Truncation parameter for CRBP \(Q_z = 10\), for BC \(h = 11\).

The second set of experiments was carried out on the more realistic problem of identifying a base-band equivalent Pulse Amplitude Modulation (PAM) transmission circuit in presence of non linearity (Proakis 1989). An IIR-MLP similar to the one already described, but with all the synaptic filters in both layers with 3 zeros and 2 poles was used for identifying the PAM system, see §4.5.2 for more details.

Fig. 6.4.2 shows that BC and FC are equivalent and perform very well. In these simulations, BC and/or FC seem to perform better than CRBP but their
complexity is higher in both cases. The Back-Tsoi algorithm always exhibits poor performances.

![Figure 6.4.2](image_url)

**Fig. 6.4.2** PAM test system identification results by an IIR-MLP network trained by four different learning methods. Learning rate: $\mu = 0.01$. Truncation parameter for CRBP $Q=10$, for BC $h=10$.

### 6.5 CONCLUSIONS

Let’s consider the computation of the gradient (of a system output or cost function) for a dynamical system $S$ with respect to $W$ parameters $w_j$. By Forward Computation, $W$ derivative systems must be computed, one for each parameter. Since they are approximately of the same complexity of the original SFG, the complexity of FC is about $W$ times the complexity of the original SFG.

On the contrary, the Backward Computation method needs to compute only one adjoint SFG for all the components of the gradient so its complexity depends linearly on $W$ but, since the adjoint graph must be computed $h$ times ($h$ is the past history length), the complexity of the BC method is about $h$ times the complexity of the original graph.

Therefore the ratio of the complexities for the two approaches is

$$\frac{C_{BC}}{C_{FC}} \approx \frac{h}{W} \quad (6.5.1)$$

An interesting application of the FC method can be on VLSI design of neural networks, since it would not require a specific circuit for implementing the
learning phase because it uses the forward (neural network) circuit. This implementation can save chip area and computes exact gradient, while other methods proposed in literature (Jabri and Flower 1992) for the same purpose are approximated.
Chapter 7

STABILITY TEST AND CONTROL

7.1 INTRODUCTION

The capabilities of Locally Recurrent Neural Networks (LRNNs) in performing on-line Signal Processing (SP) tasks are well known (Haykin 1996, Back and Tsoi 1991, Tsoi and Back 1994, Frasconi et al. 1992), (Chapters 1 and 2). In particular one of the most popular architecture is the Multi Layer Perceptron (MLP) with linear IIR temporal filter synapses (IIR-MLP) (Section 2.4).

IIR-MLP is theoretically motivated as a non-linear generalization of linear adaptive IIR filters (Shynk 1989) and as a generalization of the popular Time Delay Neural Networks (TDNNs) (Haykin 1996, Haykin 1994, Waibel et al. 1989), (Section 2.2). In fact, a TDNN can be viewed as a MLP with FIR temporal filter synapses (FIR-MLP) (Haykin 1994, Back and Tsoi 1991, Tsoi and Back 1994), (Section 2.2). Therefore the IIR-MLP is a generalization of FIR-MLP (or TDNN) allowing the temporal filters to have a recursive part.

Efficient training algorithms can be developed for general LRNNs and so the IIR-MLP (Chapter 4). They are based on Back Propagation Through Time of the error (Werbos 1990) to propagate the sensitivities through time and network layers (Section 3.5), and on a local recursive computation of output error Recursive Prediction Error (RPE) type (Williams and Zipser 1989, Shynk 1989), refer to Section 3.6. They are named Causal Recursive Back Propagation (CRBP) (Campolucci et al. 1995, Campolucci, Uncini and Piazza 1997a), (Section 4.3) and Truncated Recursive Back Propagation (TRBP) (Campolucci, Uncini and Piazza 1997b), (Section 4.4) and they differ in the technique implemented to employ on-line computation.
They are both on-line and local in space and in time, i.e. of easy implementation, and their complexity is limited and affordable. They generalize the Back-Tsoi algorithm (Back and Tsoi 1991), the algorithms in (Waibel et al. 1989, Frasconi et al. 1992), the one by Wan (1990) and standard Back Propagation.

Even if CRBP and TRBP performs in a quite stable manner, if the learning rate is chosen small enough by the user, they do not control the stability of the IIR synapses (for the IIR-MLP) or of the recursive filters (for general LRNNs). In the following, we will refer mostly to the IIR-MLP case but extensions to other LRNNs are possible and easy in most cases.

The same limitation of not controlling stability is found in all the literature of learning methods for RNNs and LRNNs, e.g. (Haykin 1994, Back and Tsoi 1991, Frasconi et al. 1992, Williams and Zipser 1989), (Chapter 3). The problem with general RNNs is that it is not easy to derive necessary and sufficient conditions for the coefficients of the network to assure asymptotic stability even in the time invariant case since the feedback loop include the non-linearity. On the other hand, for LRNNs the recursion is usually separated from the non-linearity, as in the IIR-MLP. Therefore in batch mode the overall IIR-MLP is asymptotically stable if and only if each of the IIR filters is asymptotically stable, i.e. all transfer functions poles have modules less than one (in the hypothesis of causality).

