**UNIVERSITAT POLITÈCNICA DE CATALUNYA**

*Departament de Llenguatge i Sistemes Informàtics*
*Ph.D. Programme: Artificial Intelligence*

# SYMBOLIC AND CONNECTIONIST LEARNING TECHNIQUES FOR GRAMMATICAL INFERENCE

Autor: René Alquézar Mancho
Director: Alberto Sanfeliu Cortés

March 1997

# Chapter 4

# Grammatical inference through connectionist approaches

In this chapter, an overview of the most significant previous works on grammatical inference by means of neural networks is given. Most of them have dealt with the problem of learning regular languages, and just a few have addressed the problem of learning CFLs. The reader is referred to the original papers and the excellent survey by Castaño and Casacuberta [CaCa:96] for further information.

The chapter begins with a review of the neural network architectures that have been proposed to approach the problem of grammatical inference. These include several types of discrete-time recurrent neural networks, which can be applied indeed to a variety of learning tasks involving sequences. For each type of architecture, there is one or more learning algorithms that can be used to train the network for a given task.

The reported connectionist methods for GI have been based either on a next-symbol prediction task or on a sequence classification task. While the former allows an inference to be made from just positive strings, the latter requires the presentation of both positive and negative examples in the training set. Typically, a neural learning scheme is applied to infer a network that is supposed to act as a recognizer for the language from which the training set is taken. To this end, several factors must be fixed, such as the symbol encoding, the training procedure (including the stop criterion), and the learning parameters. In some cases, a post-processing step is added to extract from the trained network a symbolic representation of the language, usually an FSA.

## 4.1 Architectures and learning algorithms

A great number of different neural network architectures have been proposed and studied in the last decades, either aimed at supervised or unsupervised learning tasks [RuMc:86,HeKP:91]. The popular *multi-layer feed-forward networks* trained by the *back-propagation* algorithm [RuMc:86] are well-suited to the approximation of functions consisting of static vector mappings from a given set of points (input/output vector pairs). They can also be applied to some sequential tasks involving a time-varying input/output by using a moving window that selects a fixed number of consecutive events in the input sequence as the input vector (*time-delay networks*). However, feed-forward networks are not adequate in general for sequential tasks, since the sequences may be arbitrarily long and distant contingencies in the past sequence may determine the correct output at the present step.

Hence, *recurrent neural networks* (RNNs) [HeKP:91], which are naturally dynamic, have been preferred for learning and processing sequences because of their inherent ability to model "internal state" information and process temporal signals. RNNs can be classified in two main groups depending on whether they operate in *continuous-time* or in *discrete-time*. It is clear that *discrete-time* RNNs are the proper choice for learning tasks that involve discrete-time signals, such as symbol strings. Consequently, the reported connectionist approaches to grammatical inference have used different types of *discrete-time* RNNs, which are reviewed in the next subsections. From now on, *discrete-time* RNNs will be simply referred to as RNNs, for short.

In order to emphasize the common features of some of the proposed RNNs and to define a suitable notational framework for the studies presented in Chapters 6 and 7, two basic general models, termed *single-layer RNN* (SLRNN) and *augmented single-layer RNN* (ASLRNN) respectively, which have been introduced firstly by Goudreau *et al.* [GoGi:93, GoGC:94], will be presented. These general models cover different architectures as particular cases, where the specific network to be used in a learning task is normally fixed before the training begins. A distinct approach is to construct incrementally a network during the course of learning with the aim of reaching a kind of "optimal" network for the given task. The following review will also include an architecture of this type that has been applied to the GI problem, the *Recurrent Cascade-Correlation* (RCC) architecture proposed by Fahlman [Fahl:91a]. Finally, the *DOLCE architecture* reported by Das and Mozer [DaMo:94], which integrates a clustering module into an ASLRNN, and the *first-order 2-layer RNN* proposed by Manolios and Fanelli [MaFa:94] will be reviewed.

### 4.1.1 Single-layer recurrent neural networks (SLRNNs)

An SLRNN is a *fully-connected* discrete-time recurrent network model (see Figure 4.1) that has $M$ inputs, which are labelled $x_1, x_2, ..., x_M$, and a single-layer of $N$ units (or neurons) $U_1, U_2, ..., U_N$, whose output (or activation values) are labelled $y_1, y_2, ..., y_N$. The values at time $t$ of inputs $x_i$ $(1 \leq i \leq M)$ and unit outputs $y_j$ $(1 \leq j \leq N)$ are denoted by $x_i(t)$ and $y_j(t)$ respectively. The activation values of the neurons represent collectively the state of the SLRNN, which is stored in a bank of latches for a delay of one time step. Each unit computes its output value based on the current state vector $\mathbf{S}^t = [y_1(t-1), y_2(t-1), ..., y_N(t-1)]^T$ and the input vector $\mathbf{I}^t = [x_1(t), x_2(t), ..., x_M(t)]^T$, so the network is fully-connected. Some number $P$ of the neurons $(1 \leq P \leq N)$, called the *output units*, can be trained and used to supply an output vector $\mathbf{O}^t = [y_1(t), y_2(t), ..., y_P(t)]^T$ in order to accomplish a given task. In the most general case, the trainable (output) units may vary at each time step. Those neurons that are never trained are called *hidden units*.

The equations that describe the dynamic behavior of an SLRNN are

$$\sigma_k(t) = f\,(\mathbf{W}_k, \mathbf{I}^t, \mathbf{S}^t) \quad \text{for } 1 \leq k \leq N, \tag{4.1}$$

$$y_k(t) = g\,(\sigma_k(t)) \quad \text{for } 1 \leq k \leq N, \tag{4.2}$$

where $f$ is a weighted sum of terms that combines inputs and received activations to give the net input values $\sigma_k$, $g$ is usually a non-linear function, and $\mathbf{W}_k$ is a vector of weights that are associated with the incoming connections of unit $U_k$. The SLRNNs can be classified according to the types of their $f$ and $g$ functions, which will be referred to as the aggregation and activation functions of the SLRNN, respectively.

The usual choices for the *aggregation function* $f$ characterize an SLRNN either as *first-order* type [WiZi:89]

$$f\,(\mathbf{W}_k, \mathbf{I}^t, \mathbf{S}^t) = \sum_{i=1}^{M} w_{ki} x_i(t) + \sum_{j=1}^{N} w_{k(M+j)} y_j(t-1), \tag{4.3}$$

or *second-order* type [GiMC:92]

$$f\,(\mathbf{W}_k, \mathbf{I}^t, \mathbf{S}^t) = \sum_{i=1}^{M} \sum_{j=1}^{N} w_{kij}\; x_i(t)\; y_j(t-1). \tag{4.4}$$

In the first-order case, the SLRNN has $N \times (M+N)$ weights, while in the second-order case the number of weights rises to $N^2 \times M$.

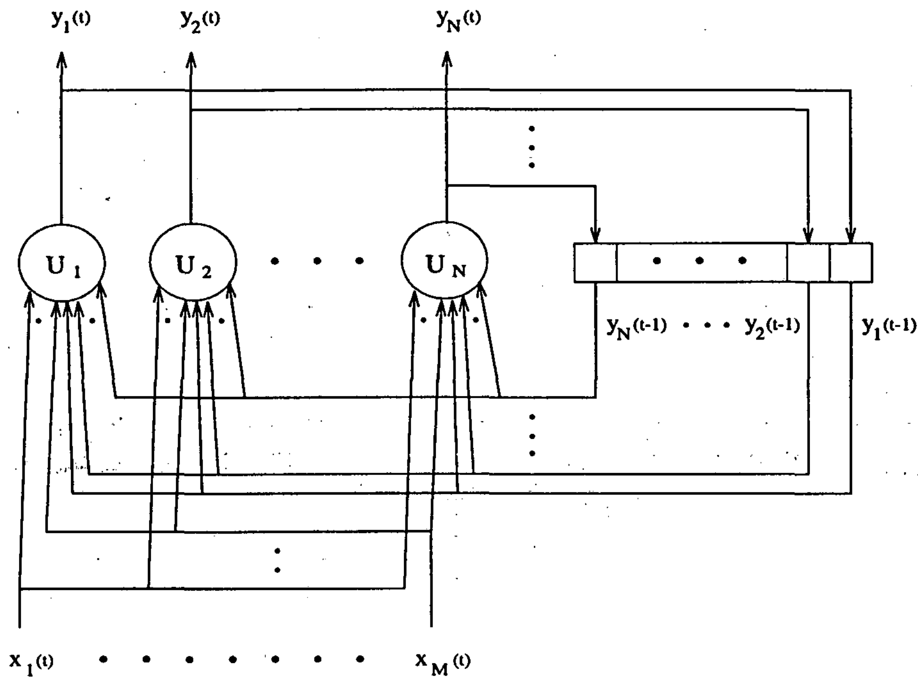


**Fig. 4.1** *Single-layer recurrent neural network (SLRNN).*

The usual choice for the *activation function* $g$ has been the *sigmoid* function

$$g_s(\sigma) = \frac{1}{1+e^{-a\sigma}}, \tag{4.5}$$

where $a$ is a positive constant (normally $a = 1$). However, other types of activation function can be selected instead of the sigmoid function, as it will be discussed later. Any differentiable function $g$ allows the use of a gradient descent method to train the SLRNN to learn a sequential task.

#### *4.1.1.1 First-order SLRNNs*

A formal description of the first-order SLRNNs was given by Williams and Zipser together with the RTRL learning algorithm [WiZi:89], although they simply called them "fully recurrent neural networks" in their paper. The notation used here slightly differs from theirs to adapt it to the general SLRNN model just defined. The basic difference resides on the instant when the time step delay is introduced: while they put the delay in the computation of the activation function, we put it after, assuming that the rate of external input update is slower than the net computation speed.

In a *first-order* SLRNN, each unit has weighted connections from all the neurons of the net, each unit receives weighted signals from all the inputs, and each unit can be trained at each time step if desired. From (4.1), (4.2) and (4.3), we obtain a single system of equations that describes the dynamic behavior of a first-order SLRNN

$$y_k(t) = g\left(\sum_{i=1}^{M} w_{ki}x_i(t) + \sum_{j=1}^{N} w_{k(M+j)}y_j(t-1)\right) \quad \text{for } 1 \le k \le N, \tag{4.6}$$

where the sigmoid function $g_s$ given by (4.5) has been typically used as activation function $g$ [WiZi:89,SmZi:89]. To allow each unit to have a bias weight, an input whose value is always 1 is included among the $M$ input signals. Let us say that the network starts running at time step $t = 1$ and the corresponding state vector $\mathbf{S}^1 = [y_1(0), ..., y_N(0)]^T$ is initialized in some arbitrary manner.

For notational convenience, let us concatenate the current input and state vectors, $\mathbf{I}^t$ and $\mathbf{S}^t$, to form an $(M+N)$-dimension vector $\mathbf{Z}^t = [z_1(t), ..., z_{M+N}(t)]^T$, such that

$$z_j(t) = \begin{cases} x_j(t) & \text{if } j \le M \\ y_{j-M}(t-1) & \text{if } j > M \end{cases} \tag{4.7}$$

Then, the network dynamics can be rewritten as follows

$$\sigma_k(t) = \sum_{j=1}^{M+N} w_{kj}z_j(t) \quad \text{for } 1 \le k \le N, \tag{4.8}$$

$$y_k(t) = g\,(\sigma_k(t)) \quad \text{for } 1 \le k \le N. \tag{4.9}$$

Now, let us discuss the supervised training of first-order SLRNNs. A training sequence with $t_f$ time steps starts at time step $t = 1$ and ends at time step $t = t_f$. Let $T(t)$ denote the set of indices of (output) units for which there exists a specified target value at time $t$, i.e. $T(t) = \{\ k \mid 1 \le k \le N,\ \exists\ d_k(t)\ \}$. Then we define the error of a unit $U_k$ at time step $t$ as

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T(t) \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

and the total squared error of the network at time step $t$ as

$$E(t) = \frac{1}{2} \sum_{k=1}^{N} [e_k(t)]^2. \tag{4.11}$$

The objective is the minimization of the total squared error over the whole training sequence.

$$E_{total}(1,t_f) = \sum_{t=1}^{t_f} E(t) \quad . \tag{4.12}$$

This minimization can be done by a gradient descent procedure, adjusting the weights $\mathbf{W}$ of the SLRNN along the negative of $\nabla_{\mathbf{W}} E_{total}(1,t_f)$, i.e.

$$\triangle w_{ij} = -\alpha \, \frac{\partial E_{total}(1,t_f)}{\partial w_{ij}} \qquad \text{for } 1 \leq i \leq N, \; 1 \leq j \leq M+N, \tag{4.13}$$

where $\alpha$ is some fixed positive learning rate.

The following learning algorithms have been proposed to train a first-order SLRNN:

1) the Back-Propagation Through Time (BPTT) algorithm [WiPe:90],
2) the Run-Time Recurrent Learning (RTRL) algorithm [WiZi:89],
3) the Schmidhuber's algorithm (a cross between BPTT and RTRL) [Schm:92].

The three algorithms above are gradient-descent learning techniques which differ in the way the gradient $\nabla_{\mathbf{W}} E_{total}(1,t_f)$ is calculated. Each algorithm has a distinct time complexity and different storage requirements (the better the time complexity, the worse the space requirements, and viceversa).

The BPTT algorithm is based on unfolding the recurrent network into a kind of multilayer feedforward network that grows by one layer on each time step [RuHW:86, WiPe:90]. In order to distinguish between different "instances" of the weight $w_{ij}$ at different times, $w_{ij}(t)$ will denote a variable for the weight of the $j$-th incoming connection of unit $U_i$ at time $t$; this is just for notational convenience, since $w_{ij}(t) = w_{ij}$ for all $t \in [1,t_f]$. In other words, $w_{ij}(t)$ can be seen as a weight of a unit in the $t$-th (non-input) layer of a feedforward network constructed by unfolding the SLRNN in time. Then, we have

$$\frac{\partial E_{total}(1,t_f)}{\partial w_{ij}} = \sum_{t=1}^{t_f} \frac{\partial E_{total}(1,t_f)}{\partial w_{ij}(t)} = -\sum_{t=1}^{t_f} \delta_i(t) z_j(t) \tag{4.14}$$

where

$$\delta_i(t) = -\frac{\partial E_{total}(1,t_f)}{\partial \sigma_i(t)} \tag{4.15}$$

and $\delta_i(t)$ can be computed for all $i \in [1,N]$ and $t \in [1,t_f]$ by the recursive backward propagation:

$$\delta_i(t) = \begin{cases} g'[\sigma_i(t)] \; e_i(t) & \text{if } t = t_f \\ g'[\sigma_i(t)] \left[ e_i(t) + \sum_{k=1}^{N} w_{k(M+i)} \delta_k(t+1) \right] & \text{if } 1 \leq t < t_f \end{cases} \tag{4.16}$$

The number of floating-point operations involved in the preceding calculation of the gradient is $O(t_f N^2)$, and hence, the BPTT algorithm needs only $O(N^2)$ computations per time step. However, it must be noted that the storage of the values $e_i(t)$, $g'[\sigma_i(t)]$, and $z_j(t)$ is required, for $1 \le i \le N$, $1 \le j \le M+N$, and $1 \le t \le t_f$. Therefore, BPTT requires potentially unlimited storage in proportion to the length of the longest training sequence, and thus, it cannot be used for arbitrarily long training sequences, i.e. when there is no known upper bound for the sequence length. In addition, BPTT is not suited for on-line learning, since weight changes can be performed only when the entire training sequence has been processed.

Williams and Zipser proposed the RTRL algorithm to perform gradient-descent learning in *continually running* first-order SLRNNs, allowing an indefinite length of the training sequence [WiZi:89]. The RTRL algorithm is based on a forward calculation of the gradient, and it can be applied both for off-line and on-line learning, though in the second case the weight change no longer follow the precise negative gradient of the total error.

Since the total error is just the sum of the errors at the individual time steps, one way to compute the gradient $\nabla_{\mathbf{W}} E_{total}(1, t_f)$ is by accumulating the values of $\nabla_{\mathbf{W}} E(t)$ for each time step. Thus, the overall weight change for any particular weight $w_{ij}$ in the network can be written as

$$\Delta w_{ij} = \sum_{t=1}^{t_f} \Delta w_{ij}(t), \tag{4.17}$$

where

$$\Delta w_{ij}(t) = -\alpha \frac{\partial E(t)}{\partial w_{ij}} = \alpha \sum_{k=1}^{N} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}} . \tag{4.18}$$

Let

$$p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ij}} \quad \text{for } 1 \le i,k \le N,\ 1 \le j \le M+N. \tag{4.19}$$

The partial derivatives $p_{ij}^k(t)$ can be computed forward in time using a dynamical system with variables $\{p_{ij}^k\}$ and dynamics given by

$$p_{ij}^k(t) = g'[\sigma_k(t)] \left[ \delta_{ik} z_j(t) + \sum_{l=1}^{N} w_{k(M+l)} p_{ij}^l(t-1) \right] \tag{4.20}$$

with initial conditions

$$p_{ij}^k(0) = 0, \tag{4.21}$$

where $\delta_{ik}$ denotes the Kronecker delta, and it is assumed that the initial state of the network has no functional dependence on the weights.

The RTRL algorithm starts by applying (4.21) to initialize the $p_{ij}^k$ variables, and afterwards, it consists of computing, at each time step from $t = 1$ to $t = t_f$, the values $p_{ij}^k(t)$, using Eq.(4.20), and then determining the errors $e_k(t)$, through (4.10), to compute the weight changes

$$\Delta w_{ij}(t) = \alpha \sum_{k=1}^{N} e_k(t)\, p_{ij}^k(t) . \qquad (4.22)$$

If an off-line learning scheme is selected, the overall correction to each weight $w_{ij}$ in the net, given by Eq.(4.17), is applied at the end of the training sequence ($t = t_f$). Otherwise, in an on-line learning scheme, each weight $w_{ij}$ is updated at each time step $t$ using the individual $\Delta w_{ij}(t)$ values given by Eq.(4.22).

The RTRL algorithm requires only fixed-size storage of the order $O(N^3)$ (due to the variables $p_{ij}^k$), but it is computationally expensive, requiring $O(N^4)$ operations per time step (due to the computation of the $p_{ij}^k(t)$ values).

Schmidhuber has proposed a third algorithm to compute the gradient $\nabla_{\mathbf{W}} E_{total}(1, t_f)$ for learning in first-order SLRNNs, which is a compromise between the BPTT and the RTRL methods [Schm:92]. His algorithm also requires fixed-size $O(N^3)$ storage, thus allowing arbitrarily long training sequences too, but the average time complexity per time step is reduced to $O(N^3)$.

Schmidhuber's algorithm is based on decomposing the calculation of the gradient into blocks, each covering a number $h$ of time steps, with $h$ in $O(N)$ (e.g. $h = N$). For each block, $N + 1$ BPTT-like passes are performed, one pass for calculating error derivatives, and $N$ passes for calculating derivatives of the net-inputs to the $N$ units at the end of each block. An RTRL-like calculation of the order $O(N^4)$ is performed at the end of each block for integrating the results of these BPTT-like passes into the results obtained from previous blocks. In fact, in the last block of the training sequence, a single BPTT-like pass to compute the error derivatives is only needed, and thus, when $t_f \leq h$ the algorithm is equivalent to BPTT. In general (for training sequences longer than the block size $h$), the algorithm performs an *average* of $O(N^3)$ computations per time step, since the $N + 1$ BPTT-like passes imply $O(N^3)$ computations per time step, and the RTRL-like $O(N^4)$ computations are spread over $O(N)$ time steps.

Let

$$q_{ij}^k(t) = \frac{\partial \sigma_k(t)}{\partial w_{ij}} = \sum_{\tau=1}^{t} \frac{\partial \sigma_k(t)}{\partial w_{ij}(\tau)} \qquad (4.23)$$

for $1 \leq i, k \leq N$, $1 \leq j \leq M + N$, and $t \geq 0$, where $w_{ij}(\tau)$ denotes the "instance" of $w_{ij}$ at time step $\tau$, as in the BPTT algorithm. We can think of $\{q_{ij}^k\}$ as a set of variables of a dynamical system that is updated once at each block of $h$ time steps; the

$\{q_{ij}^k\}$ variables play a similar role than the $\{p_{ij}^k\}$ variables used in the RTRL algorithm (though their meaning is not exactly the same). It is clear that the initial values of these variables are given by $q_{ij}^k(0) = 0$ for all possible $i, j, k$.

Let $t_0$ represent the beginning of the current block of $h$ time steps of the training sequence from $t = t_0+1$ to $t = t_0+h$ (or the beginning of the last block of the sequence from $t = t_0 + 1$ to $t = t_f$, if $t_0 < t_f \leq t_0 + h$; in this case, assume that the value $t_f - t_0$ is assigned to $h$). Then, both the gradient quantities $\partial E_{total}(1, t_0)/\partial w_{ij}$ and the values $q_{ij}^k(t_0)$ are already known for all appropriate $i, j, k$, either calculated in the previous block or initialized to zero if $t_0 = 0$.

The gradient $\nabla_\mathbf{W} E_{total}(1, t_0 + h)$ of the total error up to the time step $t_0 + h$ can be calculated as the sum of $\nabla_\mathbf{W} E_{total}(1, t_0)$ and $\nabla_\mathbf{W} E_{total}(t_0 + 1, t_0 + h)$. The latter can also be decomposed in two terms, distinguishing the effect on the current block error caused by the weights in the previous blocks from the effect caused by the weights in the current block. Hence, for any particular weight $w_{ij}$ in the network, we have the following equation with three terms:

$$\frac{\partial E_{total}(1, t_0 + h)}{\partial w_{ij}} = \frac{\partial E_{total}(1, t_0)}{\partial w_{ij}} + \sum_{t=1}^{t_0} \frac{\partial E_{total}(t_0 + 1, t_0 + h)}{\partial w_{ij}(t)} + \sum_{t=t_0+1}^{t_0+h} \frac{\partial E_{total}(t_0 + 1, t_0 + h)}{\partial w_{ij}(t)} \tag{4.24}$$

The first term of the above equation is already known, as remarked earlier. The third term can be calculated as

$$\sum_{t=t_0+1}^{t_0+h} \frac{\partial E_{total}(t_0 + 1, t_0 + h)}{\partial w_{ij}(t)} = - \sum_{t=t_0+1}^{t_0+h} \delta_i(t) z_j(t) \tag{4.25}$$

where

$$\delta_i(t) = - \frac{\partial E_{total}(t_0 + 1, t_0 + h)}{\partial \sigma_i(t)} \tag{4.26}$$

and $\delta_i(t)$ can be computed for all $i \in [1, N]$ and $t \in [t_0, t_0+h]$ by the recursive backward propagation of the BPTT algorithm (note the similarity with Eq.(4.16) of BPTT):

$$\delta_i(t) = \begin{cases} g'[\sigma_i(t)]\; e_i(t) & \text{if } t = t_0 + h \\ g'[\sigma_i(t)] \left[ e_i(t) + \sum_{k=1}^{N} w_{k(M+i)} \delta_k(t + 1) \right] & \text{if } t_0 \leq t < t_0 + h \end{cases} \tag{4.27}$$

except for the particular case of $t_0 = 0$ (in the first block of the training sequence), where, for $1 \leq i \leq N$, $\delta_i(0) = 0$ because $\sigma_i(0)$ is undefined.