In the time-variant case (on-line adaptation) the above condition is not sufficient anymore. A “slow coefficient-variation” condition must be added to assure stability (Carini et al. 1997, Regalia 1995). Even if this is what the theory state, in practice the second condition is often ignored since for practical signals the condition on the poles is sufficient (Shynk 1989) and the cases in which this is not true are “pathological” (Shynk 1989). However, the second condition is important when the IIR filter must operate near the instability region (Regalia 1995).

On the contrary, on the linear IIR adaptive filter literature, there are various techniques that can be employed to control stability (Shynk 1989, Carini et al. 1997). The simplest is to monitor the poles and do not adapt when it will bring the poles outside the circle. Since, to monitor stability, efficient criteria are available such as Jury’s one that avoids to compute the poles, this method is simpler, but slower and less robust than pole projection methods (Shynk 1989). To make stability check easier, the IIR filter can be realized as cascade or parallel of second order sections (Shynk 1989, Regalia 1995) or in lattice form (Regalia 1995, Back and Tsoi 1992).

Other stability assurance techniques include the Hyperstable Adaptive Recursive Filter (HARF) which imposes conditions on the transfer function not easy to guarantee in practice (Carini et al. 1997).
This chapter presents a new technique to control stability of IIR adaptive filters based on the idea of intrinsically stable operations that makes possible to continually adapt the coefficients with no need of stability test or poles projection (Campolucci and Piazza 1998).

The coefficients are adapted in a way that intrinsically assures the poles to be in the unit circle. This makes possible to use an higher step size (also named learning rate here) potentially improving the fastness of adaptation with respect to methods that employ a bound on the learning rate, e.g. (Carini et al. 1997) or methods that simply do not control stability or that do not adapt in case of instability.

This method can be applied to various realizations: direct forms, cascade or parallel of second order sections, lattice form. It can be implemented to adapt a simple IIR adaptive filter or a locally recurrent neural network, such as the IIR-MLP.

In the next two sections, the Intrinsic Stability technique will be explained and then simulation results will be presented.

### 7.2 INTRINSIC STABILITY TECHNIQUE

The basic idea underlying this technique is to restrict the region of the values that the coefficients can take to the stability region. In the following, we will assume that the working conditions of the filter are such that the filter stability is assured if the time invariant conditions are satisfied. Otherwise, it is always possible to add to this new method a control of the step size that assures the “slow-varying” condition, but the development of such a bound for the step size is beyond the aim of this chapter.

#### 7.2.1 First Order Case

For the sake of clarity, the explanation of the method will start from the simplest case, i.e. the first order IIR filter.

The input-output relationship is:

\[
y(t) = w_0[t]x[t] + w_1[t]x[t-1] + v_1[t]y[t-1]
\]  

(7.2.1)

where \(y[t]\) is the output and \(x[t]\) the input of the filter.

The transfer function is:

\[
\frac{w_0[t] + w_1[t]z^{-1}}{1 - v_1[t]z^{-1}}
\]  

(7.2.2)

Since the pole in the \(z\) plane is \(v_1[t]\), the filter is stable if and only if
Now the idea is to introduce a non-linear compressing transformation from a new coefficient to be adapted to \( v_1[t] \):

\[
v_1[t] = \Psi(\hat{v}_1[t])
\]  

(7.2.4)

\( \hat{v}_1[t] \) is named virtual coefficient.

Choosing \( \Psi(.) \) as a non-linear squashing function in the range (-1,1), assures \( v_1[t] \) to satisfy the stability condition.

A good choice for \( \Psi(.) \) is the hyperbolic tangent:

\[
g(x) = \frac{1 - e^{-x}}{1 + e^{-x}}
\]  

(7.2.5)

Anyway, even if other choices are possible other than \( \Psi(.) = g(.) \), the \( \Psi(.) \) function is requested to be squashing in the range (-1,1), continuos and first order differentiable.

The new parameter \( \hat{v}_1[t] \) can be adapted instead of \( v_1[t] \) that will be computed by (7.2.4). To adapt \( \hat{v}_1[t] \) a gradient descent method can be employed with only a small modification with respect to the adaptation of \( v_1[t] \), as explained in the following.

\[
\hat{v}_1[t+1] = \hat{v}_1[t] - \mu[t] \frac{\partial e^2[t]}{\partial \hat{v}_1[t]}
\]  

(7.2.6)

where \( e[t] = d[t] - y[t] \), \( \mu[t] \) is the step size and \( d[t] \) the target signal.

In the following, we will assume that the cost function is the instantaneous squared error but other cost functions, such as an average over a running window, can also be easily implemented.