The second term of Eq.(4.24) can be calculated as

$$\sum_{t=1}^{t_0} \frac{\partial E_{total}(t_0+1, t_0+h)}{\partial w_{ij}(t)} = -\sum_{k=1}^{N} \delta_k(t_0) q_{ij}^k(t_0) \tag{4.28}$$

where, for $1 \le k \le N$, $\delta_k(t_0)$ has been computed previously by Eq.(4.27) and the $q_{ij}^k(t_0)$ values are already known from the previous block. It is clear that in the first block of the sequence, when $t_0 = 0$, this second term vanishes, i.e. $\sum_{t=1}^{0} \frac{\partial E_{total}(1,h)}{\partial w_{ij}(t)} = 0$ for all the weights $w_{ij}$.

What remains is the updating of the $\{q_{ij}^k\}$ variables, which is needed except for the last block. It follows (see [Schm:92] for the derivation) that

$$q_{ij}^k(t_0+h) = \sum_{l=1}^{N} \gamma_{kl}(t_0) q_{ij}^l(t_0) + \sum_{t=t_0+1}^{t_0+h} \gamma_{ki}(t) z_j(t) \tag{4.29}$$

where

$$\gamma_{ki}(t) = \frac{\partial \sigma_k(t_0+h)}{\partial \sigma_i(t)} \tag{4.30}$$

and $\gamma_{ki}(t)$ can be computed for all $k, i \in [1, N]$ and $t \in [t_0, t_0+h]$ by $N$ BPTT-like propagations:

$$\gamma_{ki}(t) = \begin{cases} \delta_{ki} & \text{if } t = t_0 + h \\ g'[\sigma_i(t)] \sum_{l=1}^{N} w_{l(M+i)} \gamma_{kl}(t+1) & \text{if } t_0 \le t < t_0 + h \end{cases} \tag{4.31}$$

except for $t_0 = 0$, where $\gamma_{ki}(0) = 0$ for $1 \le k, i \le N$. Again, $\delta_{ki}$ denotes the Kronecker delta.

The Schmidhuber's algorithm may be used both for off-line and *quasi* on-line weight updating. In the first case, Eq.(4.13) is applied at the end of the last block, when the gradient of the total error over the sequence is known. In the second case, the weights are changed at the end of each block, using the gradient $\nabla_{\mathbf{W}} E_{total}(t_0+1, t_0+h)$ (this is, the sum of the second and third terms of Eq.(4.24)).

Concerning the storage requirements of the algorithm, besides the $\{q_{ij}^k\}$ variables, which imply an associated space size of $O(N^3)$ equivalent to that of RTRL, the values $e_i(t)$, $g'[\sigma_i(t)]$, and $z_j(t)$ must also be stored to perform the BPTT-like passes, for $1 \le i \le N$, $1 \le j \le M+N$, and $t_0 \le t \le t_0+h$. Since $h$ is $O(N)$, the storage of these values requires a space size of order $O(N^2)$. Finally, the $\{\gamma_{ki}\}$ variables need also $O(N^2)$ space. Hence, the total storage required by Schmidhuber's algorithm is of order $O(N^3)$, which is due to the $\{q_{ij}^k\}$ variables.

Normally, a training set $S$ consists of some finite number of training sequences $\{s_1, ..., s_K\}$. The three preceding algorithms permit to compute the gradient $\nabla_{\mathbf{W}} E^{s_i}_{total}(1, t_f(s_i))$ corresponding to a pass through a given training sequence $s_i$. A pass through the whole training set $S$ is called an *epoch*. During an epoch, the state of the network is reinitialized each time a new training sequence is started; obviously, the partial derivatives required by the particular learning algorithm must also be reset for each sequence. The total error over $S$ is defined simply as

$$E^S_{total} = \sum_{i=1}^{K} E^{s_i}_{total}(1, t_f(s_i)). \tag{4.32}$$

and the corresponding gradient $\nabla_{\mathbf{W}} E^S_{total}$ is just the sum of the gradients $\nabla_{\mathbf{W}} E^{s_i}_{total}(1, t_f(s_i))$ for all $i \in [1, K]$.

The procedure of updating the weights of the network only at the end of an epoch is called *batch* learning, while the procedure of updating the weights after each training sequence is called *pattern-by-pattern*[1] learning. Using a gradient-descent technique, the weights are adjusted along the negative of $\nabla_{\mathbf{W}} E^S_{total}$ in the former case, while in the latter, the negatives of the individual gradients $\nabla_{\mathbf{W}} E^{s_i}_{total}(1, t_f(s_i))$ are taken. In addition, we have seen that the RTRL algorithm allows an *on-line* learning with weight updating every time step, and the Schmidhuber's algorithm allows a *quasi on-line* learning with weight updating every block of $h$ time steps.

Whatever the mode of learning, a gradient-descent based algorithm updates the weights according to the general equation

$$\triangle^{\tau} \mathbf{W} = -\alpha \nabla_{\mathbf{W}} E(\mathbf{W}^{\tau}) \tag{4.33}$$

where $\tau$ is used as a counter of the number of weight updates computed so far during training, and $\nabla_{\mathbf{W}} E(\mathbf{W}^{\tau})$ is the gradient of the total error accumulated for the current inter-update interval, which is evaluated at the point given by the current weight vector $\mathbf{W}^{\tau}$. If the learning rate $\alpha$ is small, gradient descent can be very slow but yields a stable convergence on local minima of the error surface over the weight space. If $\alpha$ is too large, the trajectory of the weight vector can oscillate widely or may overshoot small minima basins. The problem essentially comes from error surface valleys with steep sides but a shallow slope along the valley floor. Furthermore, the error surface can be full of non-optimal local minima, some of them with large attractor basins, from which the algorithm may not be able to escape.

[1] Here, a pattern refers to a whole training sequence and not to an input-output pair within a sequence.

There are a number of ways of dealing with these problems, including the replacement of gradient descent by more sophisticated minimization algorithms[2] [HeKP:91], but a much simpler commonly used approach, the addition of a *momentum term*, is often effective. This scheme is implemented by giving a contribution from the previous change to the current weight update:

$$\Delta^\tau \mathbf{W} = -\alpha \nabla_\mathbf{W} E(\mathbf{W}^\tau) + \beta \Delta^{\tau-1}\mathbf{W} \tag{4.34}$$

where the *momentum parameter* $\beta$, $0 \leq beta < 1$, specifies the strength with which the previous update influences the current one. The net effect of the momentum term is that the local gradient deflects the trajectory through the weight space, but does not completely dominate it.

In this case, the recursive definition of $\Delta^\tau \mathbf{W}$ can be expanded in $\tau$ to give

$$\Delta^\tau \mathbf{W} = -\alpha \sum_{i=0}^{\tau-1} \beta^i \nabla_\mathbf{W} E(\mathbf{W}^{\tau-i}), \tag{4.35}$$

where it is clear that the weight update at any time $\tau$ is actually influenced by many local gradients, each evaluated at earlier points $\mathbf{W}^{\tau-i}$ along the trajectory. The result is an "average" gradient which tends to lead to fewer attractor basins by avoiding small minima basins. The larger the momentum parameter, the greater the averaging effect. Furthermore, for plateau regions of the error surface, the successive local gradients are about the same, and then it can be shown that Eq.(4.35) converges to

$$\Delta \mathbf{W} \approx -\frac{\alpha}{1-\beta} \nabla_\mathbf{W} E(\mathbf{W}) \tag{4.36}$$

with an effective learning rate of $\alpha/(1-\beta)$ [HeKP:91]. On the other hand, in an oscillatory situation, the weight update responds only with coefficient $\alpha$ to instantaneous fluctuations of the gradient. The overall effect is to accelerate the long term trend by a factor of $1/(1-\beta)$, without magnifying the oscillations.

The use of a momentum term seems adequate mainly with pattern-by-pattern or batch learning modes. If on-line learning is selected, it must be taken into account that the distribution of errors along the different time steps within a training sequence may be quite irregular, possibly with no error signal (zero gradient) during some time intervals.

[2] As far as we know, neither *conjugate gradient* methods nor *quasi-Newton* techniques have been adapted to be used in RNNs up to now.

#### 4.1.1.2 Second-order SLRNNs

Second-order SLRNNs have been proposed by Giles *et al.* [GiMC:92,GiSC:90,SuCG:90] and Pollack [Poll:91] to be used for grammatical inference. The fundamental feature that supports the use of second-order SLRNNs for the regular GI problem resides on the fact that the architecture is *implicitly* oriented to represent the transitions $\delta(state, input) = next_state$ of a DFA. The *explicit* representation of finite-state machines in both second-order and first-order SLRNNs will be studied in depth in Chapter 7.

In a *second-order* SLRNN, each unit has a weighted connection for each pair formed by an input and a unit. Each weight modifies the corresponding product of the input value by the unit activation value. As in first-order SLRNNs, each unit can be trained at each time step if desired. From (4.1), (4.2) and (4.4), we obtain a single system of equations that describes the dynamic behavior of a second-order SLRNN

$$y_k(t) = g\left( \sum_{i=1}^{M} \sum_{j=1}^{N} w_{kij}\, x_i(t)\, y_j(t-1) \right) \quad \text{for } 1 \leq k \leq N, \qquad (4.37)$$

where again the sigmoid function $g_s$ given by (4.5) has been typically used as activation function $g$ [Poll:91,GiMC:92]. As in first-order SLRNNs, the network starts running at time step $t = 1$ with a state vector $\mathbf{S}^1 = [y_1(0), ..., y_N(0)]^T$ that is arbitrarily initialized; an interesting alternative has been studied by Forcada and Carrasco, who have proposed to include the search of an optimal initial state in the learning algorithm [FoCa:95].

Some variations on the basic second-order SLRNN architecture given by (4.37) have been reported [MiGi:93,WaKu:92,ZeGS:93]. A bias weight $w_{k0}$ may be optionally added to each unit $U_k$ of the SLRNN, i.e.

$$y_k(t) = g\left( w_{k0} + \sum_{i=1}^{M} \sum_{j=1}^{N} w_{kij}\, x_i(t)\, y_j(t-1) \right) \quad \text{for } 1 \leq k \leq N, \qquad (4.38)$$

although it has been demonstrated empirically that the inclusion of bias weights does not improve the learning performance of the network [MiGi:93]. Watrous and Kuhn [WaKu:92] have used a type of second-order SLRNN, in which output units are not recurrent, that includes both first-order and second-order connections for each unit, i.e.

$$y_k(t) = g\left( \sum_{i=1}^{M} w_{ki0} x_i(t) + \sum_{i=1}^{M} \sum_{j=1}^{N-P} w_{kij}\, x_i(t)\, y_{P+j}(t-1) \right) \quad \text{for } 1 \leq k \leq N, \qquad (4.39)$$

where an input whose value is always 1 is included among the $M$ input signals. Nevertheless, the results that have been published for a regular GI task using

Tomita's languages [Tomi:82] as benchmark using the basic second-order SLRNN [GiMC:92,MiGi:93], and using the Watrous and Kuhn's architecture [WaKu:92], with a true gradient-descent training algorithm in both cases, have shown a better learning performance of the former model. Finally, Zeng *et al.* [ZeGS:93] have proposed a modification on the basic second-order SLRNN that mainly consists of using a discrete activation function $g_d$ when the network runs, but using the corresponding values given by a (differentiable) sigmoid activation function $g_s$ for training the net through a *pseudo-gradient* learning technique; this last variation will be discussed in more detail later.

Now, for clarity purposes, let us decompose the dynamics of a second-order SLRNN as

$$\sigma_k(t) = \sum_{i=1}^{M} \sum_{j=1}^{N} w_{kij}\, x_i(t)\, y_j(t-1) \quad \text{for } 1 \leq k \leq N, \tag{4.40}$$

$$y_k(t) = g\left(\sigma_k(t)\right) \quad \text{for } 1 \leq k \leq N. \tag{4.41}$$

As before, let us assume that a training sequence, from time step $t = 1$ to $t = t_f$, is given, where for each $t$, there are target values $d_k(t)$ for a certain subset of units $T(t)$. Then, the definitions of the errors $e_k(t)$, $E(t)$, and $E_{total}(1, t_f)$, given by Eqs. (4.10), (4.11), and (4.12), respectively, also apply here. Again, a gradient-descent procedure for error minimization implies adjusting the weights along the negative of $\nabla_{\mathbf{W}} E_{total}(1, t_f)$, i.e.

$$\Delta w_{kij} = -\alpha\, \frac{\partial E_{total}(1, t_f)}{\partial w_{kij}} \quad \text{for } 1 \leq k, j \leq N,\ 1 \leq i \leq M, \tag{4.42}$$

where $\alpha$ is some fixed positive learning rate, and a momentum term could be added optionally as discussed earlier.

The three gradient-descent learning algorithms for training first-order SLRNNs that have been described in the preceding subsection (BPTT, RTRL, and Schmidhuber's) may be adapted easily for training second-order SLRNNs as well. As far as we know, only the second-order version of RTRL has been reported previously [GiMC:92], in which the weight changes corresponding to each time step are given by

$$\Delta w_{kij}(t) = \alpha \sum_{l=1}^{N} e_l(t)\, \frac{\partial y_l(t)}{\partial w_{kij}} \tag{4.43}$$

and the partial derivatives $p^l_{kij}(t) = \partial y_l(t) / \partial w_{kij}$, $1 \leq l, k, j \leq N$, $1 \leq i \leq M$, can be computed forward in time using the dynamical system

$$p^l_{kij}(t) = g'[\sigma_l(t)] \left[ \delta_{lk} x_i(t) y_j(t-1) + \sum_{m=1}^{M} \sum_{n=1}^{N} w_{lmn} x_m(t)\, p^n_{kij}(t-1) \right] \tag{4.44}$$

with initial conditions

$$p^l_{kij}(0) = 0. \tag{4.45}$$

This second-order version of the RTRL algorithm can be applied on-line, using Eq.(4.43) directly, or off-line, updating the weights at the end of the training sequence according to the overall correction

$$\Delta w_{kij} = \sum_{t=1}^{t_f} \Delta w_{kij}(t). \tag{4.46}$$

The method requires fixed-size storage of the order $O(N^3M)$, due to the variables $p^l_{kij}$, and it is computationally expensive, requiring $O(N^4M^2)$ operations per time step, due to the computation of the $p^l_{kij}(t)$ values.

The BPTT algorithm for second-order SLRNNs takes the following form:

$$\frac{\partial E_{total}(1,t_f)}{\partial w_{kij}} = -\sum_{t=1}^{t_f} \delta_k(t)x_i(t)y_j(t-1) \tag{4.47}$$

where $\delta_k(t)$ can be computed for all $k \in [1,N]$ and $t \in [1,t_f]$ by the recursive backward propagation

$$\delta_k(t) = \begin{cases} g'[\sigma_k(t)]\; e_k(t) & \text{if } t = t_f \\ g'[\sigma_k(t)]\left[e_k(t) + \sum_{l=1}^{N}\delta_l(t+1)\left(\sum_{i=1}^{M} w_{lik}x_i(t+1)\right)\right] & \text{if } 1 \le t < t_f \end{cases} \tag{4.48}$$

In this case, the storage of the values $e_k(t)$, $g'[\sigma_k(t)]$, $x_i(t)$, and $y_j(t-1)$ is required, for $1 \le i \le M$, $1 \le k,j \le N$, and $1 \le t \le t_f$. On the other hand, the second-order version of the BPTT algorithm needs only $O(N^2M)$ computations per time step.

For arbitrarily long training sequences, an efficient learning algorithm for second-order SLRNNs can be defined by modifying adequately the Schmidhuber's gradient-descent algorithm. The method can be summarized as follows:

$$\begin{aligned} \frac{\partial E_{total}(1,t_0+h)}{\partial w_{kij}} &= \frac{\partial E_{total}(1,t_0)}{\partial w_{kij}} + \sum_{t=1}^{t_0} \frac{\partial E_{total}(t_0+1,t_0+h)}{\partial w_{kij}(t)} + \\ &\quad \sum_{t=t_0+1}^{t_0+h} \frac{\partial E_{total}(t_0+1,t_0+h)}{\partial w_{kij}(t)} \qquad (4.49) \\ &= \frac{\partial E_{total}(1,t_0)}{\partial w_{kij}} - \sum_{l=1}^{N} \delta_l(t_0)q^l_{kij}(t_0) - \sum_{t=t_0+1}^{t_0+h} \delta_k(t)x_i(t)y_j(t-1) \qquad (4.50) \end{aligned}$$

where the first term of (4.50) is already known from the previous block of time steps, the $\delta_k(t)$ can be computed for all $k \in [1, N]$ and $t \in [t_0, t_0 + h]$ as

$$\delta_k(t) = \begin{cases} g'[\sigma_k(t)]\ e_k(t) & \text{if } t = t_0 + h \\ g'[\sigma_k(t)]\ \left[e_k(t) + \sum_{l=1}^{N} \delta_l(t+1) \left(\sum_{i=1}^{M} w_{lik} x_i(t+1)\right)\right] & \text{if } t_0 \le t < t_0 + h, \end{cases} \tag{4.51}$$

and the $q^l_{kij}(t_0)$ values are also known from the previous block.

Initially, for $t_0 = 0$, we have that $q^l_{kij}(0) = 0$, for $1 \le l, k, j \le N$, $1 \le i \le M$; afterwards, the $q^l_{kij}$ variables are updated at the end of each block (except for the last one of the training sequence) using

$$q^l_{kij}(t_0 + h) = \sum_{n=1}^{N} \gamma_{ln}(t_0) q^n_{kij}(t_0) + \sum_{t=t_0+1}^{t_0+h} \gamma_{lk}(t) x_i(t) y_j(t-1) \tag{4.52}$$

where $\gamma_{lk}(t)$ can be computed for all $l, k \in [1, N]$ and $t \in [t_0, t_0 + h]$ as

$$\gamma_{lk}(t) = \begin{cases} \delta_{lk} & \text{if } t = t_0 + h \\ g'[\sigma_k(t)] \sum_{n=1}^{N} \gamma_{ln}(t+1) \left(\sum_{i=1}^{M} w_{nik} x_i(t+1)\right) & \text{if } t_0 \le t < t_0 + h. \end{cases} \tag{4.53}$$

For the particular case of $t_0 = 0$, we have $\gamma_{lk}(0) = 0$ and $\delta_k(0) = 0$, for $1 \le l, k \le N$.

Provided that the block size $h$ is of order $O(N)$ (e.g. $h = N$), the second-order version of Schmidhuber's algorithm performs an *average* of $O(N^3 M)$ computations per time step and requires fixed-size space of the order $O(N^3 M)$, which is due, respectively, to the calculation and storage of the $q^l_{kij}(t_0)$ values. When $t_f \le h$, the method is equivalent to the second-order version of BPTT, requiring only $O(N^2 M)$ computations per time step.

Now, let us turn our attention to the *pseudo-gradient* learning algorithm reported by Zeng *et al.* for training second-order SLRNNs with a discrete activation function [ZeGS:93]. With the aim of allowing the network to form stable states during training, Zeng *et al.* suggested to add a discretization step for computing the activation values of the units, this is

$$h_k(t) = g\ \left( \sum_{i=1}^{M} \sum_{j=1}^{N} w_{kij}\ x_i(t)\ y_j(t-1) \right) \quad \text{for } 1 \le k \le N, \tag{4.54}$$

$$y_k(t) = D\ (\ h_k(t)\ ) \quad \text{for } 1 \le k \le N, \tag{4.55}$$

where the sigmoid function $g_s$ given by (4.5) was taken as $g$, and

$$D(h) = \begin{cases} 0.8 & \text{if } h \ge 0.5 \\ 0.2 & \text{if } h < 0.5. \end{cases} \tag{4.56}$$

In addition, Zeng *et al.* assumed that the input vector $\mathbf{I}^t$ does always belong to an orthonormal basis, with just one input on, $x_{i(t)}(t) = 1$, and the rest off, $x_m(t) = 0$ for $m \neq i(t)$, which corresponds to a local representation of a finite set of input symbols [ZeGS:93]. In this way, the second-order SLRNN can be seen as $M$ first-order SLRNNs sharing the units, such that at each time step $t$, the input vector $\mathbf{I}^t$ acts like a switching control to enable just one of the nets and disable the others. Hence, their second-order SLRNN dynamics can be rewritten as

$$y_k(t) = g_d \left( \sum_{j=1}^{N} w_{ki(t)j}\, y_j(t-1) \right) \quad \text{for } 1 \leq k \leq N, \tag{4.57}$$

where

$$g_d(\sigma) = \begin{cases} 0.8 & \text{if } \sigma \geq 0.0 \\ 0.2 & \text{if } \sigma < 0.0. \end{cases} \tag{4.58}$$

In operational mode, the network is equivalent to a network with a hard-limiting activation function only (as shown above), but during training, however, both the derivative of the soft sigmoid function $g_s$ and the analog activation values $h_k(t)$ are made use of in a *pseudo-gradient* learning algorithm for updating the weights. Thus, the weights of this type of second-order SLRNN can be adjusted during training according to

$$\triangle w_{kij} = -\alpha \frac{\tilde{\partial} E_{total}(1,t_f)}{\partial w_{kij}} \quad \text{for } 1 \leq k,j \leq N,\, 1 \leq i \leq M, \tag{4.59}$$

where $\tilde{\partial}/\partial w_{kij}$ denotes the "pseudo-gradient" with respect to a weight $w_{kij}$. An approximation of the second-order version of RTRL was proposed to compute the pseudo-gradient $\tilde{\partial} E_{total}(1,t_f)/\partial w_{kij}$ for all the weights [ZeGS:93]. Indeed, Zeng *et al.* described an algorithm for the particular case in which there is an error value only at the end of the sequence and only for one unit (string classification task), i.e. $E_{total}(1,t_f) = E(t_f) = \frac{1}{2}(d_0(t_f) - h_0(t_f))^2$, but it is straightforward to define a similar learning algorithm for the general case, when there may be errors for various units at several time steps, as follows:

$$\frac{\tilde{\partial} E_{total}(1,t_f)}{\partial w_{kij}} = \sum_{t=1}^{t_f} \frac{\tilde{\partial} E(t)}{\partial w_{kij}} = -\sum_{t=1}^{t_f} \sum_{l=1}^{N} (d_l(t) - h_l(t)) \frac{\tilde{\partial} h_l(t)}{\partial w_{kij}} \tag{4.60}$$

where the pseudo-gradients $\tilde{p}^l_{kij}(t) = \tilde{\partial} h_l(t)/\partial w_{kij}$, $1 \leq l,k,j \leq N$, $1 \leq i \leq M$, can be computed forward in time at each time step as

$$\tilde{p}^l_{kij}(t) = g'[\sigma_l(t)] \left[ \delta_{lk}\delta_{i(t)i} y_j(t-1) + \sum_{n=1}^{N} w_{li(t)n} \tilde{p}^n_{kij}(t-1) \right] \tag{4.61}$$

from the initial values

$$\tilde{p}^l_{kij}(0) = 0. \tag{4.62}$$

In fact, the preceding learning algorithm could be further generalized by removing the restriction of the input local encoding. Moreover, the computation of the pseudo-gradient $\tilde{\partial} E_{total}(1, t_f)/\partial w_{kij}$ may be performed using an approximation of the second-order version of BPTT or Schmidhuber's algorithm instead of using the RTRL-like approach described.

In any case, it is clear that, in carrying out the chain rule for the gradient, the real gradient $\partial y_l(t)/\partial w_{kij}$, which is zero almost everywhere, is replaced by the pseudo-gradient $\tilde{\partial} h_l(t)/\partial w_{kij}$. Zeng *et al.* justified the use of the pseudo-gradient in the following manner [ZeGS:93]: Suppose the net-input of a unit is $\sigma_0 > 0$, so the activation value stands on the upper side of the hard threshold function $g_d(\sigma)$, and we wish to move it downhill. The derivative of $g_d(\sigma)$ does not provide any information, since it is zero at $\sigma_0$. On the other hand, the derivative of $g_s(\sigma)$ is positive at $\sigma_0$ and increases as $\sigma_0 \rightarrow 0$; so it indicates that the downhill direction is to decrease $\sigma_0$, which is also the case in $g_d(\sigma)$, and the magnitude of $g_s'(\sigma_0)$ indicates how close we are to a step down in $g_d(\sigma)$. Therefore, the gradient $g_s'(\sigma_0)$ can be used as a heuristic hint as to which direction and how close a step down is. This heuristic hint is what supports the use of the pseudo-gradient in the learning algorithm.

To end this subsection, I will mention briefly some other learning algorithms that have been used for training second-order SLRNNs. Pollack [Poll:91] used a truncated gradient-descent algorithm to train a second-order SLRNN, given by (4.37), such that the backward computation (through time) of the gradient only arrived at the penultimate time step, thus reducing the computational cost. However, the results reported for a string classification task using this truncated gradient learning algorithm were very poor [Poll:91]. Watrous and Kuhn [WaKu:92] trained their second-order SLRNN, with dynamics given by Eq.(4.39), using a BPTT-like computation of the true gradient and the BFGS algorithm for gradient-descent optimization [Luen:84]. Finally, Forcada and Carrasco proposed an extension of the second-order RTRL algorithm [FoCa:95], in which the gradient of the total error with respect to the activation values of the units at $t = 0$, $\partial E_{total}/\partial y_k(0)$ for $1 \leq k \leq N$, was also computed in order to learn the initial state (in addition to the weights).

### 4.1.2 Augmented single-layer recurrent neural networks (ASLRNNs)

The concept of *augmented* single-layer recurrent neural network (ASLRNN) was introduced by Goudreau and Giles in the context of studying the abilities and limitations of SLRNNs for representing finite state machines (FSMs) [GoGi:93, GoGC:94]. An ASLRNN is formed by adding one or more layers of feed-forward neurons to an SLRNN, with the output units in the highest layer. In this way, the single-layer of recurrent units is concerned with the internal state representation and the state transitions of an FSM, whereas the output function of the states is associated with the feed-forward layer(s).

Goudreau [GoGi:93] pointed that an ASLRNN including just one additional feed-forward layer is approximately equivalent, in terms of representational abilities, to an SLRNN with non-recurrent output units in which an extra time step is allowed before the output is read. This approach is closely related to the use of an *end-of-string symbol* for the task of sequence classification, where the output is only required at the end of the sequence, and therefore, the end symbol gives the SLRNN an extra time step to process the solution.

In the sequel, we will only deal with the former (and more general) interpretation of augmented SLRNNs. This is, we define an ASLRNN as a general layered architecture, such that the first-layer of (non-input) units is a fully-connected discrete-time RNN model, either a first-order or a second-order SLRNN, and the rest of layers contain feed-forward neurons that receive (first-order) connections from all the units in the previous layer. Hence, the activation values of the recurrent units are also fed into the feed-forward neurons of the second layer, and the neurons of the last layer constitute the output units of the network. It is assumed that the activation values of the output units are computed before new external inputs are introduced and the activation values of the recurrent units are fed back; i.e. the time discretization of the network operation is carried out by putting the time step delay after the computation of the network outputs.

Let $L$ be the total number of layers in an ASLRNN, $L \geq 2$, where for $l \in [1, L]$, $N_l$ denotes the number of units in layer $l$. For compatibility with previous notation, let $N = N_1$ be the number of recurrent units and let $P = N_L$ be the number of output units. Again, assume that the network has $M$ external inputs that are fed into the recurrent layer, which are labelled $x_1, ..., x_M$. The net-input and activation values of the units $U_1^l, ..., U_{N_l}^l$ in layer $l$ are labelled $\sigma_1^l, ..., \sigma_{N_l}^l$ and $y_1^l, ..., y_{N_l}^l$, respectively. Let $\mathbf{S}^t = [y_1^1(t-1), ..., y_N^1(t-1)]^T$ and $\mathbf{I}^t = [x_1(t), ..., x_M(t)]^T$ represent the current state and input vectors, respectively, that contain the data supplied to the recurrent layer

at time step $t$. Activations are forward propagated throughout the network within a single time step, and consequently, $\mathbf{O}^t = [y_1^L(t), ..., y_P^L(t)]^T$ represents the output vector that is obtained in response to the pair $(\mathbf{I}^t, \mathbf{S}^t)$.

The equations that describe the dynamic behavior of an $L$-layer ASLRNN are

$$y_k^1(t) \;=\; g_1\;(\;f\;(\mathbf{W}_k^1, \mathbf{I}^t, \mathbf{S}^t)) \quad \text{for } 1 \leq k \leq N, \tag{4.63}$$

for the recurrent layer, and

$$y_k^l(t) \;=\; g_l\;(w_{k0}^l + \sum_{j=1}^{N_{l-1}} w_{kj}^l y_j^{l-1}(t)\;) \quad \text{for } 2 \leq l \leq L,\; 1 \leq k \leq N_l, \tag{4.64}$$

for the higher feed-forward layers, where

- $\mathbf{W}_k^1$ denotes the vector of weights of the recurrent unit $U_k^1$,
- $w_{k0}^l$ is the bias weight of the feed-forward neuron $U_k^l$,
- $w_{kj}^l$ is the weight corresponding to the incoming connection of unit $U_k^l$ from unit $U_j^{l-1}$,
- a possibly different activation function $g_l$ is allowed for each layer $l$, but all the units in the same layer use the same activation function, and finally,
- the aggregation function of the recurrent units is given by

$$f\;(\mathbf{W}_k^1, \mathbf{I}^t, \mathbf{S}^t) \;=\; \sum_{i=1}^{M} w_{ki}^1 x_i(t) \;+\; \sum_{j=1}^{N} w_{k(M+j)}^1 y_j^1(t-1)\;\;, \tag{4.65}$$

in a *first-order* ASLRNN, and it is given by

$$f\;(\mathbf{W}_k^1, \mathbf{I}^t, \mathbf{S}^t) \;=\; \sum_{i=1}^{M} \sum_{j=1}^{N} w_{kij}^1\; x_i(t)\; y_j^1(t-1)\;\;. \tag{4.66}$$

in a *second-order* ASLRNN.

For the supervised training of ASLRNNs, we can use a gradient-descent learning method that applies common *backpropagation* [RuHW:86] to the feed-forward layers and either first-order (second-order) BPTT, RTRL, or Schmidhuber's algorithm, to the first-order (second-order) recurrent layer, where the error signals for the recurrent units are back-propagated from the output layer.

Again, we consider that the network starts running at time step $t = 1$, using an arbitrary initial state vector $\mathbf{S}^1 = [y_1^1(0), ..., y_N^1(0)]^T$, and a training sequence of $t_f$ time steps is provided, where $T(t)$ denotes the set of indices of output units $U_k^L$ for which

there exists a specified target value $d_k(t)$ at time $t$. The error of an output unit $U_k^L$ at time step $t$ is defined as

$$err_k(t) = \begin{cases} d_k(t) - y_k^L(t) & \text{if } k \in T(t) \\ 0 & \text{otherwise;} \end{cases} \tag{4.67}$$

the total squared error of the network at time step $t$ is

$$E(t) = \frac{1}{2} \sum_{k=1}^{P} [err_k(t)]^2, \tag{4.68}$$

and the total squared error over the whole training sequence $E_{total}(1, t_f)$ is just the sum of the errors $E(t)$, for $1 \leq t \leq t_f$. The weights of the ASLRNN are adjusted along the negative of $\nabla_{\mathbf{W}} E_{total}(1, t_f)$ in a pure gradient-descent scheme, but a momentum term might be added optionally to smooth the weight trajectory.

In order to calculate the gradient of the total error for each weight, we need to define

$$\delta_i^l(t) = -\frac{\partial E_{total}(1, t_f)}{\partial \sigma_i^l(t)} = -\sum_{\tau=t}^{t_f} \frac{\partial E(\tau)}{\partial \sigma_i^l(t)} \quad \text{for } 1 \leq l \leq L,\ 1 \leq i \leq N_l. \tag{4.69}$$

It is clear that the net-input of the feed-forward neurons at time step $t$ only affects the error of the network at time step $t$; therefore,

$$\delta_i^l(t) = -\frac{\partial E(t)}{\partial \sigma_i^l(t)} \quad \text{for } 2 \leq l \leq L,\ 1 \leq i \leq N_l, \tag{4.70}$$

and these derivatives can be computed by common backpropagation as follows:

$$\delta_i^l(t) = \begin{cases} g_l'[\sigma_i^l(t)]\ err_i(t) & \text{if } l = L, \\ g_l'[\sigma_i^l(t)] \sum_{k=1}^{N_{l+1}} w_{ki}^{l+1} \delta_k^{l+1}(t) & \text{if } 2 \leq l < L. \end{cases} \tag{4.71}$$

Hence, the weights of the feed-forward neurons (for layers $l \in [2, L]$) are updated according to

$$\Delta w_{ij}^l = -\alpha \frac{\partial E_{total}(1, t_f)}{\partial w_{ij}^l} = \sum_{t=1}^{t_f} \Delta w_{ij}^l(t) \tag{4.72}$$

where

$$\Delta w_{ij}^l(t) = -\alpha \frac{\partial E(t)}{\partial w_{ij}^l} = \begin{cases} \alpha\ \delta_i^l(t) & \text{if } j = 0, \\ \alpha\ \delta_i^l(t) y_j^{l-1}(t) & \text{if } j > 0. \end{cases} \tag{4.73}$$

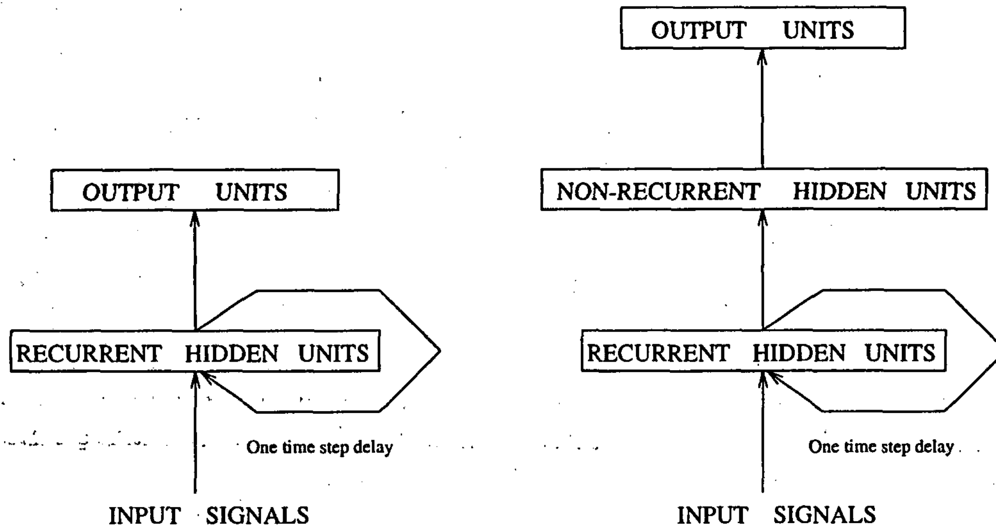


**Fig. 4.2** *Two-layer ASLRNN (left) and three-layer ASLRNN (right) architectures.*

Now, let

$$e_i(t) = -\frac{\partial E(t)}{\partial y_i^1(t)} = \sum_{k=1}^{N_2} w_{ki}^{(2)} \delta_k^{(2)}(t) \quad \text{for } 1 \leq i \leq N. \tag{4.74}$$

As shown above, these $e_i(t)$ values are obtained by back-propagating the error $E(t)$ to the units in the first-layer.

The weights of the recurrent neurons (for layer $l = 1$) can be updated from the information given by Eq.(4.74) using the proper version (either first- or second-order, depending on the type of ASLRNN) of either BPTT, RTRL, or Schmidhuber's algorithm. To this end, just substitute the $e_i(t)$ values given by (4.74) for the $e_i(t)$ or $e_k(t)$ errors appearing in Eqs. (4.16) and (4.48) for BPTT, (4.22) and (4.43) for RTRL, or (4.27) and (4.51) for Schmidhuber's algorithm. In particular, the first-layer derivatives $\delta_i^{(1)}(t)$, $1 \leq i \leq N$, defined in Eq.(4.69), can be computed for $1 \leq t \leq t_f$ using Eq.(4.16) or (4.48), for first-order or second-order ASLRNNs, respectively.

Since we have simply applied the chain rule, it can be shown that the resulting learning algorithms for ASLRNNs correctly compute the true gradient of the total error with respect to the weights of the recurrent layer. In any case, all the weights of the network should be updated at the same time, e.g. at the end of the training sequence (using any of the three algorithms), or after a block of $h$ time steps (using Schmidhuber's or RTRL), or at each time step (only with RTRL).

It is well-known that at most 2 hidden layers are enough for a feed-forward network to approximate any function and just one hidden layer is enough to approximate any continuous function [HeKP:91]. Consequently, the number $L$ of layers in an ASLRNN is typically 2 or 3, giving rise to the general two-layer and three-layer ASLRNN models sketched in Fig. 4.2, where the recurrent hidden units may have first- or second-order connections.

The *Simple Recurrent Network* (SRN) architecture proposed by Elman [Elman:90] and used by several researchers [ClSM:89, DaDa:91, SeCM:91], which is displayed in Fig. 4.3, was conceived as a modified feed-forward network with one hidden layer (trained by common backpropagation) such that the activation pattern of the hidden units is copied onto a set of "context" units, which feed into the hidden layer along with the input units. Actually, an SRN is simply a two-layer first-order ASLRNN that is trained using the backpropagation algorithm to update all the weights of the network. This implies, of course, that a severely truncated gradient is computed for the weights of the recurrent hidden units, and therefore, the learning performance is normally impaired with respect to the case of using a true gradient-descent learning method for the same network, such as the combined learning methods for ASLRNNs aforementioned.

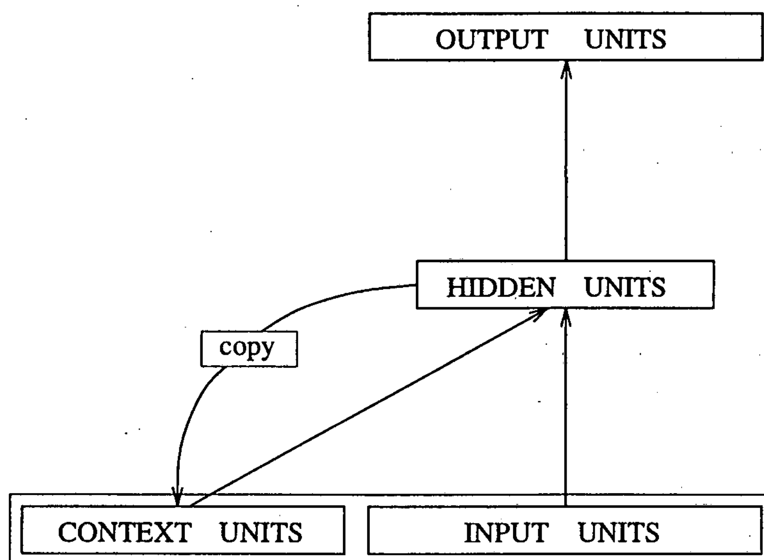


**Fig. 4.3** *The Simple Recurrent Network (SRN) architecture proposed by Elman.*

### 4.1.3 Recurrent Cascade-Correlation (RCC) architecture

The Recurrent Cascade-Correlation (RCC) [Fahl:91a] is a recurrent version of the Cascade-Correlation learning architecture of Fahlman and Lebiere [FaLe:90]. As the SLRNNs and the ASLRNNs described previously, RCC can learn from examples to map a sequence of inputs into a desired sequence of outputs. However, in this case, the network is constructed incrementally: new hidden units with recurrent connections are added to the network one at a time, as they are needed during training.

The original Cascade-Correlation is a supervised learning architecture that builds a near-minimal multi-layer network topology in the course of training [FaLe:90]. Initially, the network contains only inputs, output units, and the connections between them. The single layer of connections is trained using the Quickprop algorithm [Fahl:88] (a variation on back-propagation) to minimize the error on the training set. When the level of error stops decreasing, the performance of the network is evaluated. If the performance is good enough, the learning ends. Otherwise, a new hidden unit is added to the network in an attempt to reduce the residual error.

Before adding a new hidden unit in the Cascade-Correlation approach, a pool of *candidate units* is tested, such that each of these units receives weighted connections from the network's inputs and from any hidden units already present in the net, but their output activations are not yet connected to the output units. Then, the weights of each candidate unit are adjusted to maximize the correlation between the activation value of the unit and the residual error yielded by running the current network over the patterns in the training set. When the correlation scores stop improving, the candidate unit with the best correlation is selected as the new hidden unit, the weights associated with its incoming connections are frozen, and all the weights of the output units are re-trained, including those from the new hidden unit. The process of adding a new hidden unit and re-training the output layer is repeated until the error is small enough. Note that each new hidden unit effectively adds a new layer to the net, since it receives connections from all the old hidden units.

The main advantage of Cascade-Correlation is that a reasonably small (near-minimal) network is built automatically, thus eliminating the need for the user to determine in advance the number of layers and hidden units in a feed-forward network. In addition, learning is fast, as only a single layer of weights is being trained at any given time, and a good generalization can be expected, due to the small size of the constructed network.

The Recurrent Cascade Correlation (RCC) architecture [Fahl:91a] adds recurrent operation to the original Cascade-Correlation. To this end, each of the hidden and

candidate units is provided with a single weighted self-recurrent link that feeds back its own activation value on the previous time step.

Let $M$ be the number of network inputs and let $P$ be the number of output units in the network. At a certain stage during the training process $N$ hidden units will have been added. Let $x_j(t)$ denote the network's $j$-th input at time step $t$, and let $y_i(t)$ and $o_k(t)$ denote the activation value of the $i$-th hidden unit and the $k$-th output unit, respectively, at time step $t$. The equations that describe the dynamic behavior of the active network (without candidate units) at the current training stage are:

$$y_i(t) = g\ (\ w_{i0}^h y_i(t-1) + \sum_{j=1}^{M+i-1} w_{ij}^h z_j(t)\ ) \quad \text{for } 1 \leq i \leq N, \tag{4.75}$$

$$o_k(t) = g\ (\ \sum_{j=1}^{M+N} w_{kj}^o z_j(t)\ ) \quad \text{for } 1 \leq k \leq P, \tag{4.76}$$

where $g$ is the activation function (Fahlman proposed the sigmoid-like hyperbolic tangent), $w_{i0}^h$ refers to the weight of the self-recurrent link of the $i$-th hidden unit, and the inputs $z_j(t)$ of the incoming non-recurrent connections are given by

$$z_j(t) = \begin{cases} x_j(t) & \text{if } j \leq M \\ y_{j-M}(t) & \text{if } j > M. \end{cases} \tag{4.77}$$

For the process of selecting a new hidden unit, some number $C$ of candidate units are incorporated, with dynamics given by

$$v_i(t) = g\ (\ w_{i0}^c v_i(t-1) + \sum_{j=1}^{M+N} w_{ij}^c z_j(t)) \quad \text{for } 1 \leq i \leq C, \tag{4.78}$$

where $v_i(t)$ denotes the activation value of the $i$-th candidate unit at time step $t$, and $w_{i0}^c$ refers to the weight of its self-recurrent link. This weight is trained together with the rest of weights of the candidate unit to maximize the correlation of the unit's activation value with the residual error. More precisely, the goal of the adjustment of the $i$-th candidate weights is to maximize the value of

$$R_i = \sum_{k=1}^{P} |r_{ik}| = \sum_{k=1}^{P} \left| \sum_{s=1}^{|S|} \sum_{t=1}^{t_f(s)} (v_i(t) - \overline{v_i})(e_k(t) - \overline{e_k}) \right| \tag{4.79}$$

where $S$ is the training set of sequences, $e_k(t)$ denotes the error of the $k$-th output unit at time $t$ of a given sequence $s$, and the quantities $\overline{v_i}$ and $\overline{e_k}$ are the values of $v_i(t)$ and $e_k(t)$ averaged over all the time steps of the sequences in the training set.

In order to maximize $R_i$, the partial derivative of $R_i$ with respect to each of the candidate unit's weights $w_{ij}^c$ must be computed:

$$\frac{\partial R_i}{\partial w_{ij}^c} = \sum_{k=1}^{P} \text{sign}(r_{ik}) \sum_{s=1}^{|S|} \sum_{t=1}^{t_f(s)} (e_k(t) - \overline{e_k}) \frac{\partial v_i(t)}{\partial w_{ij}^c}, \tag{4.80}$$

where the derivatives $\partial v_i(t) / \partial w_{ij}^c$ are given by

$$\frac{\partial v_i(t)}{\partial w_{ij}^c} = \begin{cases} g'(\sigma_i^c(t)) \left( v_i(t-1) + w_{i0}^c \frac{\partial v_i(t-1)}{\partial w_{i0}^c} \right) & \text{if } j = 0 \\ g'(\sigma_i^c(t)) \left( z_j(t) + w_{i0}^c \frac{\partial v_i(t-1)}{\partial w_{ij}^c} \right) & \text{if } j \in [1, M+N]. \end{cases} \tag{4.81}$$

It is assumed that the candidate activation value and the derivatives are all zero at $t = 0$. Hence, the derivatives $\partial v_i(t-1)/\partial w_{ij}^c$ are always known from the previous time step, and it is only required to store these values (one for each weight), plus the previous activation $v_i(t-1)$, to compute the derivatives at the current step $t$.