It holds

\[
\frac{\partial e^2[t]}{\partial \hat{v}_1[t]} = 2e[t] \frac{\partial e[t]}{\partial \hat{v}_1[t]} = -2e[t] \frac{\partial y[t]}{\partial \hat{v}_1[t]}
\]  

(7.2.7)

and

\[
\frac{\partial y[t]}{\partial \hat{v}_1[t]} = \frac{\partial y[t]}{\partial v_1[t]} \frac{\partial v_1[t]}{\partial \hat{v}_1[t]} = \frac{\partial y[t]}{\partial v_1[t]} \Psi'(\hat{v}_1[t])
\]  

(7.2.8)

The well known recursive expressions to compute the derivative of the IIR filter output with respect to the denominator coefficients (output error RPE) can still
be used with the only correction of the multiplication by the derivative of the 

\( \Psi(.) \) function as stated by the last expression. For the hyperbolic tangent it simply holds:

\[
g'(x) = \frac{1}{2} \left[ 1 - g^2(x) \right]
\] (7.2.9)

That is very simple to compute since \( g(x) \) is already computed by (7.2.4). Therefore after adapting the \( \hat{v}_1[t] \) coefficient the corresponding \( v_1[t] \) must be computed by (7.2.4). Then only \( v_1[t] \) will be used to compute the new filter output by (7.2.1).

The effect of the squashing function is the following. When the pole of the filter \( v_1[t] \) is going near the boundary of the stability region, \( \Psi(.) \) will be computed in the flat region so that \( \Psi'(.) \) will be small and the adaptation slower. A part of the effect is like decreasing the step size when the filter is going to instability; anyway, it must be stressed that this technique cannot be viewed just as a control of the step size. The reason is that not only “the step size is decreased” but also the coefficient is bounded by (7.2.4), therefore assuring stability even if the step size is not decreased too much.

It seems reasonable that any technique that just controls the step size even by some accurate bound will have to reduce the step size more than the technique implementing (7.2.4) and so it will be slower. The Intrinsic Stability (IS) technique allows to avoid any stability test or pole projection method.

### 7.2.2 Higher Order Cases

This method can be easily applied to the lattice form since in that case the stability conditions state that the “reflection coefficients” must be less than one in absolute value. Therefore for any order of the filter the IS method can assure the stability of the lattice form simply applying the squashing transformation to each of the reflection coefficients and proceeding as explained.

The IS technique can be easily extended to second order sections, allowing the use of parallel and cascade form, as in the following.

The transfer function of a second order IIR filter is

\[
\frac{w_0[t] + w_1[t]z^{-1} + w_2[t]z^{-2}}{1 - v_1[t]z^{-1} - v_2[t]z^{-2}}
\] (7.2.10)

The necessary and sufficient conditions for stability are now
\[
\begin{align*}
\begin{cases}
|v_2[t]| < 1 \\
|v_1[t]| < 1 - v_2[t]
\end{cases}
\end{align*}
\]  
(7.2.11)

for each \(t\).

Therefore the squashing transformation is to be done as:

\[
v_2[t] = \Psi(\hat{v}_2[t])
\]  
(7.2.12)

\[
v_1[t] = (1 - v_2[t])\Psi(\hat{v}_1[t])
\]  
(7.2.13)

Where the second equation must be computed after the first one, since the new value of \(v_2[t]\) must be available.

The last two expressions allow satisfying exactly the stability conditions with no restriction of the stable region.

The two \(\hat{v}_1[t]\) and \(\hat{v}_2[t]\) coefficients can be independently adapted by gradient descent and then \(v_1[t]\) and \(v_2[t]\) can be computed by the last two expressions.

To generalize the IS method to a general order IIR filter in direct form, the poles must be represented in module and phase form. The modules can be squashed in the range \((0,1)\) by a unipolar sigmoid:

\[
f(x) = \frac{1}{1 + e^{-x}}
\]  
(7.2.14)

or again by \(g(x)\) in the range \((-1,1)\) to make easier to realize the pole in the origin.

On the other hand, the phases can be adapted with no restriction since they do not influence stability.

A gradient descent in the virtual modules (i.e. before compression) and in the real phases of the poles can be implemented. Then the real modules must be found evaluating \(\Psi(\cdot)\) and then the coefficients of the denominator of the transfer function computed from the poles. Must be stressed that although is necessary to implement the transformation from poles to coefficients is not necessary to implement the inverse from coefficients to poles, that is much more complex.

The other overhead of the IS technique in the general order case with respect to the first or second order case is that the derivatives of the coefficients with respect to the real modules and phases of the poles must be computed to implement the gradient descent in the poles space as a modification of the standard output error RPE. The formulas are not complex to derive and compute but they are not shown here.
7.3 SHAPE ADAPTATION OF THE SQUASHING FUNCTION

In any of the realization forms of the IIR filter, a number of squashing functions must be implemented equal to the order of the filter for a single IIR filter or for each of the synapses of the IIR-MLP neural network.

Therefore a question should be addressed: is it reasonable to implement exactly the same squashing function for the entire IIR filter or IIR-MLP network? Or may be the function should be optimized, in each case?

We believe that a much improved technique should adapt the squashing function depending on the parameter to be squashed. The simulations performed confirm this idea.

A good way to optimize $\Psi(.)$ is to make it automatically adaptable because the number of coefficients can be so big that is not realistic to optimize each function by hand. Moreover, there is no way to a-priori determine the shape of $\Psi(.)$ without accounting for the current adaptation process.

Therefore, we propose the use of non-linear squashing functions that can be automatically and independently adapted.

To save complexity it is very reasonable to implement $\Psi(.)$ as an hyperbolic tangent with adaptive slope, i.e.:

$$g(x, s) = \frac{1 - e^{-sx}}{1 + e^{-sx}} \quad (7.3.1)$$

where $s$ is named slope which must be adapted by gradient descent independently from the coefficients of the filter.