When the correlations $R_i$ $(1 \leq i \leq C)$ do not improve with further training, the candidate unit associated with the highest correlation $R_i$ is selected as the new hidden unit (with index $N+1$), its weights $w_{(N+1)j}^h = w_{ij}^c$ $(0 \leq j \leq M+N)$ are frozen, and new connections are established from the new unit to each of the output units. Then, the weights $w_{kj}^o$ $(1 \leq k \leq P, 1 \leq j \leq M+N+1)$ of the output units are re-adjusted to minimize the total error on the training set. Several such cycles, one for each hidden unit, are performed until the residual error is considered acceptable or stabilizes.

The RCC architecture described above retains the advantages of the original feed-forward version (automatic choice of network topology, fast learning, good generalization, ability to create complex high-order feature detectors through the cascade of hidden units), while it permits to learn tasks involving sequences.

### 4.1.4 The DOLCE architecture

If a recurrent neural network (RNN) is wanted to learn to emulate a finite-state machine (FSM), as in the case of regular grammatical inference, the fact that the state dynamics in the continuous state space of the network, which is due to a continuous activation function in the hidden units, does not match well to the discrete behavior of an FSM should be taken into account. We will see in a later section that a DFA can be extracted from the dynamics of the internal states of a trained RNN by using clustering techniques a posteriori. An alternative is to use a discrete activation function, as in the work by

Zeng *et al.* [ZeGS:93] commented earlier, but then, a heuristic pseudo-gradient learning algorithm must be used to train the RNN.

Das and Mozer [DaMo:94] have proposed a different alternative, that consists of integrating an adaptive clustering module within the network, whose parameters are adjusted during training, along with the network weights, using a true gradient-descent scheme. The DOLCE architecture proposed by Das and Mozer, which is sketched in Fig. 4.4a, allows discrete states to evolve in a net as learning progresses, where the clustering module is used to quantize the state space dynamically. The DOLCE architecture is based on the assumption that a finite set of discrete internal states is required for the task, and that the actual network state belongs to this set but has been corrupted by noise. In this setting, DOLCE learns to recover the discrete state with maximum a posteriori probability from the noisy state.

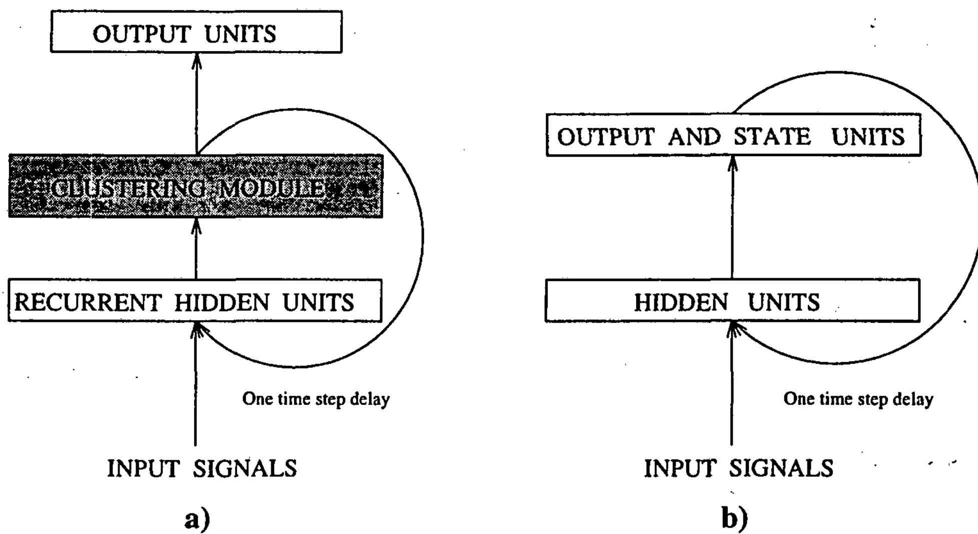


**Fig. 4.4** a) *The DOLCE architecture proposed by Das and Mozer.*
b) *The first-order 2LRNN proposed by Manolios and Fanelli.*

The DOLCE architecture is similar to a two-layer second-order ASLRNN, with the difference that the hidden layer activities are processed by the clustering module before being propagated. The dynamic behavior of the network can be specified as

$$h_k(t) = g_1 \left( \sum_{i=1}^{M} \sum_{j=1}^{N} w^h_{kij} \, x_i(t) \, y_j(t-1) \right) \quad \text{for } 1 \le k \le N, \tag{4.82}$$

$$y_k(t) = D_k \left( \mathbf{h}^t, \text{ clustering_parameters} \right) \quad \text{for } 1 \le k \le N, \tag{4.83}$$

$$o_i(t) = g_2 \left( w^o_{i0} + \sum_{j=1}^{N} w^o_{ij} y_j(t) \right) \quad \text{for } 1 \le i \le P, \tag{4.84}$$

where $g_1, g_2$ are differentiable activation functions for the hidden and output layer, respectively, and for every time step $t$, $\mathbf{I}^t = [x_1(t), ..., x_M(t)]^T$ is the input vector, $\mathbf{h}^t = [h_1(t), ..., h_N(t)]^T$ is the hidden state vector (i.e. the activation pattern observed in the hidden units at time step $t$), $\mathbf{S}^t = [y_1(t), ..., y_N(t)]^T$ is the state vector obtained after clustering, and $\mathbf{O}^t = [o_1(t), ..., o_P(t)]^T$ is the output vector of the network; finally, $D_k$ is a differentiable function that computes the $k$-th component of the state vector from the hidden pattern, depending on some clustering parameters.

At first, the objective of the clustering module is to map regions in state space to a single point in the same space, thus partitioning the state space in clusters, where each cluster corresponds to a discrete internal state. Moreover, the clusters must be adaptive and dynamic, changing over the course of learning. To be used in DOLCE, a clustering algorithm must incorporate a pressure to produce a small number of clusters, and it must allow for a soft or continuous clustering during training, in order to be integrated into a gradient-based learning procedure. Das and Mozer proposed the use of a Gaussian mixture model for the clustering module, where the mixture model parameters are adjusted to minimize the performance error of the network.

The assumptions underlying the use of a Gaussian mixture model are the following [DaMo:94]:

1) a finite set of $n$ *true* internal states $\mathbf{C} = \{\mathbf{c}_1, ..., \mathbf{c}_n\}$ is required, where each $\mathbf{c}_i$ is a vector of $N$ activation values;
2) any observed hidden activation pattern $\mathbf{h}$ belongs to $\mathbf{C}$ but has been corrupted by noise due to inaccuracy in the network weights;
3) the noise is Gaussian and decreases as the weights are adjusted to better perform the task.

A Gaussian mixture distribution that models the observed hidden states is

$$p(\mathbf{h}|\mathbf{C}), \sigma, \mathbf{q}) = \sum_{i=1}^{n} \frac{q_i}{(2\pi\sigma_i^2)^{N/2}} e^{-|\mathbf{h}-\mathbf{c}_i|^2/2\sigma_i^2} \tag{4.85}$$

where $\sigma_i^2$ is the variance of the noise that corrupts state $\mathbf{c}_i$, $q_i$ is the prior probability that the true state is $\mathbf{c}_i$, and $N$ is the dimensionality of the state space (i.e. the number of hidden units). The parameters of the mixture distribution ($n, \mathbf{C}, \sigma$, and $\mathbf{q}$) are all known at every time step, but they are changed during training, except the number of true states $n$, that is only modified indirectly, as will be explained.

Given a noisy observed hidden state $\mathbf{h}$, the maximum a posteriori (MAP) estimator of $\mathbf{h}$ in $\mathbf{C}$, denoted $\hat{\mathbf{h}}$, should be determined to replace the noisy state in all subsequent

computation. The probability of an observed state $\mathbf{h}$ being generated by a given true state $\mathbf{c_i}$ is

$$p(\mathbf{h}|\mathbf{c_i}) = (2\pi\sigma_i^2)^{-N/2}\, e^{-|\mathbf{h}-\mathbf{c_i}|^2/2\sigma_i^2}, \tag{4.86}$$

and, using Bayes' theorem, the posterior probability of true state $\mathbf{c_i}$ given $\mathbf{h}$ is

$$p(\mathbf{c_i}|\mathbf{h}) = \frac{q_i\, p(\mathbf{h}|\mathbf{c_i})}{\sum_{j=1}^{n} q_j\, p(\mathbf{h}|\mathbf{c_j})}. \tag{4.87}$$

Therefore, the MAP estimator is $\hat{\mathbf{h}} = \mathbf{c_i}$, where $\mathbf{c_i}$ is the true state that maximizes $p(\mathbf{c_j}|\mathbf{h})$, for $1 \leq j \leq n$. However, because a differentiable transformation is required for gradient-descent learning, a "soft" version of MAP, given by

$$\bar{\mathbf{h}} = \sum_{i=1}^{n} \mathbf{c_i}\, p(\mathbf{c_i}|\mathbf{h}) \tag{4.88}$$

was proposed and used by Das and Mozer instead of $\hat{\mathbf{h}}$.

Now, we can write the soft transformation performed by the clustering module at time step $t$ as

$$y_k(t) = D_k\,(\,\mathbf{h}^t, \mathbf{C}, \sigma, \mathbf{q}\,) = \sum_{i=1}^{n} c_{ik}\, \frac{q_i\,(2\pi\sigma_i^2)^{-N/2}\, e^{-|\mathbf{h}^t-\mathbf{c_i}|^2/2\sigma_i^2}}{\sum_{j=1}^{n} q_j\,(2\pi\sigma_j^2)^{-N/2}\, e^{-|\mathbf{h}^t-\mathbf{c_j}|^2/2\sigma_j^2}} \quad \text{for } 1 \leq k \leq N. \tag{4.89}$$

During training, the weights of the network can be updated along the negative of the gradient of the total error, using a gradient-descent method based on one of the combined algorithms mentioned in Section 4.1.2, such that the derivatives of the above transformation are included in the application of the chain rule. For instance, if RTRL is the underlying algorithm for training the weights of the recurrent layer, then the required partial derivatives $p_{kij}^l(t) = \partial y_l(t)/\partial w_{kij},\ 1 \leq l,k,j \leq N,\ 1 \leq i \leq M$, can be decomposed as

$$p_{kij}^l(t) = \sum_{r=1}^{N} \frac{\partial y_l(t)}{\partial h_r(t)}\, \frac{\partial h_r(t)}{\partial w_{kij}} \tag{4.90}$$

and computed forward in time using

$$p_{kij}^l(t) = \sum_{r=1}^{N} \frac{\partial y_l(t)}{\partial h_r(t)}\, g_1'[\sigma_r(t)] \left[\delta_{rk} x_i(t) y_j(t-1) + \sum_{m=1}^{M}\sum_{a=1}^{N} w_{rma} x_m(t)\, p_{kij}^a(t-1)\right] \tag{4.91}$$

instead of Eq.(4.44), where $\partial y_l(t)/\partial h_r(t),\ 1 \leq l, r \leq N$, can be calculated by derivating the system of equations (4.89) with respect to each component $h_r(t)$ of the hidden state vector $\mathbf{h}^t$.

Concerning the learning of the clustering parameters, the following approach was proposed by Das and Mozer [DaMo:94]. The number of clusters or Gaussian bumps $n$ is initially set to a large value (since $n$ should not be less than the number of states in the target FSM), and the training procedure includes a technique for eliminating implicitly the unnecessary true states in $\mathbf{C}$. At the start of training, each Gaussian center $\mathbf{c_i}$ is initialized to a random location in the hidden state space, the standard deviation of each Gaussian $\sigma_i$ is set to a large value, and the prior probabilities are set uniformly to $q_i = 1/n$, for $1 \leq i \leq n$. The mixture model parameters $\mathbf{C}, \sigma$, and $\mathbf{q}$ are adjusted by gradient descent in a global cost measure $\mathcal{C}$ defined as

$$\mathcal{C} = E_{total}(1, t_f) - \lambda \sum_{i=1}^{n} q_i \ln q_i \tag{4.92}$$

where $E_{total}(1, t_f)$ is the total squared error related to the performance of the network over a training sequence, which is measured at the output units, and the second term is a complexity cost, the entropy of the prior distribution $\mathbf{q}$, where $\lambda$ is a regularization parameter. The complexity cost is minimal when only one Gaussian has a nonzero prior probability, and maximal when all the prior probabilities are equal. Hence, the second term of the cost measure encourages unnecessary Gaussians to drop out of the mixture model.

Das and Mozer used an extension of BPTT to calculate the gradient of $\mathcal{C}$ with respect to the model parameters $\mathbf{C}, \sigma$, and $\mathbf{q}$. Actually, they proposed an optimization procedure that is performed not over $\sigma$ and $\mathbf{q}$ directly but rather over hyper-parameters $\mathbf{a}$ and $\mathbf{b}$, where $\sigma_i^2 = \exp(a_i)/\beta$ and $q_i = \exp(-b_i^2)/\sum_{j=1}^{n} \exp(-b_j^2)$, for $1 \leq i \leq n$. The temperature parameter $\beta$ scales the overall spread of the Gaussians, which corresponds to the level of noise in the model. Since the level of noise should decrease as performance on the training set improves, $\beta \propto 1/E_{total}$ is recommended. In this way, if $E_{total} \rightarrow 0$, then $\beta \rightarrow \infty$ and the probability density under one Gaussian at $\mathbf{h}$ will become infinitely greater than the density under any other; hence, in such a case, the soft MAP estimator $\overline{\mathbf{h}}$ becomes equivalent to the MAP estimator $\hat{\mathbf{h}}$, and a set of discrete states is obtained.

### 4.1.5 First-order 2-layer RNNs

To end this section, the first-order 2-layer RNN architecture proposed by Manolios and Fanelli [MaFa:94], which is sketched in Fig. 4.4b, is briefly discussed[3]. Motivated by the fact that feed-forward networks with one hidden layer are universal approximators [HoSW:89], Manolios and Fanelli argued that their architecture is the simplest first-order network with two recurrent layers that is a universal approximator for discrete-time, time-invariant dynamic systems [MaFa:94]. Such a system can be described by

$$\begin{aligned} \mathbf{s}(t) &= \Phi\left[\mathbf{s}(t-1), \mathbf{x}(t)\right] \\ \mathbf{o}(t) &= \Omega\left[\mathbf{s}(t-1), \mathbf{x}(t)\right] \end{aligned}$$

where $\mathbf{s}(t) \in \Re^n$ is the state of the system at time $t$, $\mathbf{x}(t) \in \Re^m$ is the input of the system at time $t$, $\mathbf{o}(t) \in \Re^p$ is the output of the system at time $t$, $t$ is a non-negative integer, $\Phi$ is the state transition function and $\Omega$ is the output function of the system. Note that FSMs can be seen as a particular case of the above systems such that contain a finite number of states, inputs and outputs.

The dynamic behavior of the network in Fig. 4.4b can be specified as

$$h_i(t) = g_1\left(w_{i0}^h + \sum_{j=1}^{M} w_{ij}^h x_j(t) + \sum_{k=1}^{N} w_{i(M+k)}^h y_k(t-1)\right) \quad \text{for } 1 \leq i \leq H, \quad (4.93)$$

$$y_k(t) = g_2\left(w_{k0}^y + \sum_{i=1}^{H} w_{ki}^y h_i(t)\right) \quad \text{for } 1 \leq k \leq N, \quad (4.94)$$

where $g_1, g_2$ are differentiable activation functions with range $[0,1]$ for the hidden and output/state layer, respectively, and for every time step $t$, $\mathbf{I}^t = [x_1(t), ..., x_M(t)]^T$ is the input vector, $\mathbf{h}^t = [h_1(t), ..., h_H(t)]^T$ is the hidden activation vector, and $\mathbf{S}^t = [y_1(t), ..., y_N(t)]^T$ is the state vector. A subset of the state units corresponds to the output units, which are treated exactly as the state units, except that they are trainable units and their activations are considered the output of the network, e.g. $\mathbf{O}^t = [y_1(t), ..., y_P(t)]^T$ (with $P < N$) can be defined as the output vector.

Manolios and Fanelli proposed a gradient-descent learning technique to train the above architecture using a modification of the BPTT algorithm to compute the full gradient. They also suggested to use batch learning, i.e. to modify the weights only at the end of each training epoch, so that the order in which the training set is presented is irrelevant [MaFa:94].

[3] Do not confuse it with a first-order 2-layer ASLRNN, as displayed in Fig. 4.2.

## 4.2 Grammatical inference and recognition using recurrent neural networks

In this section, the two approaches that have been followed to use RNNs for grammatical inference and recognition are reviewed. The former consists of training a network to predict the next symbol in a gramatically valid string, and therefore, only positive examples are included in the training set. The latter consists of training a network to classify strings according to an unknown target grammar, and to this end, both positive and negative examples must be included in the training set. For both approaches, the related learning task is defined in terms of network's input/output, data representation and training scheme, together with the corresponding procedure for string recognition. Likewise, a summary of the results of the empirical studies carried out by different researchers using some of the architectures described in the preceding section is presented.

### 4.2.1 GI from positive examples: the next-symbol prediction task

Let $S^+$ be a positive sample (maybe including repeated strings) of a language $L$ over an alphabet $\Sigma$. Let $\Sigma' = \Sigma \cup \{\$\}$ be an extended alphabet including a special symbol \$ that is used as a mark of both the end and beginning of a string, an let $S^+_\$$ be the sample obtained by converting each string $s \in S^+$ into the form $\$s\$$. A recurrent neural network can be trained to predict the next input symbol (from $\Sigma'$) at each step of the presentation of any string in $S^+_\$$ (*next-symbol prediction task*). If the sample $S^+$ is "sufficiently representative"[4] of the language $L$ and the RNN is small enough to generalize the examples, the network obtained after a successful training may be used as an acceptor of the language $L$, i.e. to accept the grammatical strings and reject the ungrammatical ones. In this sense, we can say that a GI process from positive examples is carried out.

Some reported works have followed this approach for inferring RNNs (of distinct architectures) that behave as regular language acceptors [SeCM:88, ClSM:89, SmZi:89, Fahl:91b, CaCV:93]. In most of these works, an alternative extended alphabet $\Sigma'' = \Sigma \cup \{B, E\}$, $B, E \notin \Sigma$, with different special symbols for marking the beginning and end of a string, has been used, but this representation is somewhat redundant, since symbol $B$ is only used as a network input and symbol $E$ is only used as a network output (as explained next).

[4] The meaning of "sufficiently representative" here will be discussed later.

A *local encoding* is typically used to represent the symbols of the language in the RNN, with one input signal and one output unit for each symbol in $\Sigma'$. This is, $M = P = |\Sigma'|$, the presentation of an input symbol is performed by setting to 1 its associated input signal and setting to 0 the rest of input signals[5], and when the network is wanted to predict a given symbol during training, the target value for the output unit corresponding to that symbol will be 1 and the target value for the rest of output units will be 0. In addition, the activation value of the output unit associated with a given symbol, which should be in the interval $[0,1]$, is interpreted as the prediction value (or probability) for that symbol to be the next symbol in the current string. Indeed, the local encoding is only required for the output units (with $P = |\Sigma'|$) to yield the prediction values, whereas a distributed encoding (with $M$ not necessarily equal to $|\Sigma'|$) could be used for the input symbol presentation.

Hence, the process of recognizing a string $s = c_1 ... c_{|s|}$ by the network is carried out by presenting the symbols of the string $s' = \$s\$$ sequentially, one at each time step, starting with the initial \$ and ending with the actual last symbol $c_{|s|}$, and checking, at each time step, whether the activation value of the output unit associated with the next symbol in $s'$ is greater than a *predetermined threshold*. The string $s$ is accepted when all of its symbols, plus the \$ marking the end of the string, are successively predicted by the network, and it is rejected as soon as one of them is not predicted (i.e. its output unit activation does not exceed the threshold). Note that, to reject all the ungrammatical strings, the activation values of the output units corresponding to all the illegal successors at each step must be below the selected threshold.

In order to learn the next-symbol prediction task, for each string $s' = \$c_1 ... c_{|s|}\$$ in $S^+_\$$, the network is trained using the sequence of pairs of consecutive symbols in $s'$, i.e. $(\$, c_1), (c_1, c_2), ..., (c_{|s|}, \$)$, where the first element of each pair refers to the input symbol and the second one refers to the target symbol to be predicted. Each of these input/output pairs is translated into an input/output vector representation following the local encoding aforementioned. Now, it is obvious that using two special symbols $B$ and $E$, for the beginning and end of string respectively, instead of the unique special symbol \$ implies an unnecessary symbol addition. In any case, the internal state of the network must be reset at the start of each string.

The process of grammatical inference, that is carried out during network training, is implicitly based on the statistics of the presented positive sample. In fact, it has been pointed out [SeCM:91] and empirically shown [CaCV:93] that the RNN obtained after training represents a stochastic regular grammar, or stochastic DFA, where the state transition probabilities through each symbol are approximately inferred and used as prediction values for the next symbol in a valid string. Therefore, the positive

[5]In first-order RNNs, $M = |\Sigma'| + 1$ if a fixed 1-valued input is included for the bias weight.

sample $S^+$, that is supplied during training, should be considered as a stochastically generated sample, which depends on the state transition probabilities of the target stochastic source. It can be assumed that if the sample $S^+$ is large enough, then $S^+$ will also be structurally complete with respect to the target stochastic DFA containing the transitions with non-zero probability.

This stochastic nature of the learning problem also affects the selection of the prediction threshold to be used for string recognition by the net. In the case of stochastic regular languages, it is clear that the range of suitable values for the prediction threshold depends on the target stochastic DFA. More precisely, since the output-unit activation values are related to the state transition probabilities, the value $u$ of the prediction threshold must satisfy

$$0 < u < \min_{\forall q_i, q_j \in Q,\ \forall a_k \in \Sigma} \{ p(q_i, q_j, a_k) \mid p(q_i, q_j, a_k) > 0 \} \tag{4.95}$$

where $Q$ is the set of states in the target stochastic DFA and $p(q_i, q_j, a_k)$ denotes the probability of a transition from state $q_i$ to state $q_j$ through the symbol $a_k$. If the minimal non-zero probability of a transition in the target stochastic DFA is unknown, then the prediction threshold may be set heuristically to some small positive value $u$ (e.g. a value in the real interval $[0.05, 0.20]$), expecting that the chosen value satisfies the above condition.

On the other hand, the criterion for ending the training stage should be independent of a-priori knowledge of the target grammar or DFA, because otherwise, the problem becomes that of training a network to implement a known grammar, and it is not proper to refer to it as grammatical inference. Some possible criteria for ending the training stage are to reach a maximum number of string presentations or to stop when the total prediction error on the training set stabilizes [ClSM:89]. The criterion used by Smith and Zipser [SmZi:89], consisting of overpassing a predetermined number of correct predictions, requires prior knowledge of the target stochastic DFA to establish which are the legal succesor symbols at each step, and which threshold $u$ is adequate to confront with the activation values of the output units.