In this way, the slope of the compression can be optimized for each coefficient to be squashed while the permitted range is (-1,1) unchanged.

Since it holds:

$$\frac{\partial g(x, s)}{\partial x} \bigg|_{x=0} = \frac{s}{2} \quad (7.3.2)$$

it is clear that increasing $s$ make the function $g(.)$ closer to the threshold and therefore the IS technique can easier, i.e. faster, place poles near the unit circle.

This can be important when the requested behavior is near the instability that moreover is just the case when a stability control should be employed. The simulations verify this intuition.
The gradient descent on $s$ can be employed by the analogous of (7.2.6), (7.2.7) and the following expression in the first order case (extension is feasible)

$$\frac{\partial y[t]}{\partial s[t]} = \frac{\partial y[t]}{\partial v_1[t]} \frac{\partial v_1[t]}{\partial s[t]} = \frac{\partial y[t]}{\partial v_1[t]} \frac{\partial \Psi(\hat{v}_1[t], s[t])}{\partial s[t]}$$  \hspace{1cm} (7.3.3)

For the hyperbolic tangent it holds:

$$\frac{\partial g(x, s)}{\partial s} = \frac{x}{2} [1 - g^2(x, s)]$$  \hspace{1cm} (7.3.4)

that again is easy to compute since $g(x, s)$ is already calculated.

The IS technique with Adaptive slope will be named AIS method.

### 7.4 SIMULATIONS RESULTS

In this section, the AIS method will be applied to the training of an IIR-MLP neural network by the TRBP algorithm.

The test problems chosen are two difficult problems of on-line non-linear dynamical system identification.

The neural networks used have 2 layers, 3 hidden neurons with hyperbolic tangent activation function, 1 linear output neuron.

The results are shown in terms of plots of the MSE (in dB) vs. iterations (each iteration is an entire learning epoch) averaged over 20 runs each with a different coefficient initialization. The variance is also shown on the top right corner of the plots.

The first experiment is a non-linear ARMA system identification problem proposed in (Back and Tsoi 1993), (§4.5.1), simulated under the same conditions. The network used has all the synapses with 2 zeros and 2 poles.

Fig. 7.4.1 shows the improved performances of the AIS technique over not use any stability control. In this case, the adaptation of the slope is necessary.

The second test is the identification of a 16-PAM transmission channel in presence of a non-linearity, see §4.5.2 for details.

The network used has all the synapses with 4 zeros and 2 poles.

Fig. 7.4.2 again shows the improved performances of the AIS technique over the standard method that does not use any stability control.

The learning rate used for the simulations plotted is approximately the best possible for each method, i.e. standard and AIS. For AIS an higher learning rate can be used due to the stabilization effect of the method.
Fig. 7.4.1  IIR-MLP learning performances identifying the Back-Tsoi test system by the TRBP(8,2) algorithm implementing: standard method (no stability control) and AIS technique. $\mu=0.03$.

Fig. 7.4.2  IIR-MLP learning performances identifying the 16-PAM test system by the TRBP(8,2) algorithm implementing: standard method and AIS technique. $\mu=0.01$ for TRBP(8,2) standard. For AIS: $\mu=0.05$. 
Chapter 8

SECOND ORDER LEARNING METHODS

8.1 INTRODUCTION

The Backward Computation (BC) – Forward Computation (FC) theory is derived on Chapters 5 and 6 for first order learning algorithms, but an extension is possible to second order methods. This extension is left to the reader, but some new techniques to implement second order methods are proposed on this Chapter, on which the BC-FC methods can be usefully applied.

Several learning algorithms for neural networks have been proposed in the literature and many of them are based on the gradient descend algorithm, well known in optimization theory. Second-order algorithms can have better performances than first order ones, because they also use the second-order information stored in the Hessian matrix. There are several examples of these algorithms in literature (Battiti 1992, Bishop 1992), but a sub-class of them, based on the conjugate gradient method, has shown good properties in terms of rate of convergence and computational complexity.

Conjugate directions methods are based on choosing the search direction and the step size of a minimization formula by using second order information. It holds

\[ E(w + y) = E(w) + \nabla_w E(w)^T y + \frac{1}{2} y^T H(w) y \quad (8.1.1) \]

where \( E(w) \) is a generic cost function, of weights \( w \), to be minimized, \( H(w) \) is the Hessian matrix, and \( y \) the weight variation.
The Conjugate Gradient (CG) algorithm is based on the following two iterative formulas respectively for updating weights \( w_k \) and conjugate directions \( p_k \) (Fletcher and Reeves 1964):

\[
\begin{align*}
\mathbf{w}_{k+1} &= \mathbf{w}_k - \frac{\mathbf{p}_k^T \nabla E(\mathbf{w}_k)}{\mathbf{p}_k^T H(\mathbf{w}_k) \mathbf{p}_k} \mathbf{p}_k \\
\mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \frac{\mathbf{r}_{k+1}^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{r}_k} \mathbf{p}_k
\end{align*}
\]  

(8.1.2)  

(8.1.3)

where \( k \) is the iteration index and \( \mathbf{r}_k = -\nabla E(\mathbf{w}_k) \).