Servan-Schreiber, Cleeremans, and McClelland presented the results of learning the next-symbol prediction task for the two regular grammars shown[6] in Fig.4.5 using Elman's SRNs trained by back-propagation [SeCM:88, ClSM:89]. These two grammars have also been used as benchmark by other authors to compare the performance of their recurrent networks for next-symbol prediction [SmZi:89, Fahl:91b, SoAl:94]. For the *simple Reber grammar* (Fig.4.5a), the memory of at least two precedent symbols is

[6]Properly speaking, what is displayed is the corresponding DFAs for the two grammars.

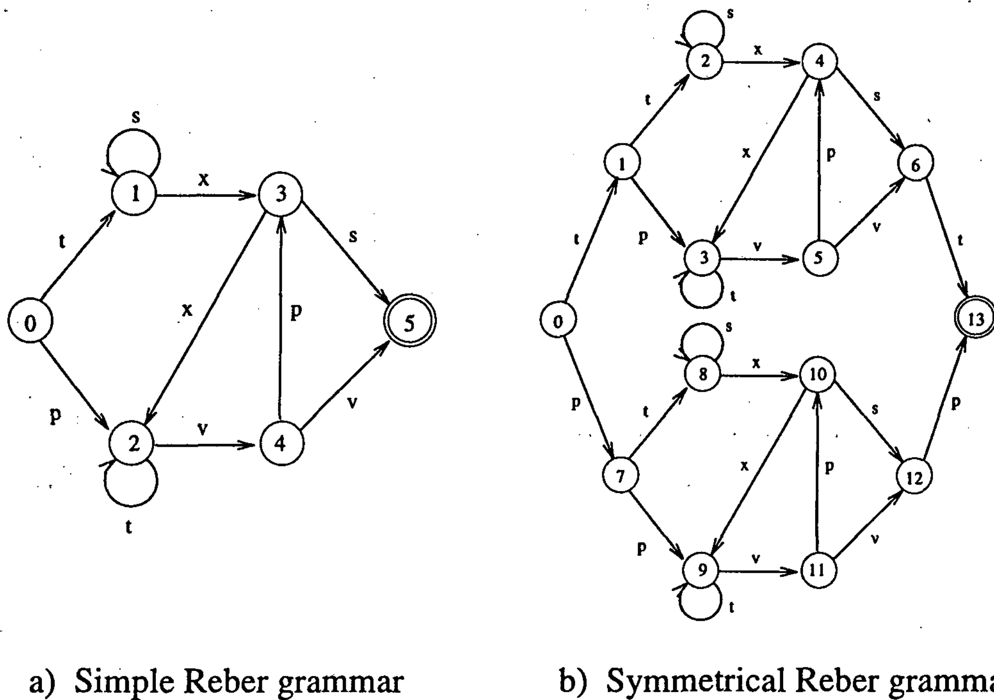


**Fig. 4.5** *Two regular grammars used as benchmark for next-symbol prediction.*

needed to predict correctly the next symbol at every step[7]. For the more complex *symmetrical Reber grammar* (Fig.4.5b), the last symbol of a valid string is determined by the first one (without considering the start and end $ symbols) and is independent of the substring in between. Therefore, the memory of the past sequence required to predict the last symbol is considerably larger in this case, since the net must note the initial 't' or 'p' and must retain this information while processing an embedded string of arbitrary length, what makes the prediction task much harder.

For each one of the two Reber grammars, Cleeremans *et al.* [ClSM:89] generated a training set of positive strings randomly, such that a probability of 0.5 was given to each of the two possible continuations for the states with two legal successors. The prediction threshold was set to 0.3, and two special symbols 'B' and 'E' were used to mark the beginning and end of the strings, respectively. Cleeremans *et al.* reported that the simple Reber grammar could be learned after 60,000 training strings

[7]It must be taken into account that when there are two (or more) legal successors from a given state, the network will never be able to do a perfect job of prediction. During training, the net will see contradictory examples, sometimes with one successor and sometimes the other, so the net will eventually learn to partially activate both legal outputs. In such cases, a prediction will be correct if the activations of the output units for the two legal successors are above the chosen threshold whereas the rest of output-unit activations are below it.

using an SRN with 3 hidden units and after 20,000 strings using an SRN with 15 hidden units. These were the best results obtained, not averages over a number of runs. On the other hand, the SRN was unable to learn the prediction task for the symmetrical Reber grammar, even with 15 hidden units and 300,000 training strings [ClSM:89]. It must be noted, however, that the SRNs (see Fig.4.3) were trained using back-propagation as learning algorithm, what means that the error gradient is only approximated by a truncated computation. Interestingly, the learning performance of the SRN for the difficult second grammar improved (correct prediction in about 70% of test strings) when the transition probabilities in the upper and lower copies of the embedded grammar were slightly altered so that the upper and lower copies were stochastically different.

Hence, Cleeremans *et al.* concluded that SRNs are not able to learn grammars in which the previous context is relevant for prediction at mid- or long-term but is not relevant at the intermediate steps before. Nevertheless, the SRNs are able to encode long distance sequential contingencies if the statistical properties of the embedded strings depend (even subtly) on the early information. The size of the network, i.e. the number of hidden units, is also a key factor that affects the temporal memory capacity and the required learning time. And, of course, the degree of success of the GI process, for a given network configuration, depends ultimately on the complexity of the target grammar.

A modification of the SRN architecture termed "Forced SRN" (or FSRN) was proposed by Maskara and Noetzel to improve the learning power of the SRN model [MaNo:92]. Specifically, they tried to overcome the failures in prediction that occur when two different states of the target DFA have the same output (i.e. the same legal successors and associated probabilities) and the SRN is not able to distinguish them, because it develops a similar code for both in the hidden layer activation pattern. The proposed modification consisted of adding two more pools of output units (in the output layer) which must be trained to reproduce the current input symbol and the contents of the context units, respectively. This tends to avoid the development of a similar activation pattern for two different states. Maskara and Noetzel reported better prediction results and GI performance of the FSRNs for two simple DFAs, which were not inferred by SRNs with the same number of hidden units due to the above state discrimination problem [MaNo:92].

Smith and Zipser presented the results of learning the next-symbol prediction task for the two Reber grammars in Fig.4.5 using first-order SLRNNs trained by the RTRL algorithm [SmZi:89]. These SLRNNs were also trained by means of pairs {*current symbol, next symbol*} within valid strings (positive examples) that were generated stochastically from each grammar. But, to the contrary of the works cited before, the state of the network was not reset at the beginning of each string; instead, the

network ran continuously over a sequence of examples by including a special pair {E, B} to link any two consecutive strings. A local encoding of the symbols was adopted both for the input signals and the trainable (output) units. In addition, some number of hidden units were included to provide the network with a representation space for state or context information. The recognition of grammatical strings was performed symbol by symbol in the way described previously, using the activation values of the output units as predictors of the corresponding symbols and a threshold of value 0.3 to decide the acceptability of the arriving input symbols.

Smith and Zipser reported the following results [SmZi:89]. Between 19,000 and 63,000 training strings were required for a first-order SLRNN, with 2 hidden units and sigmoid activation function, to learn the simple Reber grammar. The symmetric Reber grammar was only learned in some unspecified fraction of attempts, and successful runs ranged from 25,500 strings with 12 hidden units to 173,000 with 3 hidden units. These results, compared with those reported by Servan-Schreiber *et al.* for the same task using Elman's SRNs [SeCM:88], seem to indicate that first-order SLRNNs trained by RTRL, a true gradient-descent learning algorithm, learn better than SRNs trained by simple back-propagation.

As remarked earlier, the criterion to stop the training phase used by Smith and Zipser, which consisted of exceeding a predetermined number of correct predictions, either during weight updating or with frozen weights, is somewhat tricky from the point of view of grammatical inference, since the knowledge of the target DFA is needed to check the correctness of the prediction at each step (i.e. to know which symbols must have an associated activation above or below the threshold).

Fahlman further studied the prediction task for the two Reber grammars using the constructive recurrent cascade-correlation (RCC) approach [Fahl:91b]. Ten and twenty trials were run for the simple and symmetrical Reber grammars, respectively, each using a different training set. Each training set consisted of a fixed set of 128 (256) positive strings generated by the simple (symmetrical) Reber grammar, which was presented repeatedly. Fahlman reported an average of 25,000 string presentations to learn the simple Reber grammar using a pool of 8 candidate units; RCC achieved perfect prediction performance after building 2 hidden units in nine of the ten trials and 3 hidden units in the other. For the symmetrical Reber grammar, RCC needed an average of 202,000 string presentations to learn the prediction task in 11 of the 20 trial runs (55% success ratio), using a pool of 32 candidate units; for the successful runs, the number of hidden units built ranged from 5 to 15 [Fahl:91b]. Hence, the performance of the RCC architecture was rather similar in both cases to that of the RTRL-trained first-order SLRNNs.

In all the preceding works, the sigmoid function was used as activation function in all the units of the networks. In Chapter 6, the results of learning the next-symbol prediction task for the two Reber grammars using first-order SLRNNs and ASLRNNs with different activation functions are presented. These results, which have been partially reported in [AlSa:94a, AlSo:94, SoAl:94], display a great improvement both in learning time and success ratio with respect to the results of the studies that have been reviewed here.

To end this section, I will mention a work by Castaño *et al.* [CaCV:93] that dealt explicitly with the inference of stochastic regular grammars by recurrent networks through next-symbol prediction. The neural net and learning algorithm (Elman's SRNs trained by back-propagation), trained task, and symbol representation were identical to those used by Cleeremans *et al.* [ClSM:89], which have been already described. Three stochastic DFAs were studied; the former two, which only differed in the adopted transition probabilities, generated strings that begin with the substring "ba", include any number of 'c's except two and four, and end with "ae"; the third stochastic DFA corresponded to the simple Reber grammar, but such that the probability of the loops was increased over 0.5. For each stochastic grammar, a training set of 30,000 strings was generated. The training scheme consisted of iterating a process of selecting randomly 5,000 strings from the training set and evaluating the net on a validation set, until a total number of 200,000 (repeated) training strings were presented to the net. The validation set for each grammar contained 10,000 examples not included in the training set.

Let $p_1 p_2 ... p_{|s|}$ be the probability of generating a given string $s = c_1 c_2 ... c_{|s|}$ by the stochastic regular grammar to be simulated, where each $p_i$ corresponds to the probability of the grammar rule used to generate the $i$-th symbol; and let $q_1 q_2 ... q_{|s|}$ be the *predicted probability* of the same string with the inferred network, where $q_i$ corresponds to the output activation associated with the $i$-th symbol of the string. Castaño *et al.* considered that the string $s$ was *correctly estimated* by the inferred net if the condition

$$\left| \ln \left[ \frac{(p_1 p_2 ... p_{|s|})^{1/|s|}}{(q_1 q_2 ... q_{|s|})^{1/|s|}} \right] \right| < Thr \qquad (4.96)$$

was verified, where $Thr$ is a chosen threshold and the expression between brackets corresponds to the *normalized likelihood quotient.* Since the aim is to achieve $p_i = q_i$, for $i = 1, ..., |s|$, the leftmost term should be very close to zero for every string in the validation set [CaCV:93].

However, the above criterion presents the drawback of being independent with respect to the ordering within the string. This is, let $s$ be a string such that $\exists i, j \in [1..|s|]$, $i \neq j$, $p_i = q_j$, $p_j = q_i$, and $p_k = q_k$ for $k \in [1..|s|]$, $k \neq i, j$; the criterion given by Eq.(4.96) would compensate these two wrong probabilities and the

string $s$ would be erroneously considered as correctly estimated. To avoid this kind of situations, Castaño *et al.* [CaCV:93] suggested another criterion, which is given by the following expression:

$$\frac{1}{|s|} \sum_{i=1}^{|s|} |\ln p_i - \ln q_i| < Thr. \quad (4.97)$$

The results reported by Castaño *et al.* for the inference of the three stochastic DFAs aforementioned using Elman's SRNs and the two preceding criteria showed that the outputs of the trained networks really approached the probabilities of the corresponding DFAs. Moreover, the correct string estimation ratio on the validation set after training was close to 100% in all the cases studied (using the former criterion and taking $Thr = 0.08$ for the first two DFAs and $Thr = 0.15$ for the third one) [CaCV:93]. More recently, Carrasco *et al.* have studied the inference of stochastic DFAs using second-order ASLRNNs with similar good results [CaFS:96].

### 4.2.2 GI from positive and negative examples: the string classification task

Let $S = (S^+, S^-)$ be a sample of a language $L$ over an alphabet $\Sigma$. Again, let $\Sigma'$ refer to the extended alphabet $\Sigma \cup \{\$\}$, where now the special symbol \$ is only used as an end-of-string mark, and let $S_\$$ denote the sample obtained by converting each string $s \in S$ into the form $s\$$. A recurrent neural network can be trained to classify the strings in $S$ (or $S_\$$), i.e. to discriminate between the positive and negative examples (*string classification task*). If the RNN is powerful enough to learn the task but small enough to generalize the supplied data, the trained network may be used as an acceptor of a language $L'$ such that $L' \supseteq S^+$ and $L' \cap S^- = \emptyset$. In the best cases, the final network accepts the target language, i.e. $L' = L$. Hence, by training a RNN to learn the string classification task, a GI process from positive and negative examples is carried out.

Unfortunately, the conditions on the language $L$, the sample $S$, and the used RNN, that are required for the net to perform a successful inference, are still not well understood. The reported works that have followed this connectionist approach to GI have basically dealt with the problem of inducing regular languages [Poll:91, GiMC:92, WaKu:92, MiGi:93, ZeGS:93]. Some techniques to extract a DFA from the trained network have also been proposed, which will be reviewed in the next section. Although it was not clearly stated in the cited papers, if the network is wanted to emulate a target DFA accepting $L$, then the sample $S$ taken as training set should be structurally complete with respect to such DFA. This requirement becomes apparent, according to the RGI theory, if the trained network is used to guide a symbolic state

merging process from the sample prefix tree automaton [AlSa:94b] (see Chapter 6).

As in the works reported on the next-symbol prediction task, a local encoding has been typically used to represent the alphabet symbols in the RNNs aimed at learning the string classification task[8]. In this case, there is one binary input signal associated with each symbol, that is activated when the corresponding symbol is input during the sequential presentation of a string. In some of the works reported on the string classification task, a special end-of-string symbol is appended to each string presented to the network (e.g. [GiMC:92, MiGi:93]), so that the sample $S_\$$ over the alphabet $\Sigma'$ is presented during the training stage; in other works, however, the end-of-string symbol is not used and the original sample $S$ over $\Sigma$ is supplied to the net [ZeGS:93]. Anyway, the internal state of the network, given by the recurrent unit activations, is normally reset at the start of each string.

If a local encoding is followed, the number of input signals $M$ will be equal to $|\Sigma|+1$ or $|\Sigma|$ depending on whether the end-of-string symbol is used or not (plus one, if a fixed 1-valued input is included for the bias weight). In fact, for most of the RNN architectures, the addition of the end-of-string symbol is not needed to learn the string classification task. Goudreau and Giles have shown that first-order SLRNNs need to be augmented to implement any DFA, and the use of the end-of-string symbol can be seen as a way of augmenting the network, since it allows for an additional time step before the network output for the string is yielded [GoGi:93].

On the other hand, just a single output unit ($P=1$) is required to classify a given string $s$ as positive or negative. The activation value of this output unit after the string presentation, $o_1(t_f(s))$, which is a value in the interval $[0,1]$, is interpreted as the degree of acceptance granted by the net to the string $s$. Thus, $s$ will be accepted if $o_1(t_f(s)) > 1-\epsilon$ and it will be rejected if $o_1(t_f(s)) \leq \epsilon$, where $\epsilon$ is a tolerance threshold such that $0 < \epsilon \leq 0.5$. Note that, for $\epsilon < 0.5$, a string $s$ may be neither accepted nor rejected by the net[9].

In order to learn the string classification task, the network is trained using a set of string-response $(s,r)$ pairs, that includes all the strings in $S$, where $r = T_{accept}$ if $s \in S^+$ and $r = T_{reject}$ if $s \in S^-$, and the constant target values $T_{accept}$, $T_{reject}$ are chosen such that $T_{accept} \simeq 1$ and $T_{reject} \simeq 0$. Although $T_{accept} = 1$ and $T_{reject} = 0$ seems the most logical selection [MiGi:93, ZeGS:93], other target response values have been

[8] Actually, nothing forces to do so, and a non-local distributed representation of the input symbols might be valid as well.

[9] This kind of symmetry with a possible uncertainty, that is implicit in the problem of GI from both positive and negative examples, has inspired the use of *Unbiased FSA*, which will be defined and studied in Chapter 5, for symbolic and hybrid RGI.

used in some works (e.g. $T_{accept} = 0.8$ and $T_{reject} = 0.2$ in [GiMC:92]; $T_{accept} = 0.9$ and $T_{reject} = 0.1$ in [WaKu:92]). In any case, the proper response $r$ is used as target value $d_1(t_f(s))$ for the output unit at the time step $t_f(s)$, when the full string $s$ has just been presented[10]. The error value $e_1(t)$, corresponding to the output unit at time step $t$, will be zero for all the previous instants from the beginning of the string presentation, $t_b(s)$, and therefore, the error $E_{total}(t_b(s), t_f(s))$ associated with the string $s$ will be given by

$$E_{total}(t_b(s), t_f(s)) = \frac{1}{2} \left( d_1(t_f(s)) - o_1(t_f(s)) \right)^2. \tag{4.98}$$

Unlike the next-symbol prediction task, the total error on the training set should approach to zero after a successful learning. Hence, this is a possible criterion to stop training the net. Another stop criterion, which is less strict, consists of reaching a 0% percentage of string classification error over the whole sample; i.e., all the strings in $S^+$ must be accepted and all the strings in $S^-$ must be rejected by the trained net, according to the chosen tolerance $\epsilon$. Other stop criteria can be used when a network fails to learn the task, such as to reach a maximum number of training epochs or to detect that the network has converged to a (local) minimum of the total squared error (TSE) function. Note that, in the string classification task, the TSE value in a global minimum is expected to be approximately zero.

A set of seven simple regular grammars, originally selected for study by Tomita [Tomi:82], has been used as benchmark in several studies about RGI using different types of RNNs trained to learn the string classification task [Poll:91, GiMC:92, WaKu:92, MiGi:93, DaMo:94]. Fig.4.6 displays the minimal-size fully-defined DFA for each one of the Tomita's grammars. It can be observed that the number of states in these DFAs is between 2 and 5. The corresponding regular languages, all of which are over the binary alphabet $\Sigma = \{0, 1\}$, can be described as follows:

$T1$ : $1^*$

$T2$ : $(10)^*$

$T3$ : all the strings not containing an odd number of consecutive 0's after an odd number of consecutive 1's

$T4$ : all the strings not containing "000" as a substring

$T5$ : all the strings with an even number of 0's and an even number of 1's

$T6$ : all the strings such that $|\#1\text{'s} - \#0\text{'s}| \bmod 3 = 0$

$T7$ : $0^*1^*0^*1^*$

[10] As indicated previously, the final time step $t_f(s)$ may be defined as the one when the last symbol of $s$ (or the end-of-string symbol \$, if used) is presented to the net.

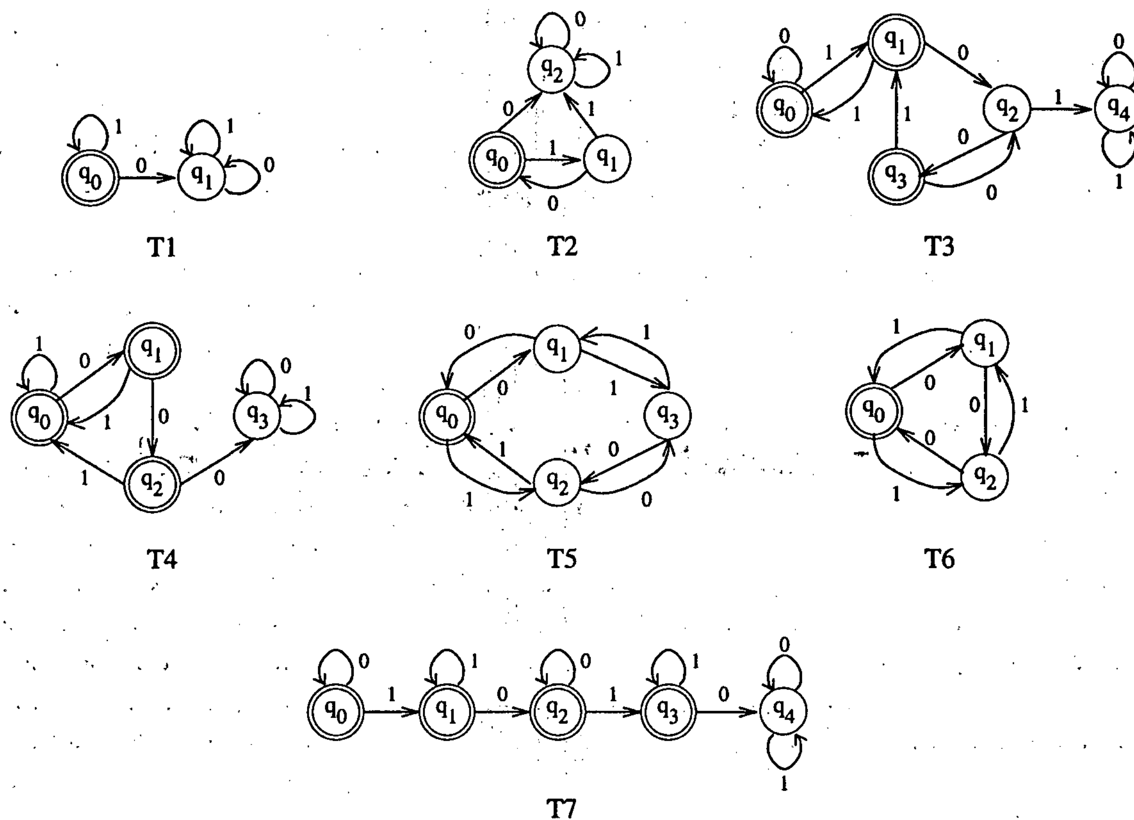


**Fig. 4.6** *The seven Tomita's regular grammars used as benchmark for string classification by RNNs.*

Tomita also selected for each language a set of positive and negative examples to be used as a learning set. Around 20 short strings, with a rather similar number of positive and negative examples, were contained in each of his training sets. Hence, the selected training sets were really small sparse samples of the target languages. Nevertheless, since languages $T1$ and $T2$ are very sparse, their corresponding positive samples covered practically all the short grammatical strings up to length 10; the rest of languages are more dense, and they were sparsely covered by their positive samples. By using a method of heuristic search over the space of FSA with up to eight states, Tomita was able to induce a recognizer for each of the seven languages defined [Tomi:82].