This algorithm has two drawbacks. The first is that for each iteration the Hessian matrix \( H(\mathbf{w}_k) \) has to be calculated and stored; the second one is that this algorithm works only for functions with positive definite Hessian matrices, and the quadratic approximations can be very poor when the current point is far from a local minimum.

Möller (1993) has proposed a solution to the second problem based on the Levenberg-Marquardt algorithm combined with the conjugate gradient approach. The problem of the Hessian matrix definition is solved trying to make always positive the quantity in the denominator of (8.1.2), adding a positive term, which is determined recursively. This algorithm is called Scaled Conjugate Gradient (SCG) (Möller 1993) and results to be better than CG in terms of convergence properties.

With respect to the first problem, some methods exist to extract information on the Hessian matrix without calculating or storing it and without making numerical approximations.

From (8.1.2) we note that the algorithm needs to calculate the vector \( H(\mathbf{w}_k) \mathbf{p}_k \), and not only the matrix \( H(\mathbf{w}_k) \). Therefore, having an efficient technique to directly obtain the product \( H(\mathbf{w}_k) \mathbf{p}_k \), there is no need for calculating and storing the whole Hessian matrix.

Therefore, in this work two methods for calculating such product have been applied to SCG, obtaining two computationally efficient algorithms (Campolucci, Simonetti et al. 1998), with good speed of convergence but without the complexity and memory usage typical of second order methods, already known in literature.

The methods derived have been applied to the difficult problem of training recurrent neural networks both in off-line and on-line mode. Several simulation results for non-linear system identification tests, known in literature, by locally
Recurrent neural networks are reported for both the off-line and on-line case, comparing with first order algorithms and showing a faster training.

The complexity of the proposed methods is accurately compared with that of the corresponding first order algorithms showing an average increase of about 2-3 times in terms of number of operations per iteration.

In this chapter, the two new algorithms called SCG-R and SCG-u, that apply SCG and respectively the two techniques described above, are presented referring to locally recurrent neural networks, comparing their performances with RBP and TRBP (Chapter 4). Although for the sake of brevity they are presented only for IIR-MLP networks, they can be easily extended to any locally recurrent network.

### 8.2 Hessian Information Extraction

The first method (Pearlmutter 1994) allows calculating the product $Hp$, where $p$ is a generic vector and $H$ is the Hessian matrix, by a simple, accurate and numerically robust technique, using the following operator

$$R_p \{ f(w) \} = \frac{\partial}{\partial r} f(w + rp) \bigg|_{r=0}$$

(8.2.1)

that gives

$$R_p \{ \nabla E(w) \} = \frac{\partial}{\partial r} \nabla E(w + rp) \bigg|_{r=0} = H(w)p$$

(8.2.2)

The previous formula uses the gradient of the cost function $E$ to calculate the vector $H(w)p$, therefore the computational complexity is reduced with respect to the direct use of second order derivatives.

Because $R_p \{ \}$ is a differential operator, it obeys to the usual rules for differential operators, in particular

$$R_p \{ cf(w) \} = c R_p \{ f(w) \}$$

$$R_p \{ f(w) + g(w) \} = R_p \{ f(w) \} + R_p \{ g(w) \}$$

$$R_p \{ f(w)g(w) \} = R_p \{ f(w) \}g(w) + f(w)R_p \{ g(w) \}$$

Moreover, it holds (Pearlmutter 1994)

$$R_p \{ w \} = p$$

(8.2.3)

These rules are sufficient to derive a new set of equations on a new set of variables called $R$-variables, from the equations used to compute the gradient.
This equations set can be considered an adjoint system to the gradient calculation that computes the vector \( R_p \{ \nabla E(w) \} \), which is the desired vector \( H(w)p \).

The second method (Boray and Srinath 1992) performs the calculation of \( H(w)p \), where \( p \) is now the conjugate gradient and \( H \) the Hessian matrix, as a difference between two gradients, computed in two different points.

This technique has four main phases:

1. compute the gradient \( \nabla_w E(w) \) of the cost function with respect to the weights vector \( w \);
2. compute \( u = w - \nabla_w E(w) \);
3. compute the gradient \( \nabla_u E(u) \) of the cost function with respect to the vector \( u \);
4. compute:
   \[
   H(w)p = \nabla_u E(w) - \nabla_u E(u) \tag{8.2.4}
   \]
as a difference between the two gradients already calculated.

Must be stressed that both the two methods above must be performed every iteration, i.e. every step of the conjugate gradient descend.

While the first one gives an exact computation of the product \( H(w)p \), the second one gives only an approximation of it because \( E(w) \) is not a quadratic cost function, in general.

The two previous techniques result to be simple and efficient and can be applied to any neural network (static or dynamic). They allow exploiting the Hessian properties without explicitly calculating it, and using only first order formulas.

Moreover, the second technique has also an important implementation advantage, because it uses directly the same formulas as the gradient calculation.

Must also be stressed that the two previous techniques have never been applied to training neural networks by the SCG algorithm, particularly the second one has been implemented only for adaptive filtering (Boray and Srinath 1992). The SCG combined with the two techniques discussed above gives two new algorithms which are faster than the first order counterpart on some tasks, as shown in the section on simulation results.
After using these algorithms with some static problems, with good performances, we have applied them to train locally recurrent neural networks, as explained in the next sections.

**8.3 THE SCG-R AND SCG-U SECOND ORDER ALGORITHMS**

With the same notation used in Chapter 4, let consider the generic neuron \( k \) in layer \( l \) of a IIR-MLP neural network with inputs \( x \), weights \( w \) and targets \( d \), trained by an epoch of \( T \) time steps. The vector of all the weights of a whole IIR-MLP network is given by

\[
\tilde{w} = \begin{bmatrix} w \\ v \end{bmatrix}
\]

where

\[
w = \begin{bmatrix} w^{(1)} & \ldots & w^{(l)} & \ldots & w^{(M)} \end{bmatrix}^T
\]

\[
w^{(l)} = \begin{bmatrix} w_{10(l)}^{(l)} & \ldots & w_{kj(p)}^{(l)} & \ldots & w_{N_i(N_i+1)}^{(l)}(l_{N_i-1,N_i}^{(l)} - 1) \end{bmatrix}
\]

\[
v = \begin{bmatrix} v^{(1)} & \ldots & v^{(l)} & \ldots & v^{(M)} \end{bmatrix}^T
\]

\[
v^{(l)} = \begin{bmatrix} v_{10(l)}^{(l)} & \ldots & v_{kj(p)}^{(l)} & \ldots & v_{N_i(N_i+1)}^{(l)}(l_{N_i-1,N_i}^{(l)} - 1) \end{bmatrix}
\]

Therefore the generic component of the conjugate direction, because of its definition, will depend on index \( k, j \) and \( l \), giving the following vector

\[
\tilde{p} = \begin{bmatrix} p \cdot w \\ p \cdot v \end{bmatrix}
\]

where

\[
p \cdot w = \begin{bmatrix} p \cdot w^{(1)} & \ldots & p \cdot w^{(l)} & \ldots & p \cdot w^{(M)} \end{bmatrix}^T
\]

\[
p \cdot v = \begin{bmatrix} p \cdot v^{(1)} & \ldots & p \cdot v^{(l)} & \ldots & p \cdot v^{(M)} \end{bmatrix}^T
\]

\[
p \cdot w^{(l)} = \begin{bmatrix} p \cdot w_{10(l)}^{(l)} & \ldots & p \cdot w_{kj(p)}^{(l)} & \ldots & p \cdot w_{N_i(N_i+1)}^{(l)}(l_{N_i-1,N_i}^{(l)} - 1) \end{bmatrix}
\]

\[
p \cdot v^{(l)} = \begin{bmatrix} p \cdot v_{10(l)}^{(l)} & \ldots & p \cdot v_{kj(p)}^{(l)} & \ldots & p \cdot v_{N_i(N_i+1)}^{(l)}(l_{N_i-1,N_i}^{(l)} - 1) \end{bmatrix}
\]
As stated above, applying the two previous techniques to calculate the products $H(\tilde{w})\tilde{p}$, we obtain two new algorithms based on the conjugate gradient method. These algorithms contain only first order formulas, i.e. gradient calculation. A critical point here is to use an efficient gradient calculation scheme; to this purpose the RBP and TRBP algorithms have been chosen for the application to locally recurrent networks.

In the following, we show the batch and on-line versions of the two new second-order algorithms for training locally recurrent neural networks.

### 8.3.1 The SCG-R algorithm

Applying the $R_p$ operator first to the forward and then to the backward phase of the RBP algorithm (Section 4.2) we have a set of formulas that represent the SCG-R algorithm in batch mode.

Because of the $\tilde{w}$ and $\tilde{p}$ definition, it holds

$$R_p \left\{ \frac{\partial E}{\partial w_{mm(p)}} \right\} = [R_p \{\nabla E(\tilde{w})\}]_{mm(p)^{(i)}} = [H(\tilde{w}) \tilde{p}]_{mm(p)^{(i)}} \tag{8.3.1}$$

As stated above this is a batch mode algorithm that cannot work on-line (Section 4.3), but we can obtain an on-line version of SCG-R simply applying the $R_p$ operator to the TRBP algorithm that is on-line.

The forward phase is the same as in the previous case. Applying the $R_p$ operator to the backward phase we get the formulas shown in the Appendix.

We note that in practice the computational complexity and the memory usage of SCG-R is about 2 times that of RBP (or TRBP), because we must compute and store the adjoint $R_p \{ \}$ terms. However, the entire algorithm is performed using only first order formulas.

### 8.3.2 The SCG-u algorithm

In this case, we calculate the products $H(\tilde{w})\tilde{p}$ using the following expression

$$H(\tilde{w})\tilde{p} = \nabla_{\tilde{w}} E(\tilde{w}) - \nabla_{\tilde{u}} E(\tilde{u}) \tag{8.3.2}$$

where

$$\tilde{u} = \tilde{w} - \nabla_{\tilde{w}} E(\tilde{w}) \ . \tag{8.3.3}$$
The crucial point of the $\nabla_\sim E(\tilde{u})$ computation is given by applying RBP (or TRBP for on-line mode) also to a second network that has the same structure of the first one, but with weights vector $\tilde{u}$ instead of $\tilde{w}$.