Pollack [Poll:91] and Watrous and Kuhn [WaKu:92] tried the induction of recognizers for languages $T1$-$T7$ from the original Tomita's training sets[11], using a type of second-order SLRNNs with a non-recurrent output unit and 3 recurrent hidden

[11] With the only difference that the occurrences of the empty string $\lambda$ were removed in the training sets used by Pollack [Poll:91].

units, including both second-order and first-order connections[12]. In both works, a single input signal was used to represent the input symbols in a non-local manner, but an end-of-string symbol was only used in [WaKu:92]. However, the main difference between both studies was that while Pollack used a learning algorithm that computed a truncated approximation to the gradient, Watrous and Kuhn used a real gradient-descent algorithm where the complete gradient was calculated through BPTT.

Pollack performed 10 runs on each training set with distinct random starting weights and trained the networks until all the examples were correctly classified (using $\epsilon = 0.4$) or a total number of 1,000 epochs was reached. He reported that none of the seven Tomita's languages were induced, and furthermore, only for $T1$ and $T4$ his architecture was able to learn to classify the sample strings in the 10 runs [Poll:91].

Watrous and Kuhn [WaKu:92] performed 5 runs on each training set, training the networks until a minimum of the mean-squared error measure was reached, using $T_{accept} = 0.9$ and $T_{reject} = 0.1$, with no maximum number of epochs fixed. All the binary strings up to length 10 were used for testing the generalization of the trained networks (using $\epsilon = 0.1$). For languages $T1$ and $T2$, two of the five runs led to a 100% successful generalization, and the language was recognized nearly correctly by a third network. For the rest of Tomita's languages, even though the training sets could be correctly learned, none of the runs led to the target language recognition, yielding rather poor classification rates on the test sample.

Watrous and Kuhn performed two further experiments [WaKu:92]. Using a larger training set containing 205 randomly selected strings of length 10 or less, language $T4$ was perfectly induced in 1 run and nearly correctly induced in other 3 of the 5 runs attempted. On the other hand, language $T5$ was not inferred in any of the 5 runs performed using a larger network with 11 hidden units, obtaining in every case a convergence to a small MSE value in the training phase, but a bad generalization on the test set.

Giles *et al.* presented in [GiMC:92] some results of learning the string classification task for the language $T4$ using second-order SLRNNs with dynamics given by Eq.(4.37) and trained by the second-order version of the RTRL algorithm (i.e. a true gradient-descent algorithm). Instead of using Tomita's small training set, Giles *et al.* randomly chose a global training set $S$ of 1,024 strings from the set of all binary strings of length less than 16, and they trained the networks incrementally in several cycles, starting on a small working set $S_0$ (of 32 strings) and adding up at each cycle some small number of strings from $S$ that had not been correctly classified at the previous cycle, until the whole training set $S$ was learned (using $\epsilon = 0.2$) or a maximum nunber of 20 cycles

[12]Equation (4.39) gives the dynamics of a recurrent network of this type.

was reached. For each cycle, a maximum number of epochs was also set. The target response values were $T_{accept} = 0.8$ and $T_{reject} = 0.2$. An end-of-string symbol was used, and the initial hidden unit activation values were never reset for on-line training.

Giles *et al.* [GiMC:92] measured the generalization capacity of the trained nets using three different combinations of test sample and tolerance. The largest test sample consisted of a randomly chosen set of 850,000 strings of length 16 to 99, for which the maximum tolerance $\epsilon = 0.5$ was used. For this test set, the number of runs, with distinct starting random weights, that led to perfect generalization for $T4$, after varying the number of neurons $N$ between 3 and 5, was 6 of 10 for $N = 3$, 4 of 11 for $N = 4$, and 2 of 6 for $N = 5$ (i.e. the smaller the number of neurons, the better the generalization performance). Curiously, although the size of the final working set for each run was much smaller than the size of the training set $S$ (between 42 and 72 strings in front of the 1,024 strings in $S$), nearly all networks converged during training, what it means that the complete training set $S$ was learned. For all cases that converged, the FSA accepting $T4$, which is shown in Fig.4.6, was extracted from the trained net through a clustering technique (see next section) followed by an FSA minimization. Indeed, some of the extracted FSA were perfect recognizers for language $T4$ whereas the trained networks, from which the FSA were extracted, were not. Giles *et al.* reported that similar generalization and FSA extraction results were obtained for the other Tomita's languages, but these experimental results were not presented [GiMC:92].

Miller and Giles presented in [MiGi:93] an experimental comparison of first-order and second-order SLRNNs (both with sigmoid activation function), as applied to the string classification task for grammatical inference, using the seven regular grammars by Tomita[13] and a randomly generated DFA of 10 states as benchmark. The training scheme used was very similar to the one that has been described for the preceding work ([GiMC:92]). In this case, an incremental training was performed using as training set a complete sample (instead of a random sparse sample) containing all the binary strings up to length 9 (1,023 total), presented in alphabetical order. Training proceeded for a maximum of 10 cycles of 500 epochs each, starting on a working set of 50 strings and adding up to 50 additional (incorrectly classified) strings at each cycle. Miller and Giles [MiGi:93] performed training simulations for first- and second-order SLRNNs with 3 to 9 neurons; more precisely, 10 runs were performed for each of these 14 network configurations using training sets $(S^+, S^-)$ corresponding to the 8 benchmark grammars. The successfully converged networks were tested for generalization on all strings of length 10 to 15 (64,512 total).

[13] Actually, the language $T5$ described earlier was replaced by the language $T5' = ((01|10)(01|10))^*$, associated with a minimal fully-defined DFA of 7 states.

Second-order SLRNNs converged, i.e. the training set was learned, in all the runs on Tomita's languages except a few for $T5'$ and $T7$ with $N = 3$, whereas first-order SLRNNs (without bias weights) converged a lesser number of times, with the main difficulties in languages $T6$ and $T7$ up to $N = 8$ and in languages $T3$ and $T5'$ up to $N = 4$. Concerning the more difficult random 10-state DFA, first-order SLRNNs consistently failed to converge, while second-order SLRNNs showed consistent convergence success for $N = 7, 8, 9$. Therefore, it was concluded that "first-order nets are less reliable, in terms of likelihood of convergence, particularly with small numbers of neurons". In general, a lower average number of epochs (for the runs which did converge) was shown for second-order SLRNNs, although it must be noted that the computation time of an epoch for second-order SLRNNs is longer than the time for first-order SLRNNs with the same $N$, due to the greater cost of the learning algorithm. Also, as it might be expected, faster convergence was achieved for larger numbers of neurons. Miller and Giles reported that no improvement on successful convergence occurred by adding a bias weight to each unit both for first- and second-order SLRNNs [MiGi:93].

Two generalization tests were carried out at tolerance $\epsilon = 0.2$ and $\epsilon = 0.5$, respectively, for the networks that converged. For $\epsilon = 0.5$, both first- and second-order nets performed well, since languages $T1$ and $T2$ were perfectly induced and the average number of errors for the other languages was less than 1% of the test set in all cases, with the worst performance on $T5'$ and $T7$. However, the generalization results for $\epsilon = 0.2$ were considerably worse, except for $T1$ and $T2$, for which the test results were near perfect. In this test, two trends were observed: (a) second-order SLRNNs generalized worse than first-order nets; and (b) the classification errors generally increased with larger numbers of neurons. The worst results were obtained for language $T6$ and second-order SLRNNs, for which the error rates ranged from 12% to 25% of the test set. In addition, Miller and Giles presented diagrams in which convergence time and generalization performance appear not to be correlated, i.e. neither quick nor slow convergence implies a better generalization [MiGi:93].

Finally, Miller and Giles applied a DFA extraction method (described in [GiMC:92]) to the successfully converged networks and counted the number of matches between the resulting DFAs and the target DFAs. They reported results showing that target DFAs were more often extracted from second-order SLRNNs, except for language $T5'$, though for this language both architectures performed rather poorly. The random 10-state DFA could also be extracted by second-order SLRNNs in some unspecified fraction of attempts. The average classification performance of the extracted DFAs on the test set was not reported. On the other hand, the average size of the DFAs before minimization was always larger for the DFAs extracted from first-order nets. This was interpreted as a corroboration of the studies by Goudreau *et al.* [GoGi:93, GoGC:94], which indicated that first-order SLRNNs must use *state-splitting* to learn

Finally, Manolios and Fanelli [MaFa:94] applied the first-order 2-layer RNN architecture that has been described in Section 4.1.5 to the problem of inferring the DFAs associated with Tomita's languages $T1$, $T2$, $T4$ and $T6$, from positive and negative string examples. The networks selected included one input signal and just two state units with one of them designated as output unit; in addition, two hidden units were used for $T1$ and $T2$, while three hidden units were used for $T4$ and $T6$. The target response values were $T_{accept} = 1.0$ and $T_{reject} = 0.0$.

The training sets used for all the tested Tomita's languages consisted only of all strings up to length 4, including the empty string. Thus, the set of training strings used was small (31 strings) and their length very short. For the generalization test phase, 8 test sets were chosen, namely all strings of length $\leq 5$, $\leq 6$, ..., $\leq 12$; and for language $T6$, two additional test sets of 1,000 random strings of length $\leq 100$, and 1,000 random strings of length $\leq 1,000$, were selected respectively. A DFA extraction method, which is described in the next section, was also applied for each trained network and test set [MaFa:94].

Manolios and Fanelli reported that the small first-order 2-layer RNNs aforementioned were able to learn the string classification task for all the Tomita's languages attempted, and many of the networks (without saying precisely which part of them) could generalize from the training sets to strings of length 12. For the language $T6$, 2 of the 4 runs led to networks that generalized perfectly on strings up to length 1,000. In most cases, the extracted DFAs were either the target minimal DFAs or larger equivalent DFAs that could be reduced to the former by DFA minimization [MaFa:94].

## 4.3 FSA extraction from RNNs for regular GI

We have seen in the preceding section that several types of RNNs can be used for grammatical inference, either by learning to predict the next symbol in a valid string from a given set of positive examples, or by learning to classify the grammatical/ungrammatical strings from a set of both positive and negative examples. Although, in principle, the complexity of the target languages to be induced is not restricted by the definition or the implementation of these two learning tasks, the reported works have only dealt with the inference of (simple) regular languages. In these cases, the trained network is wanted to emulate an FSA accepting the target regular language. The topic of the similarity between RNNs and FSA, or in other words, how a trained RNN manages to behave as an FSA, has been studied by several researchers. Moreover, it has been shown that the symbolic representation of an FSA,

which is approximately emulated by the net, can be extracted from the dynamics of the RNN, and various FSA extraction methods have been proposed, which are summarized in this section.

Some previous studies have shown, either by means of hierarchical clustering analysis or through graphical representations, that both SLRNNs and ASLRNNs tend to develop an internal state representation in form of clusters in the activation space of the recurrent hidden units when they are trained to learn the next-symbol prediction or the string classification tasks [ClSM:89, SmZi:89, GiMC:92, ZeGS:93]. A similar behavior has been shown for the recurrent state units in 2LRNNs [MaFa:94]. The following related phenomena have been discovered:

a) during the earlier steps of learning (*decide stage*), the clusters corresponding to distinct states overlap [ZeGS:93,MaFa:94];
b) during the later learning (*reinforce stage*), the formed clusters contract, i.e. decrease their variance [ZeGS:93,MaFa:94];
c) after a successful learning with a sufficient small number of hidden units, the activation pattern clusters may represent the states of the minimal target FSA [SeCM:88,SmZi:89];
d) after a successful learning with a non-minimal number of units, more than one pattern cluster may be formed for the same state of the minimal target FSA, each one associated with a different group of incoming paths[15] [SmZi:89, ClSM:89]; usually, an equivalent FSA containing a larger number of states is simulated that reduces to the target FSA after minimization [GiMC:92,MiGi:93];
e) during the generalization test with strings longer than those used in the training set, the state clusters are typically unstable due to drifting activations, and this causes a severe impairment in the network generalization performance [ZeGS:93,MiGi:93];
f) after training 2LRNNs, when the clusters are sufficiently tight (their standard deviations are below some threshold) the network is able to generalize perfectly, i.e. to classify strings of arbitrary length correctly; on the other hand, when clusters deviations are larger (above some threshold), the clusters loosen as the network is presented with longer strings and string misclassification occurs [MaFa:94].

Consequently, a symbolic description of the FSA that is approximately simulated by a trained RNN can be extracted from the network dynamics along a pass through the training set (non-training epoch) by determining the distinct activation clusters,

[15]The influence of the previous context in the state representation decays rapidly and depends on the network size [SmZi:89].

mapping them to states, and annotating the transitions that move the activation pattern from one cluster to another or make it stay within the same cluster.

The problem of the unstability of state clusters for long input strings may be solved by using a self-clustering RNN with discretized activations (*rigid quantization*; Zeng *et al.* [ZeGS:93]) or a clustering module integrated in the learning architecture (*adaptive quantization*; Das and Mozer [DaMo:94]). In both approaches, which have been already described, the trained network develops a discrete state representation, such that each state is associated with a single isolated point in the activation space of the hidden units. In the former case, a heuristic pseudo-gradient learning algorithm can be used to train the network weights [ZeGS:93], while in the latter case, a true gradient-descent learning technique can be applied to obtain both the network weights and the clustering parameters [DaMo:94]. In these two discrete network approaches[16], the FSA extraction method is simply based on labeling the different vectors in the activation space with distinct states (since there are only isolated points instead of clusters) and recording the input symbols that yield the transitions between them. Note that, obviously, only regular languages can be inferred using these discrete neural GI approaches.

On the other hand, if a continuous network (i.e. an RNN with a continuous activation function in the recurrent units and no clustering module) is selected or preferred for learning a regular GI task and a symbolic result is desired, then a clustering-based FSA extraction method is required. Indeed, it has been demonstrated that by using clustering techniques, it is possible to extract a symbolic FSA from the RNN dynamics during or after learning [GiMC:92,MaFa:94] and to improve the network generalization performance a posteriori [DaDa:91,GiOm:93]. Several techniques have been proposed: hierarchical clustering [SeCM:88,ClSM:89], dynamic clustering [DaDa:91], search on a regular partition [GiOm:93] (also used in [GiMC:92,MiGi:93]), moving markers [MaFa:94], and the $k$-means algorithm [ZeGS:93]. The last two methods return FSA that contain a predetermined number of states; for these two methods and for the regular partition method as well, a subsequent symbolic process of FSA minimization (recall the Moore's minimization algorithm described in Chapter 2) has been suggested to obtain the final extracted FSA. Since the hierarchical clustering and the $k$-means algorithm are well-known methods for non-supervised classification in statistical pattern recognition (e.g. see [DeKi:82]), only the rest of FSA extraction methods proposed, which incorporate some specific features of the problem domain into a clustering algorithm, are reviewed hereinafter.

Das and Das [DaDa:91] proposed an iterative process using a dynamic clustering algorithm and a performance test to stabilize Elman's SRNs after learning (i.e. to solve the drifting activations problem), which can be used as well to extract a DFA from a

[16] and also in the recovery of an inserted FSA (see Chapter 7)

trained SRN. Once the training stage is complete, the set $\zeta$ of all hidden activation patterns over a non-training epoch must be collected. The clustering algorithm uses a parameter $\gamma$, which determines how close the state representations can be in the activation space, and which is initialized to a large value. If a pattern $\zeta_i$ is close enough to any of the existing clusters (i.e. if the distance between $\zeta_i$ and the cluster mean is less than $\gamma$), then it is included in the nearest cluster, and the mean and variance of that cluster are updated. If $\zeta_i$ is not close enough to any of the current clusters then a new cluster is formed with mean as $\zeta_i$ and variance as $\gamma$.

After all patterns from $\zeta$ have been classified, the network performs a new pass through the training set to test the goodness of the clustering. In this non-training epoch, instead of feeding back the actual activations of the hidden layer, the mean activation pattern of the cluster to which the actual pattern belongs is fed back in the next time step. If the total squared error $E_{total}$ over this epoch is below a certain tolerance (another parameter), the process is stopped; otherwise, the value of $\gamma$ is decremented by a small amount and the whole procedure of clustering is repeated until performance is satisfactory. Das and Das claimed that the above method converges with certainty only after the network has reasonably learned the task, and the obtained clusters tend to become well defined once an adequate value of $\gamma$ has been determined [DaDa:91].

During the generalization test on unseen strings, the prototype pattern (cluster mean) of the state to which the actual hidden activation pattern belongs is fed back, where an activation pattern is considered to belong to a state if it falls within the variance of the corresponding cluster. Alternatively, a DFA can be extracted from the network, just by registering the transitions among the obtained clusters that occur in the test pass through the training set, and this DFA can be used thereafter instead of the stabilized network. Das and Das reported that the automata extracted in their experiments were not the minimal target DFAs (for *mod* 2, *mod* 3, and *mod* 4 regular languages), but they were equivalent to them [DaDa:91].

Giles and Omlin [GiOm:93] described a simpler approach to partition the activation space of the recurrent units in second-order SLRNNs and extract a DFA from the dynamics of a network trained to learn a string classification task. Since an activation function $g$ with range $[0, 1]$ (e.g. a sigmoid) is assumed to be used in the network, the objective was to identify clusters in the bounded activation space (or state space) $[0, 1]^N$, where $N$ is the number of recurrent units. Giles and Omlin proposed a dynamical state space exploration to identify the DFA states, which avoids the computationally infeasible exploration of the entire space.

Their DFA extraction method divides the activation range of each of the $N$ neurons into $q$ intervals or *quantization levels* of equal size, producing a partition of $q^N$ regions in

the state space. Starting in a defined initial network state, a string of inputs causes the network to follow a discrete state (region) trajectory connecting the actual activation patterns in the continuous space $[0,1]^N$. All strings up to a certain length are presented to the net in lexicographic order, thus generating a search tree with the initial state as its root and arity equal to $|\Sigma|$, the number of alphabet symbols. In this search tree, nodes correspond to hypothetical DFA states and arcs correspond to transitions. A breadth-first search is performed trying to discover new states and transitions according to the three following rules:

i) when a region is reached that has not been visited previously, then a new DFA state is created and a new transition is defined between the preceding and the current state;
ii) when a previously visited region is reached, then only the new transition is defined between the preceding and the current state and the search tree is pruned at that node;
iii) when an input causes a transition to the same region, then a loop transition is created and the search tree is pruned at that node.

A DFA state is designated as final (or accepting) state if the activation value of the unique output unit (one of the recurrent units) is larger than 0.5; otherwise, the state is designated as non-final. The DFA extraction algorithm terminates when no new DFA states are created from the string set chosen and all possible transitions from all the created states have been extracted. Usually, only a small subset of all the partition regions are visited in the search and become DFA states [GiOm:93].

It is clear that the DFA extracted by Giles and Omlin's method depends on the quantization level $q$ chosen, and, in general, different DFAs will be extracted for different values of $q$. Furthermore, different DFAs may be extracted depending on the order of strings presented which leads to different successors of a node visited by the search tree. However, by applying a DFA minimization algorithm a posteriori, many different DFAs extracted for a variety of initial conditions ($q$ and $N$ values, string orders) will typically collapse into the same equivalence class represented by a minimal DFA [GiMC:92, GiOm:93].

It must be remarked that the above extraction method does not guarantee the obtaining of a DFA *consistent* with the given training set (i.e. one that correctly classifies all the involved strings). On the other hand, if several DFAs are extracted with different quantization levels, then one or more of them may be consistent with the training set. For such a case, Giles and Omlin [GiOm:93] proposed a heuristic algorithm that selects the "best" hypothesis among the consistent extracted DFAs. Let $A_q$ be the minimized extracted DFA corresponding to a given quantization level $q$. Then, a sequence of DFAs $A_2$, $A_3$, ..., can be generated, and the best model is

the first consistent DFA $A_k$ found in this order. This heuristic rule was motivated by some simulation results, which showed that the consistent DFAs extracted using small values of $q$ provided the best generalization performance. In general, consistent DFAs always outperform the trained network from which are extracted, and their generalization performance seems to approach asymptotically the network performance with increasing quantization level $q$ [GiOm:93]. An explanation for these results was given by Giles and Omlin: "large consistent DFAs tend to overfit the given training set and thus yield poorer generalization performance".

Manolios and Fanelli [MaFa:94] developed a DFA extraction method based on vector quantization, and they applied it to the activation space of the state units in first-order 2LRNNs. Again, the state space was considered to be the $N$-dimensional unit hypercube $[0,1]^N$. Their DFA extraction algorithm is given as input a set of network states (activation patterns) yielded by the network in response to a string set, and analyzes them trying to identify the underlying clusters. To this end, $n$ *markers* are distributed randomly within the hypercube, and for every network state, the closest marker is moved toward it a certain distance $d$, which is equal to the distance between the marker and the network state divided by the number of times the marker has been moved plus one. This guarantees that any marker moved will be exactly in the centroid of the network states it is closest to. Consequently, every marker moved represents one cluster of network states.

Nevertheless, the FSA that is built from the final clusters may sometimes be non-deterministic, since two or more network states within one cluster may have different transitions for the same symbol. Manolios and Fanelli suggested that whenever an NFA is obtained after clustering, the extraction algorithm must restart itself with a different set of randomly distributed markers. Several such iterations may be performed until a DFA can be extracted. Finally, the application of a minimization algorithm to the extracted DFA was recommended to obtain the final DFA [MaFa:94].

Manolios and Fanelli did not explain how the DFA states are designated as accepting or non-accepting. At first, there are two options: either the activation value of the output unit is used to discriminate the network states (as in [GiOm:93]), or the positive/negative labels of the training strings are used to label the network states reached after string presentation. However, in both cases, it could happen that a cluster contained both an accepting network state and a non-accepting network state, and thus an inconsistent DFA were obtained. Actually, the issue of the DFA consistency with the given training set was not discussed in [MaFa:94]. However, a possible solution could be to restart the clustering procedure whenever an inconsistent DFA is obtained (similarly to the case of NFAs).

## 4.4 Connectionist approaches to context-free GI

All the RNNs seen so far basically behave as FSAs or FSMs, since the network's next state and output depend only on the current state and input. It must be noted, however, that the network state space is potentially infinite if continuous activation functions are employed in the recurrent state units (and an analog or high precision computation capability is assumed for network operation). In fact, this continuous state space is what obliges to use clustering techniques for FSA extraction from such an RNN. Nonetheless, the described RNNs display severe limitations in processing and learning high-level languages beyond the regular languages. A discussion of the approaches that have been attempted to overcome these limitations, and a nice hybrid architecture specifically oriented to the representation and learning of pushdown automata (PDA), which will be reviewed later in this section, were presented by Sun *et al.* in [SuGC:93].

A brute-force method to enhance the computational power of an RNN is to increase the size of the existing network structure while training on a more complex language such as a CFL. The assumption is that the RNN size has no bound, and the knowledge gained as the network grows gives clues to the representation of the underlying grammar. In practice, this is difficult to achieve, and what usually happens is that the trained RNN will only recognize the complex language up to a certain string length (i.e. a regular language), and to generalize correctly on longer unseen strings, it will need to be re-trained on those strings. This kind of approach was followed, for example, in the connectionist natural language works by Allen [Allen:90] and Moisl [Moisl:92]. Allen reported that a certain type of RNN, which behaves like an FSM, was able to learn some *finite* CFGs [Allen:90]. Moisl trained Elman's SRNs to implement FSMs which simulated some *deterministic pushdown transducers* (DPDTs) [Moisl:92].

Also aimed at natural language processing, Wyard and Nightingale presented a simple non-recurrent network (HODYNE) that learns to behave as a CFL recognizer when supplied with a training set of positive and negative strings [WyNi:90]. The HODYNE architecture consists of a (so-called) "high-order" input layer, in which each input node is associated with a different tuple (pair, triple, etc.) of consecutive terminal symbols[17] in the input strings, an output layer with a number of units, each connected to all or part of the input nodes, and no hidden layer. The network is dynamic in the sense that new input nodes and connections can be created during training as new tuples of symbols appear. For a given input string, the input nodes are just "ON" (valued 1) or "OFF" (0), depending on whether the corresponding tuple is present in

[17]The terminal symbols of the grammars in the paper are called *preterminals* [WyNi:90], since they correspond to syntactic categories of words, so that the grammatical strings represent language sentences as sequences of such preterminal symbols (e.g. *det adj noun verb prep det noun*).

the string or not, and the output units compute a simple weighted sum of their inputs, with no thresholding or activation function. Only the output unit with the highest net-input fires, so there is a single output unit winner. For the task of grammatical recognition, only two output units are needed to answer "YES" or "NO", respectively.

The learning algorithm proposed for HODYNE was quite simple and rather similar to the perceptron convergence procedure of Rosenblatt [Rose:62]. Weights are only adjusted when the network produces the wrong output. In this case, the links between the activated input nodes and the output winning unit are weakened, and the links between the activated input nodes and the desired response output unit are strengthened. A particular mathematical function was selected to compute the weight changes, such that the weight values are modified smoothly and they do not depart too far from their initial value ($w = 1$) [WyNi:90]. In addition, an incremental training procedure including several cycles was recommended, in which the complete training set is splitted and a new group of examples is only added to the current training set when classification performance on the latter is satisfactory (e.g. 95% of correctness).

Despite the simplicity of the approach, the results reported in [WyNi:90] are quite impressive: a 99% generalization performance was achieved by networks using only symbol pairs and triples for two test CFGs (of around 10 rules), after learning a training set of around 2,000 positive and 2,000 negative strings, where recursion was limited in the positive examples and negative examples were generated randomly; however, the classification performance was degraded by using near-misses instead of random negative strings (ranging from 79% for one-error negative strings to 93% for four-errors negative strings) . Altogether, HODYNE infers CFL recognizers by learning the tuples of consecutive symbols which are most commonly and strongly indicative of both legal and illegal strings. Hence, HODYNE operates in a way similar to the large-scale statistical speech recognition systems based on *n-grams*, with the main difference being that in the latter the statistical information is stored as a table of frequencies, while in the former it is indirectly represented in the weights of the net.

A different strategy was reported by Williams and Zipser [WiZi:89]. They coupled a first-order SLRNN to a potentially infinite memory tape to emulate a Turing machine and to learn an FSA controller for the balanced-parentheses CFG. Actually, the SLRNN was trained to be the correct finite-state controller of a given Turing machine by supervising the input-output pairs, where the input is the tape reading from a target Turing machine and the output is the desired action of the controller. Hence, the behavior of the target controller was known a priori and not learned by the net, and therefore, it is not properly a case of grammatical inference.

The possible connectionist representations of a stack, a device which is intimately related to CFL generation and recognition, have also been studied:

Pollack proposed an internal neural network emulator of a stack memory, the *recursive auto-associative memory* (RAAM) model, as a plausible model for cognitive processing [Poll:90]. The RAAM model consists of a 3-layer feed-forward network trained by back-propagation, in which the input and output layers contain two sets of units, one to represent the stack and one to represent a symbol (the top), and the hidden units encode the stack representations. A coding process is performed by the hidden layer to emulate a push action, while a decoding process is performed by the output layer to emulate a pop action. At each step of a sequence of push actions, the contents of the hidden layer is copied into the stack input units. During training, the output units must reproduce the contents of the input units. For a stack with limited length, this model is equivalent to training an FSA, but theoretically, the stack may have an infinite length. Even for a limited length stack, the RAAM model is inefficient, since to be able to represent all the possible configurations of a stack with length $L$ and $m$ alphabet symbols, an FSA with $m^L$ states must be built or learned by the network.

Siegelmann and Sontag presented a connectionist Turing machine model, and they showed that a stack can be simulated in terms of binary representations of a fractional number which are manipulated by neural network generated actions [SiSo:91, SiSo:92]. The focus of their work was on representational issues and not on a practical learning system. Indeed, this type of stack representation is difficult to handle by a learning algorithm based on continuous space optimization, because, although a fractional number is continuous, any small perturbation of the fraction causes a discrete change of the stack contents the fraction is representing.

In the following subsections, two of the most outstanding connectionist approaches to context-free GI, the neural network pushdown automata (NNPDA) model proposed by Sun *et al.* [SuCG:90,SuGC:93] and the CFG production rule learning architecture proposed by Das and Mozer [DaMo:93,MoDa:93], are described in detail, together with a summary of the results reported about their application to CFGI from positive and negative raw examples.

### 4.4.1 The neural network pushdown automaton (NNPDA)

The NNPDA model is sketched in Fig. 4.7. It consists of two major components: a high-order SLRNN controller and an external *continuous* stack memory. At each time step, the network carries out an input-output mapping. Three input vectors are given to the SLRNN: the current internal state $\mathbf{S}^t$, the input symbol representation $\mathbf{I}^t$ and the stack reading $\mathbf{R}^t$. The outputs obtained from the network are the next-time internal state $\mathbf{S}^{t+1}$ and stack action $\mathbf{A}^{t+1}$. This action is performed on the external stack, which in turn renews the next stack reading. The weights of the SLRNN controller are trained

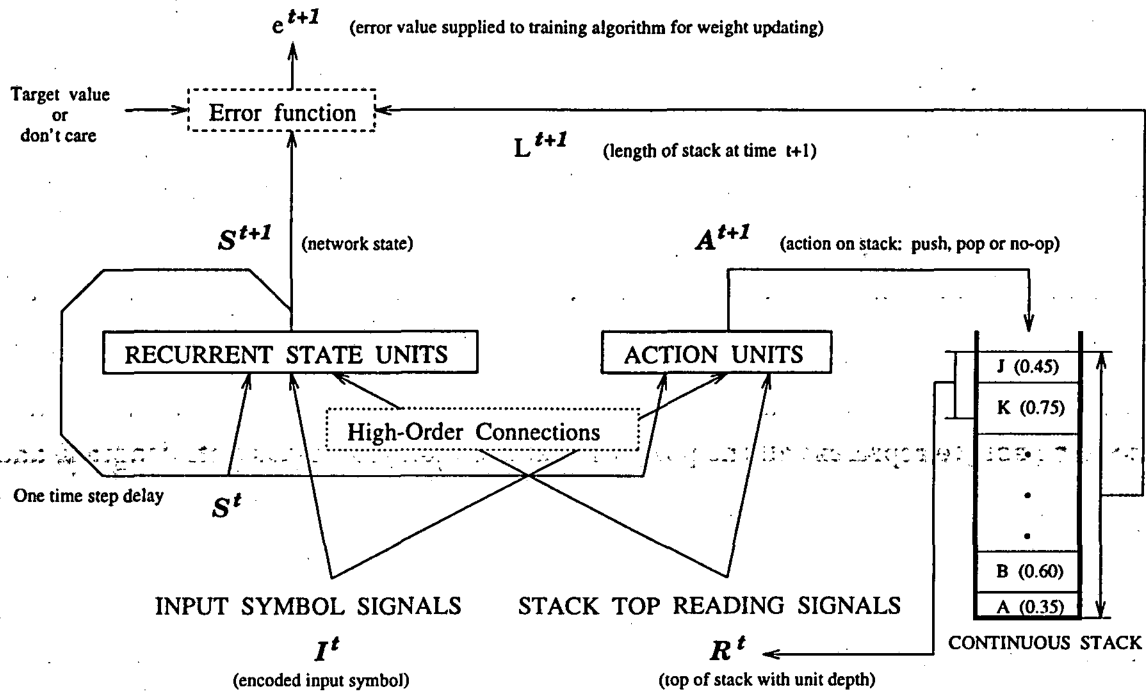


**Fig. 4.7** *The NNPDA architecture: a high-order SLRNN coupled with an external continuous stack.*

by minimizing an error function, which depends on the final state and the stack length at the end of a string presentation.

The following assumptions are made in the model, that restrict the class of CFLs the NNPDA can learn and recognize:

a) only *deterministic* pushdown automata are considered;
b) the same set of symbols is used both for the input and stack alphabets;
c) only one action unit is used, that can represent three operations: a *push* of the current input symbol onto the stack; a *pop* of the current stack; and *no-operation (no-op)* on the stack.

Hence, only a certain class of deterministic CFGs may be inferred by the NNPDA model. To use the NNPDA as a classifier, input strings are fed into the NNPDA one symbol at a time, and the contents of both a special state unit (output unit) and the stack at the end of the string presentation, after processing an end-of-string symbol, will determine whether the input string is accepted or not. Thus, the task to be learned is almost identical to the string classification task described in 4.2.2, except that, now, the controlled external stack is also involved in the classification decision.

#### 4.4.1.1 NNPDA architecture and learning algorithm

The neural network controller in the NNPDA is an extension of the second-order SLRNN used by Giles *et al.* [GiMC:92]. In fact, Sun *et al.* described three different extended versions that can be used for the NNPDA architecture [SuGC:93]:

1) a "natural" third-order SLRNN;
2) a variation of the second-order SLRNN; and
3) a so-called "full-order" network, with third-order connections for the state units and a high-order computation for the action unit.

Let the high-order RNN controller have a single-layer of $N$ recurrent *state units* $U_1, U_2, ..., U_N$, whose activation values are labelled $y_1, y_2, ..., y_N$, and one non-recurrent *action unit*[18] $A_1$ with an activation value denoted $a_1$. Let the RNN also have $M$ *input symbol signals*, which are labelled $x_1, x_2, ..., x_M$, and $M$ *stack top reading signals*, denoted $r_1, r_2, ..., r_M$. It must be emphasized that the same number of signals is used for both the input symbol and the stack reading. This is due to the fact that the same alphabet is assumed for both sets of symbols and the same local representation of the symbols is required (as explained later).

Let $\mathbf{S}^t = [y_1(t-1), y_2(t-1), ..., y_N(t-1)]^T$, $\mathbf{I}^t = [x_1(t), x_2(t), ..., x_M(t)]^T$, and $\mathbf{R}^t = [r_1(t), r_2(t), ..., r_M(t)]^T$ be the internal state, input, and stack reading vectors at time step $t$, respectively, and let $a_1(t)$ be the activation value of the action unit computed from the previous vectors. Some number $P$ of the state units ($1 \leq P \leq N$), called the output units, can be used to supply an output vector $\mathbf{O}^t = [y_1(t), y_2(t), ..., y_P(t)]^T$ at every time step $t$; for the CFGI-oriented string classification task, $P = 1$, and $o_1(t) = y_1(t)$ is the only output unit value at $t$.

The discrete-time dynamics of the high-order RNN controller can be written in general form as

$$y_k(t) = F_S\left(\mathbf{W}_k^y, \mathbf{S}^t, \mathbf{I}^t, \mathbf{R}^t\right) \quad \text{for } 1 \leq k \leq N, \qquad (4.99)$$

$$a_1(t) = F_A\left(\mathbf{W}_1^a, \mathbf{S}^t, \mathbf{I}^t, \mathbf{R}^t\right) \qquad (4.100)$$

where $\mathbf{W}_k^y$ is the vector of weights of the state unit $U_k$, $\mathbf{W}_1^a$ is the vector of weights of the action unit $A_1$, and $F_S$, $F_A$ refer to high-order nonlinear functions associated with the state transition and action mappings, respectively. Note that a third mapping is needed for the stack reading $\mathbf{R}^t$, which depends on the entire history of input symbols

[18] More action units could be used if desired to represent the different operations allowed on the continuous stack, but one action unit is enough [SuGC:93].

and actions, but this mapping is determined by the definition of the stack mechanism and is considered external to the RNN controller.

The most natural option for the high-order RNN controller is given by third-order SLRNNs driven by equations:

$$y_k(t) = g\left(\theta_k^y + \sum_{i=1}^{N}\sum_{j=1}^{M}\sum_{l=1}^{M} w_{kijl}^y\, y_i(t-1)\, x_j(t)\, r_l(t)\right) \quad \text{for } 1 \le i \le N, \tag{4.101}$$

$$a_1(t) = 2\,g\left(\theta_1^a + \sum_{i=1}^{N}\sum_{j=1}^{M}\sum_{l=1}^{M} w_{1ijl}^a\, y_i(t-1)\, x_j(t)\, r_l(t)\right) - 1 \tag{4.102}$$

where $g$ is typically the sigmoid function defined in Eq.(4.5) with range $[0,1]$. Using unary local representations for $\mathbf{S}^t$, $\mathbf{I}^t$, and $\mathbf{R}^t$, a $[1,0,-1]$ encoding of the [push, no-op,pop] operations in the action unit, and a proper discretization of the activation function $g$, any given deterministic PDA could be implemented by a third-order SLRNN of the above type, by setting the weights of the network according to the following rules [SuGC:93]:

i) for every transition defined from a state $q_i$ to a state $q_k$ through input symbol $c_j$ and stack top symbol $z_l$, let $w_{kijl}^y = 1$ and $w_{hijl}^y = 0$, for $h \neq k$;

ii) for every push, no-op, or pop operation to be performed upon reading input symbol $c_j$ and stack top symbol $z_l$ in state $q_i$, let $w_{1ijl}^a = 1,\ 0,\ -1$, respectively.

However, to simplify the network and learning algorithm requirements, second-order SLRNNs can be used in practice to infer some deterministic PDAs [SuGC:93]. In this case, the network dynamics is given by

$$y_k(t) = g\left(\theta_k^y + \sum_{i=1}^{N}\sum_{j=1}^{2M} w_{kij}^y\, y_i(t-1)\, x'_j(t)\right) \quad \text{for } 1 \le i \le N, \tag{4.103}$$

$$a_1(t) = 2\,g\left(\theta_1^a + \sum_{i=1}^{N}\sum_{j=1}^{2M} w_{1ij}^a\, y_i(t-1)\, x'_j(t)\right) - 1 \tag{4.104}$$

where

$$x'_j(t) = \begin{cases} x_j(t) & \text{if } j \le M \\ r_{j-M}(t) & \text{if } j > M \end{cases} \tag{4.105}$$

thus concatenating the two vectors $\mathbf{I}^t$ and $\mathbf{R}^t$ in a single input vector $\mathbf{IR}^t$. Das *et al.* [DaGS:92] reported some experimental comparisons between third-order and second-order based NNPDAs in which the third-order nets gave better learning results.

A third type of SLRNN controller was proposed and used by Sun *et al.* to learn difficult CFGs, such as the palindrome grammar [SuGC:93]. This so-called "full-order" network combined the third-order state dynamics of Eq.(4.101) with a high-order calculation for the action unit, given by the equation

$$a_1(t) = 2\, g\left(\theta_1^a + \sum_{i=0}^{2^N-1} \sum_{j=1}^{M} \sum_{l=1}^{M} w_{1ijl}^a \, y_i'(t-1)\, x_j(t)\, r_l(t)\right) - 1 \qquad (4.106)$$

where, representing the subscript $i$ as an $N$-bit binary number $i_{N-1}...i_m...i_0$, $(i_m \in \{0,1\},\ m = 0,..,N-1)$, the values $y_i'(t-1)$, $(i = 0,1,...,2^N-1)$, are computed as $N$-th order products of the $\mathbf{S}^t$ components as follows:

$$y_i'(t-1) = \prod_{m=0}^{N-1} (i_m y_{m+1}(t-1) + (1-i_m)(1-y_{m+1}(t-1))) \qquad (4.107)$$

It is obvious that full-order networks can only be built with a small number of state neurons $N$, since the number of weights is proportional to the exponential factor $2^N$.

Concerning the external stack memory, a *continuous* stack is needed if the stack memory is wanted to be manipulated by a gradient-descent learning algorithm. This means to make the stack variables a continuous function of the network weights, so that an infinitesimal change of weights will cause an infinitesimal change of the action value, which in turn causes an infinitesimal change of the stack reading. In this way, the stack variable can be included in the error function to be minimized, and the error gradient with respect to the weights can be computed.

In a discrete stack, the *pop* action simply removes the top symbol and the *push* action places the symbol read from input string onto the top of the stack. In the continuous stack, "continuous" symbols are stored, each being a normal symbol but with an associated length $0 < L \leq 1$, and the discrete *pop* and *push* actions are replaced by continuous actions. The value of the action unit $a_1(t)$, which is always within the interval $[-1,1]$, is interpreted as the intensity of the actions to be taken on the stack and affects the length of the symbols to be pushed or popped. Given a small number $\epsilon$ close to zero, a *push* is performed if $a_1(t) > \epsilon$, and a *pop* is performed if $a_1(t) < -\epsilon$; otherwise, a *no-op* takes place. The magnitude $|a_1(t)|$ determines the length of the input symbol to be pushed or the length of the top segment to be popped (maybe affecting one or more consecutive symbols). Other continuous action representations may be devised, e.g. with two action units, for *pop* and *push* operations respectively, an additional *replace* operation could be performed [SuGC:93].

In a discrete stack, a read operation only reads one symbol from the top of the stack and sees nothing below. This reading method is not suitable for the continuous stack, since it can lead to discontinuities in the contents of the stack reading vector

(i.e. a small change in the action value may cause a discrete jump in stack reading). Hence, to avoid these situations, the continuous stack is read with unit depth ($L = 1$) from the stack top. In this way, a continuous reading function can be constructed with respect to the network weights (although this fact alone does not guarantee the differentiability of the reading function), and a probabilistic interpretation of the stack top reading can be made, where symbol probabilities are given by the total length of each symbol in the stack top, and reading with $L = 1$ ensures the required probability normalization. When the stack length is less than 1, the difference can be interpreted as the probability to read the empty stack.

From the above discussion, the proper neural representation of the stack top reading, given by vector $\mathbf{R}^t$, can be established. Fistly, it must be fully compatible with the input symbol representation provided by $\mathbf{I}^t$, since the same set of discrete symbols must be encoded, and in the discrete limit, the learned NNPDA is required to behave as a conventional PDA. Secondly, during training, the stack reading should be able to represent the symbol probabilities related to the contents of the stack top segment of length 1. In addition, a one-to-one mapping between each vector $\mathbf{R}^t$ and the stack top symbol composition it represents is needed. For the first requirement, a unary local representation using $M$ signals (one for each symbol) is selected to encode the discrete symbols for both $\mathbf{R}^t$ and $\mathbf{I}^t$, while for the second requirement, a linear combination of the vectors corresponding to the discrete symbols appearing in the stack top, weighted by the above probabilities, is written in the stack reading vector $\mathbf{R}^t$.

However, it must be remarked that this neural representation of the stack top proposed by Sun *et al.* [SuGC:93] has one drawback: it is not sensitive to the order among the stack top symbols, and hence, is somewhat ambiguous. Also note that, during the operation of the NNPDA, the contents of the entire stack is conventionally stored by means of two arrays, one to store the symbols and one for their lengths, plus an integer variable that counts the total number of symbol occurrences inside the stack (i.e. the dimension of the two arrays). Thus, the stack top reading signals $r_i(t)$, $1 \leq i \leq M$, can be easily updated from this information and the last value of the action unit $a_1(t-1)$. The initial values of $a_1(0)$ and $r_i(0)$, $1 \leq i \leq M$, are zero, whereas the initial state of the network $\mathbf{S}^1 = [y_1(0), ..., y_N(0)]$ may be an arbitrary constant vector in $[0,1]^N$.

Finally, let $L(t)$ denote the stack length at time $t$. This length can be evaluated recursively through the simple equation

$$L(t) \;=\; L(t-1) + a_1(t), \tag{4.108}$$

because only the push or pop actions can change the stack length. Initially, $L(0) = 0$, and at every time step $t \geq 0$, the constraint $L(t) \geq 0$ should be imposed. Consequently,

whenever a "pop empty stack" occurs (detected by $L(t) < 0$) the current input sequence is interrupted and the input string being presented is considered illegal.

Up to here, the architecture and operation mode of the NNPDA have been described. Now, for training the NNPDA, a suitable learning algorithm is needed. A possible learning algorithm is derived by minimizing an error function using a gradient-descent optimization method. In general, the error function $E^s_{total}$ to be minimized will be the sum over an input sequence $s$ (or over a training set including several sequences) of an error function at each time step $E(t)$. The function $E(t)$ will be defined as a scalar error measure which depends on both the output unit(s) value(s) and the stack length at time $t$. In the next subsection, the specific objective function to be used for CFGI by training the NNPDA to learn the string classification task is discussed. For the time being, just assume that weight updating is carried out by means of the common gradient rule

$$\Delta \mathbf{W} = -\alpha \nabla_{\mathbf{W}} E^s_{total} \tag{4.109}$$

and that $\nabla_{\mathbf{W}} E^s_{total}$ is just the sum of the gradients $\nabla_{\mathbf{W}} E(t)$ for all $t$ in the sequence $s$. Furthermore, assume that $E(t)$ is a simple differentiable function of $L(t)$ and $y_i(t)$, for $1 \leq i \leq P$. Then, the preceding gradients are readily calculated using the chain rule as far as the derivatives $\partial L(t)/\partial \mathbf{W}$ and $\partial y_i(t)/\partial \mathbf{W}$, $1 \leq i \leq N$, can be computed for every $t$. From Eq.(4.108), it follows that

$$\frac{\partial L(t)}{\partial \mathbf{W}} = \frac{\partial L(t-1)}{\partial \mathbf{W}} + \frac{\partial a_1(t)}{\partial \mathbf{W}}, \tag{4.110}$$

where $\partial L(0)/\partial \mathbf{W} = 0$, and the computation of $\partial a_1(t)/\partial \mathbf{W}$ is required.