Therefore, we have the same formulas of RBP, or TRBP, applied to a set of different variables called $u$-variables.

Computational complexity and memory storage of the SCG-$u$ are about double with respect to RBP (or TRBP) algorithm, because in practice RBP (or TRBP) is performed two times for each iteration: first to calculate $\nabla_\sim E(\tilde{w})$ and then to calculate $\nabla_\sim E(\tilde{u})$.

We have omitted the SCG-$u$ formulas because they are the same as for RBP (or TRBP in the on-line case).

### 8.4 SIMULATIONS RESULTS

In this section, the results of applying the new SCG-$R$ and SCG-$u$ algorithms to two problems of identification of non-linear dynamic systems from literature are presented.

The locally recurrent architecture chosen for the simulations is the IIR-MLP with two layers: three hidden neurons with hyperbolic tangent activation function and one linear output neuron.

Both for batch and for on-line mode we compare SCG with the corresponding first-order algorithms, respectively RBP and TRBP. The results are given in terms of Mean-Square-Error (MSE) expressed in dB, and its variance, computed on the learning set after each epoch (after all the input-output samples were presented) and averaged over 20 runs, each with a different weights initialization.

The first set of experiments consists in identifying the non-linear system with memory presented in (Back and Tsoi 1993). The network used has both MA and AR order equal to three for each neuron. From Fig.s 8.4.1 and 8.4.2 is clear that the new SCG algorithms have good performances.
Chapter 8 SECOND ORDER LEARNING METHODS

Fig. 8.4.1  Back-Tsoi test results in batch mode: MSE and its variance.

![Graph showing MSE (dB) vs. iterations (epochs) with labeled curves for different algorithms like SCG-R, SCG-u, RBP, TRBP(10,10), SCG-(R,u) (10,10).]

Fig. 8.4.2  Back-Tsoi test results in on-line mode, using SCG-{R,u} (h,h') or TRBP(h,h'). h is the past history length considered and h' is every how many samples parameters are adapted.

The second set of experiments was carried out on the more realistic problem of identifying a base-band equivalent Pulse Amplitude Modulation (PAM) transmission system in presence of non linearity, see §4.5.2 for details. The network used in this case has MA and AR order respectively equal to four and two for each neuron. Fig.s 8.4.3 and 8.4.4 show that the new SCG algorithms perform very well on this test.
In these simulations the second-order algorithms seem to perform better than first-order ones. Moreover the complexity of SCG algorithms, in terms of number of floating point operations (flop) per iteration, is only about 2 times higher with respect to RBP and TRBP, as shown in Tables 8.4.1 and 8.4.2. This is an important result because other second-order algorithms already presented in literature have in general a much higher complexity.

<table>
<thead>
<tr>
<th>Learning Method</th>
<th>Back-Tsoi (1000 samples)</th>
<th>P.A.M. (2048 samples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBP</td>
<td>1.336e+005</td>
<td>1.422e+005</td>
</tr>
<tr>
<td>SCG-R</td>
<td>3.438e+005</td>
<td>3.531e+005</td>
</tr>
<tr>
<td>SCG-u</td>
<td>2.665e+005</td>
<td>2.701e+005</td>
</tr>
</tbody>
</table>

Table 8.4.1  
*Computational complexity on the Back-Tsoi and P.A.M. (batch mode) tests in terms of number of flop per iteration.*
8.4.2 Computational complexity on the Back-Tsoi and P.A.M. (on-line mode) tests in terms of number of flop per iteration.

<table>
<thead>
<tr>
<th>Learning Method</th>
<th>Back-Tsoi (1000 samples)</th>
<th>P.A.M. (2048 samples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRBP(10,10)</td>
<td>1.852e+006</td>
<td>2.692e+006</td>
</tr>
<tr>
<td>SCG- R (10,10)</td>
<td>3.726e+006</td>
<td>5.361e+006</td>
</tr>
<tr>
<td>SCG- u (10,10)</td>
<td>3.534e+006</td>
<td>5.065e+006</td>
</tr>
</tbody>
</table>

Table 8.4.2

8.A APPENDIX: THE SCG-R ALGORITHM (ON-LINE MODE)

The following formulas are obtained applying the operator $R\{,\}$ to the formulas of forward and backward phases of the TRBP algorithm for an IIR-MLP neural network (Section 4.4).

Forward phase:

$$R\{y^{(i)}_{nm}[t]\} = \sum_{p=0}^{i(t-1)} (w^{(i)}_{nm(p)} R\{x^{(i-1)}_m[t-p]\} + p w^{(i)}_{nm(p)} x^{(i-1)}_m[t-p]) +$$

$$+ \sum_{p=1}^{i(t)} (v^{(i)}_{nm(p)} R\{y^{(i)}_{nm}[t-p]\} + p v^{(i)}_{nm(p)} y^{(i)}_{nm}[t-p])$$

$$R\{s^{(i)}_n[I]\} = \sum_{m=0}^{N_i} R\{x^{(i)}_{nm}[I]\} \quad R\{\dot{x}^{(i)}_n[I]\} = R\{s^{(i)}_n[I]\} \tilde{g}_m(s^{(i)}_n[I])$$

Backward phase:

$$R\{\Delta \tilde{w}^{(i)}_{nm(p)}[I + 1]\} = -\frac{1}{2} \sum_{p=-h+1}^{h} R\{\Delta \tilde{w}^{(i)}_{nm(p)}[I + 1]\}$$

where, respectively for the MA and AR parts of each synapsis,

$$R\{\Delta w^{(i)}_{nm(p)}[I + 1]\} = R\{\tilde{s}^{(i)}_n[I] \frac{\partial s^{(i)}_n[I]}{\partial w^{(i)}_{nm(p)}} + \delta^{(i)}_n[I] R\{ \frac{\partial s^{(i)}_n[I]}{\partial w^{(i)}_{nm(p)}} \} \}$$

$$R\{\Delta v^{(i)}_{nm(p)}[I + 1]\} = R\{\tilde{s}^{(i)}_n[I] \frac{\partial s^{(i)}_n[I]}{\partial v^{(i)}_{nm(p)}} + \delta^{(i)}_n[I] R\{ \frac{\partial s^{(i)}_n[I]}{\partial v^{(i)}_{nm(p)}} \} \}$$

where
\[ R \left( \frac{\partial s_{n}^{(l)}[t]}{\partial w_{nm}^{(l)(p)}} \right) = R\{v_{n}^{(l-1)}[t-p]\} + \sum_{r=1}^{l_{m}} \left( R\{v_{n}^{(l)(p)}\} \frac{\partial s_{n}^{(l)}[t-r]}{\partial w_{nm}^{(l)(p)}} + v_{n}^{(l)} R\left( \frac{\partial s_{n}^{(l)}[t-r]}{\partial w_{nm}^{(l)(p)}} \right) \right) \]

\[ R \left( \frac{\partial s_{n}^{(l)}[t]}{\partial v_{nm}^{(l)(p)}} \right) = R\{y_{n}^{(l-1)}[t-p]\} + \sum_{r=1}^{l_{m}} \left( R\{v_{n}^{(l)(p)}\} \frac{\partial s_{n}^{(l)}[t-r]}{\partial v_{nm}^{(l)(p)}} + v_{n}^{(l)} R\left( \frac{\partial s_{n}^{(l)}[t-r]}{\partial v_{nm}^{(l)(p)}} \right) \right) \]

and

\[ R\{s_{n}^{(l)}[t]\} = R\{e_{n}^{(l)}[t]\} \text{sgn}(s_{n}^{(l)}[t]) + e_{n}^{(l)}[t] \text{sgn}^{\prime}(s_{n}^{(l)}[t]) R\{s_{n}^{(l)}[t]\} \]

and where the variables \( \frac{\partial s_{n}^{(l)}[t]}{\partial w_{nm}^{(l)(p)}}, \frac{\partial s_{n}^{(l)}[t]}{\partial v_{nm}^{(l)(p)}} \), \( \delta_{n}^{(l)}[t] \), are defined as in the TRBP algorithm.

Moreover, it holds

\[
\begin{cases} 
-R\{x_{n}^{(M)}[t]\} & t \in [\tau - N_{c} + 1.\tau] \\
0 & t \in [\tau - h + 1.\tau - N_{c}] 
\end{cases} \quad l = M
\]

\[ R\{e_{n}^{(l)}[t]\} = \sum_{q=1}^{N_{c}} \sum_{k=-N_{c}-1}^{N_{c}} \left( R\{s_{q}^{(l+1)}[k]\} \frac{\partial y_{qn}^{(l+1)}[k]}{\partial x_{n}^{(l)}[t]} + \delta_{q}^{(l+1)}[k] R\left( \frac{\partial y_{qn}^{(l+1)}[k]}{\partial x_{n}^{(l)}[t]} \right) \right) \quad l = M - 1
\]

\[ R\{s_{q}^{(l+1)}[k]\} = \sum_{q=1}^{N_{c}} \sum_{k=-N_{c}-1}^{N_{c}} \left( R\{s_{q}^{(l+1)}[k]\} \frac{\partial y_{qn}^{(l+1)}[k]}{\partial x_{n}^{(l)}[t]} + \delta_{q}^{(l+1)}[k] R\left( \frac{\partial y_{qn}^{(l+1)}[k]}{\partial x_{n}^{(l)}[t]} \right) \right) \quad l < M - 1
\]

where

\[ R \left( \frac{\partial y_{qn}^{(l+1)}[t+p]}{\partial x_{n}^{(l)}[t]} \right) = \begin{cases} R\{e_{n}^{(l+1)}[t+p]\} & 0 \leq p \leq L_{qn}^{(l+1)} - 1 \\
0 & \text{otherwise} \end{cases} + \sum_{r=1}^{\min(L_{qn}^{(l+1)},p)} \left( p \frac{\partial y_{qn}^{(l+1)}[t+p-r]}{\partial x_{n}^{(l)}[t]} + v_{n}^{(l+1)} R\left( \frac{\partial y_{qn}^{(l+1)}[t+p-r]}{\partial x_{n}^{(l)}[t]} \right) \right) \]


Int. Conference on Acoustic, Speech and Signal Processing, Atlanta (USA), May 1996.