As we have seen in Section 4.1, there are two basic ways to perform gradient computations in RNNs, depending on whether the chain rule differentiation is propagated forward (RTRL) or backward in time (BPTT). In principle, both methods can be extended to train the NNPDA. But, since the derivation of the forward propagation algorithm is more direct, Sun *et al.* presented and proposed three extensions of RTRL for training each of the NNPDAs resulting from using a second-order, third-order, or full-order SLRNN, respectively [SuGC:93]. In all the cases, the needed recursions for $\partial a_1(t)/\partial \mathbf{W}$ and $\partial y_i(t)/\partial \mathbf{W}$, $1 \leq i \leq N$, were found by differentiating the controller dynamical equations, but these recursions required to express the derivatives $\partial r_l(t+1)/\partial \mathbf{W}$ in terms of $\partial a_1(t)/\partial \mathbf{W}$, $\partial y_i(t)/\partial \mathbf{W}$ and $\partial r_l(t)/\partial \mathbf{W}$, $1 \leq l \leq M$, $1 \leq i \leq N$.

With this aim, Sun *et al.* [SuGC:93] demonstrated that an approximate recursive relation for the derivatives $\partial r_l(t+1)/\partial \mathbf{W}$ is given by

$$\frac{\partial r_{l'}(t+1)}{\partial w_{ijl}} = \left(\delta_{l' l_1^{t+1}} - \delta_{l' l_2^{t+1}}\right) \frac{\partial a_1(t)}{\partial w_{ijl}} \quad \text{for } 1 \leq l' \leq M, \tag{4.111}$$

where $l_1^{t+1}$ and $l_2^{t+1}$ are the ordinal numbers of the stack reading signals that represent the top and the bottom symbols respectively in the stack top reading $\mathbf{R}^{t+1}$, $\delta_{ij}$ refers to the Kronecker delta, and $w_{ijl}$ is an abbreviated notation for the weights $w^y_{kijl}$ $(1 \leq k \leq N)$ and $w^a_{1ijl}$. It only remains to set the initial conditions $\partial a_1(0)/\partial \mathbf{W} = 0$ and $\partial y_i(0)/\partial \mathbf{W} = 0, \;\; 1 \leq i \leq N$, to obtain forward propagated dynamical systems that provide the partial derivatives required to compute the gradients $\nabla_{\mathbf{W}} E(t)$.

#### 4.4.1.2 Context-free grammatical inference through NNPDAs

In order to infer an NNPDA acting as acceptor of a target language generated by a deterministic CFG, an NNPDA is trained to learn the string classification task for a set of positive and negative sample strings. The approach is quite similar to that described in Section 4.2.2., with the difference that, now, the controlled stack is also involved in the classification decision.

Let $S = (S^+, S^-)$ be a sample of a CFL $L$ over $\Sigma$, which is used as training set. Again, let $\Sigma' = \Sigma \cup \{\$\}$, where $\$$ is a special end-of-string symbol, and let $S_\$$ denote the sample obtained by converting each string $s \in S$ into the form $s\$$. An NNPDA can be trained to classify the strings in $S$ or $S_\$$, so that after convergence, the NNPDA accepts a language $L'$ such that $L' \supseteq S^+$ and $L' \cap S^- = \emptyset$. In some cases, a perfect generalization may be achieved, and the trained NNPDA accepts the target CFL, $L' = L$, by simulating a deterministic pushdown automaton for $L$.

The strings are presented one symbol at a time to the NNPDA through the input symbol signals. Sun *et al.* [SuGC:93] used an end-of-string symbol, thus giving the NNPDA an extra time step to yield the proper response to every input string. However, the use of an end-of-string symbol is not mandatory. As commented earlier, a local encoding must be followed to represent the symbols in both the input and stack reading signals, implying $M = |\Sigma'|$ (if the end-of-string symbol is included). A single state unit is designated as output unit ($P = 1$) to distinguish between final and non-final states. The classification of a given string $s$ by the NNPDA will be determined by examining both the final activation value of this output unit $o_1(t_f(s))$ and the stack length resulting from the last action $L(t_f(s))$, once the whole string has been processed.

For a conventional pushdown automaton, if either the state reached after string presentation is a final state, or the stack is ended empty, the input string is accepted (and otherwise rejected) [HoUl:79]. However, Sun *et al.* claimed that a combination of the two criteria seems necessary to train an NNPDA, and they proposed to define a legal string (for an NNPDA) as one that leads both to a final state *and* an empty stack [SuGC:93]. Hence, for legal strings, the target value of the output (final-state) unit

activation after the string presentation is $o_1(t_f(s)) = 1$ (recall that $o_1(t_f(s)) \in [0,1]$), and the target value of the stack length is $L(t_f(s)) = 0$. On the other hand, the target values of these variables for illegal strings cannot be clearly determined (for example, it is difficult to decide which must be the desired value of stack length for an illegal string). Thus, the target condition selected by Sun *et al.* for illegal strings was that either $o_1(t_f(s)) = 0$ (non-final state) *or* $L(t_f(s)) \geq 1$ (non-empty stack). The corresponding error function to be minimized for both positive and negative strings can be defined as

$$E^s_{total} = E(t_f(s)) = \frac{1}{2} (\nu + L(t_f(s)) - o_1(t_f(s)) )^2, \tag{4.112}$$

where $\nu$ is a parameter assigned as global target value for each training example, such that $\nu = 1$ for positive examples and $\nu = \min\{0, o_1(t_f(s)) - L(t_f(s))\}$ for negative examples. By definition, the error measure $E(t)$ is zero for all the previous instants from the beginning of the presentation of string $s$.

Therefore, in order to learn the string classification task, the NNPDA is trained using a set of pairs $\{(s,\nu) \mid s \in S, \ \nu \in \Re^- \cup \{0,1\}\}$, that includes all the examples in the given sample, where $\nu$ is determined on-line for the strings in $S^-$ at the end of each string presentation. Using the gradient-descent learning algorithm aforementioned from the above error function, we obtain from Eqs.(4.109) and (4.112) that the NNPDA weights are updated at the end of each string[19] according to

$$\Delta \mathbf{W} = -\alpha \, (\nu + L(t_f(s)) - o_1(t_f(s)) \left( \frac{\partial L(t_f(s))}{\partial \mathbf{W}} - \frac{\partial o_1(t_f(s))}{\partial \mathbf{W}} \right) \tag{4.113}$$

where the partial derivatives of $L(t_f(s))$ and $o_1(t_f(s))$ with respect to the weight matrix $\mathbf{W}$ can be calculated recursively, as explained in the previous subsection (recall that $o_1(t_f(s))$ is actually a state unit activation, e.g. $o_1(t_f(s)) = y_1(t_f(s))$). However, whenever a negative stack length appears ($L(t) < 0$), the presentation of the current string $s$ is stopped and some heuristic weight correction rule is applied: the weights will be modified to increase the stack length $L(t)$ for positive strings ($\Delta \mathbf{W} \sim \partial L(t)/\partial \mathbf{W}$) and will be unchanged or even modified to reduce the stack length for negative strings ($\Delta \mathbf{W} \sim -\partial L(t)/\partial \mathbf{W}$).

For each presented string, the expression $H \equiv o_1(t_f(s)) - L(t_f(s))$ can be considered as a measure of how well both of the two conditions "final state" and "empty stack" are satisfied. Thus, $H = 1$ and $H \leq 0$ is desired for positive and negative strings, respectively. After training, the same measure $H$ can be used to test the generalization performance of the NNPDA on unseen input strings. A given string

[19] Batch weight updating might also be used.

$s$ will be accepted if $H > 1 - \epsilon$ and it will be rejected if $H \leq \epsilon$ (or a "pop empty stack" action is performed previously during its presentation), where $\epsilon$ is a tolerance threshold such that $0 < \epsilon \leq 0.5$. Sun *et al.* used $\epsilon = 0.5$, the maximum tolerance, in their experiments [SuGC:93]. The correct classification of the whole training set can be used as stop criterion for the training stage, but some maximum number of training epochs should be fixed a priori for the case that the NNPDA fails to converge.

#### *4.4.1.3 Extraction of pushdown automata from trained NNPDAs*

Although a trained NNPDA could generalize perfectly in some cases, classification errors will normally occur for strings longer than those included in the training set due to an unstable behavior of the analog NNPDA. Therefore, to improve the generalization performance, the extraction of a discrete PDA from a trained NNPDA is desirable. Sun *et al.* proposed the following quantization procedure to carry out this PDA extraction process [SuGC:93]:

First, the action unit is quantized into three discrete values $(-1, 0, 1)$ according to

$$a_1(t) = \begin{cases} 0 & \text{if } |a_1(t)| \leq A \\ -1 & \text{if } a_1(t) < -A \\ 1 & \text{if } a_1(t) > A \end{cases} \tag{4.114}$$

where the threshold $A = 0.5$ was recommended. In this way, discrete *no-op*, *pop* and *push* actions are generated, which make the continuous stack behave as a discrete conventional stack.

Second, a cluster analysis of the internal states is performed. All the input strings that have been recognized correctly are fed into the trained NNPDA and a set of internal state vectors (points in $[0, 1]^N$) is obtained. This set is partitioned into several clusters using the $k$-means clustering algorithm [DeKi:82], where the number of clusters $k$ is determined by minimizing the average distance from each state to its cluster center. After clustering, the cluster centers are stored as the representative points of the quantized internal states, which correspond to the $k$ states of the discrete PDA being built. During further testing, each analog internal state is quantized to its nearest cluster center, and the state transition rules of the PDA can be extracted.

A regular quantization method, similar to that by Giles and Omlin [GiOm:93] described in Section 4.3, which divides the activation range of each state unit into $q$ intervals of equal size, was also proposed as an alternative to the above method for state clustering and extraction [SuGC:93].

Likewise, to simplify the state structure of the extracted PDA, a minimization procedure was described. Even though it is well-known that there exists no minimization algorithm for obtaining a unique minimal PDA, nor for finding whether two PDAs (or CFGs) are equivalent [HoUl:79], a reduction algorithm can be applied for deterministic PDAs in which the input and stack symbols are the same and only one symbol is involved in the push and pop operations. For this type of PDA, each state transition between any two states can be characterized by a three-tuple $(\alpha, \beta, \gamma)$, where $\alpha$ is the input symbol, $\beta$ is the stack reading symbol, and $\gamma = 1, -1, 0$ represents push, pop and no-op actions. By considering each combination of $(\alpha, \beta, \gamma)$ as an input symbol of a regular grammar, an equivalent FSA can be built from the extracted PDA, where state transitions are caused by the $(\alpha, \beta, \gamma)$ "symbols". Hence, by applying an FSA minimization algorithm to this FSA, the extracted PDA can be effectively reduced.

#### *4.4.1.4 Some experimental CFGI results using NNPDAs*

Sun *et al.* reported some experiments of CFGI from positive and negative examples using NNPDAs [SuGC:93]. Three simple deterministic CFGs were tested:

1) the balanced parenthesis grammar over $\Sigma = \{'(', ')'\}$;
2) the $1^n0^n$ grammar;
3) the deterministic palindrome grammar generating $\{ \ xcx^R \mid x \in (a+b)^* \ \}$.

Different types of NNPDA and training procedures were used for each particular problem.

1) **Balanced parenthesis grammar**

Second-order NNPDAs with $N = 3$ state units were trained to recognize sequences of balanced parentheses. An end-of-string symbol was used, and a unary input representation was followed with $M = 3$. The training set consisted of 50 strings: all 30 strings up to length 4 and 20 randomly selected longer strings up to length 8. Five runs were performed, in which the NNPDAs required approximately 100 training epochs to learn the training set. During the generalization tests, all the strings up to length 20 could be correctly recognized, but due to analog error accumulation, some longer strings were incorrectly classified. However, the target PDA for this grammar could be extracted from the trained NNPDA using the regular partition search method [GiOm:93] with quantization level $q = 5$.

### 2) $1^n0^n$ grammar

The language of the $1^n0^n$ CFG is a subset of the language generated by the balanced parenthesis CFG. A second-order NNPDA with $N = 5$ state units was trained to recognize this language. The end-of-string symbol was used again. A small training set containing 27 short strings (12 positive and 15 negative strings) was initially used for training. After 100 training epochs, the NNPDA correctly classified the training set and only failed to classify 6 strings among all the strings up to length 8. These incorrectly classified strings were added to the training set and the NNPDA was retrained for another 100 epochs. After performing 5 such cycles of test-retrain steps on strings of gradually increased length, the trained NNPDA correctly classified all the strings up to length 20 and 20 randomly chosen strings up to length 160. A correct PDA for the grammar was extracted from the trained NNPDA using the regular partition search method with $q = 2$, and this extracted PDA could be simplified to an equivalent PDA with 4 states through the reduction algorithm described in the last subsection.

### 3) Deterministic palindrome grammar

The language generated by the deterministic palindrome CFG is $\{ xcx^R \mid x \in (a+b)^* \}$, where $x^R$ is the reversed order form of $x$, and $c$ is a symbol used to mark the boundary between $x$ and $x^R$. The minimal PDA for this CFL contains 3 states: one to push the symbols read before the $c$, another to pop matched symbols after seeing the $c$, and a third, "trap state", which is reached whenever after reading the $c$ the input and stack top symbols do not match. The input string is legal only if the PDA ends at the second state with empty stack.

Neither second-order nor third-order NNPDAs were able to learn a correct PDA for the deterministic palindrome CFG [DaGS:92, SuGC:93]. Two major difficulties were detected. First, an insufficient information to supervise the stack actions for the illegal strings. Second, some structural limitations of the second- and third-order NNPDAs to implement certain PDAs when a unary or mutually orthogonal state representation is not used, as it is typically the case when the internal states evolve freely during learning.

To solve the first problem, heuristic hints (requiring some a-priori knowledge of the target CFG) were introduced in the objective function to be minimized [DaGS:93], in order to discriminate the "trap state" and to supervise somehow the stack length. To overcome the second problem, Sun *et al.* [SuGC:93] proposed a variation of the "full-order" network described previously, in which a linear activation function is used in the action unit, i.e. the equation that describes the dynamics of the action unit is

in this case

$$a_1(t) = \sum_{i=0}^{2^N-1} \sum_{j=1}^{M} \sum_{l=1}^{M} w^a_{1ijl}\, y'_i(t-1)\, x_j(t)\, r_l(t) \qquad (4.115)$$

instead of Eq.(4.106), where still the $y'_i(t-1)$ values are given by Eq.(4.107). To guarantee that the action unit value $a_1(t)$ is in the desired range $[-1, 1]$, the action unit weights $w^a_{1ijl}$ are truncated to the range $[-1, 1]$; moreover, by quantizing each weight of the action unit to three levels $(-1, 0, 1)$ after learning, each weight can represent an action rule (note that this kind of encoding may also be used to insert prior knowledge before training).

For the palindrome CFG, the error function that was minimized during training was distinct from the one described by the general case given by Eq.(4.112). Instead of using the output unit to discriminate between final and non-final states, the output unit was trained to discriminate between "trap state" (definitely illegal string) and "non-trap states" (potentially legal string). To implement this hint, whenever a prefix of the input string is read that leads to the trap state, a target value $T = 0$ is supplied for the output unit and the input string is interrupted; otherwise, for potentially legal strings, the weight correction is made at the end of the string and the target value for the output unit is $T = 1$. Thus, the weight correction could be written in this case as

$$\Delta \mathbf{W} = \alpha \left[ (T - o_1(t)) \frac{\partial o_1(t)}{\partial \mathbf{W}} + (L - L(t)) \frac{\partial L(t)}{\partial \mathbf{W}} \right] \qquad (4.116)$$

where $T$ is one of the above target values of "trap state" and "non-trap state", and $L$ is the target value of stack length, which is defined as $L = 0$ for legal strings, and it is adapted on-line for illegal strings to increase the stack length according to the rule

$$L = \begin{cases} L(t) + 0.1 & \text{if } L(t) \geq 0.9 \\ 1 & \text{if } L(t) < 0.9 \end{cases} \qquad (4.117)$$

with the aim of avoiding the confusion with the legal strings.

Another a-priori knowledge hint was used to initialize and fix some weights during training. These weights were determined to force a push action whenever a special "empty stack" symbol is seen in the stack top reading (except for the particular case of the legal string "$c$").

The full-order network trained to learn the palindrome CFG contained $N = 4$ state neurons, $M = 3$ input symbol signals (for $a$, $b$ and $c$, since no end-of-string symbol was used) and $M+1$ stack top reading signals (to include a special "empty stack" symbol). Two training sets were used: the former included all the strings up to length 3, and the latter included all the strings up to length 5; in addition, a few more positive strings were added to each set trying to balance the weight correction forces of the positive

and negative strings. The first training set was used to train the NNPDA for 200 epochs, and then the second training set was used for another 200 epochs. During the generalization tests after learning, every input string $s$ was entirely presented (no "trap state" interruptions) and accepted by the NNPDA if $(o_1(t_f(s)) > 0.5) \wedge (L(t_f(s)) \leq 0.5)$ (otherwise rejected). All the 29,523 strings up to length 9 were used as test set, and only 4 classification errors were made by the trained NNPDA [SuGC:93].

A quantization of the trained NNPDA was also carried out by substituting the sigmoid activation function by a step function in the state units, and by quantizing each of the action unit weights to three levels $(-1, 0, 1)$ using $-0.5$ and $0.5$ as threshold values. This last quantization can be shown to be equivalent in discrete behavior to the quantization of the action unit activation value. All the 21,523,359 strings up to length 15 were correctly classified by the quantized NNPDA, using the previous classification rule, except that as soon as $o_1(t)$ becomes zero the input string is rejected. Finally, a discrete PDA was extracted from the quantized NNPDA which perfectly recognized the deterministic palindrome CFG. This PDA could be further reduced to an equivalent PDA with seven states (one of them the "trap state")[20].

The preceding experiments revealed some interesting aspects concerning NNPDA operation and training for CFGI. Namely,

- the lack of supervision knowledge for the stack length of illegal strings is a problem in training NNPDAs, which can be alleviated by enforcing a-priori knowledge in the training stage (through the error function and/or initially fixed weights);
- the introduction of the "full order" connection and its linear formulation for the stack action unit augments the learning power of the NNPDA (at the expense of a number of weights exponential in $N$) and provides a way to insert discrete action rules into a continuous NNPDA trained by gradient-descent;
- although some simple deterministic CFGs could be inferred by the NNPDA approach, Sun *et al.* acknowledged that it is not clear whether their connectionist approach is an efficient way to learn CFGs or how well it can perform for larger and more complex CFGs [SuGC:93].

[20] Remember that the minimal PDA for the target grammar contains only 3 states.

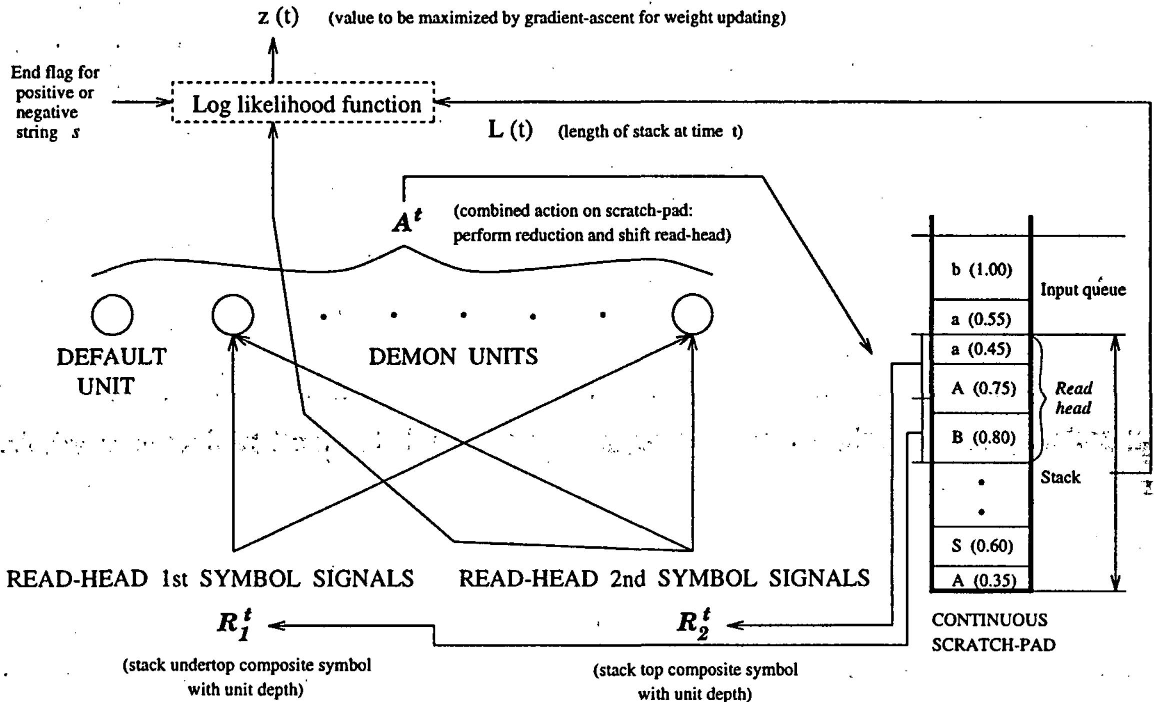


**Fig. 4.8** *The architecture proposed by Das and Mozer to induce CFG productions.*

### 4.4.2 A connectionist symbol manipulator that induces CFG production rules

Das and Mozer [DaMo:93, MoDa:93] presented a connectionist symbol manipulator that can be trained to induce CFG productions (or rewrite rules). Whereas in the approach by Sun *et al.* [SuGC:93] just described, the primary interest was to learn the dynamics of a deterministic pushdown automaton recognizing an unknown target CFG, the approach by Das and Mozer focused on learning the explicit production rules of a target CFG to parse correctly the strings generated by the grammar. In this case, however, the system requires some prior knowledge about the target CFG $G = (V_N, V_T, P, S)$: the number of non-terminal symbols $|V_N|$ and the maximum number of rules $K$ that have the same left-hand side must be specified in advance. Moreover, the target CFG $G$ is assumed to be in a normal form such that all the production rules in $P$ reduce two symbols to a nonterminal.

The architecture proposed by Das and Mozer [DaMo:93] is depicted in Fig. 4.8. It consists of a single layer network controller (but not an SLRNN) coupled with an external *continuous* scratch-pad memory. The scratch pad is implemented as a

combination of a continuous stack, which is similar to the one in the NNPDA model except in the fact that two composite symbols are readable simultaneously from the stack top, and an input queue. The purpose of the scratch pad is to hold the transitional stages of an input string during a reduction process, that leads to the start symbol $S$ of the grammar when the string is recognized by bottom-up LR parsing.

Before a string is presented, the continuous stack is reset to contain only a single special symbol, the null symbol \$, with an "infinite" length. Then the string is placed in the input queue, and the network is allowed to run for $2l - 2$ time steps, which is exactly the number of steps required to parse any grammatical string of length $l$ (due to the implicit restrictions on the production rules). At each time step, a *read-head* determines the location on the scratch pad memory of two unit-length symbols to be supplied to the network, which correspond to the stack top and undertop symbols. These (possibly composite) symbols are represented in two vectors $\mathbf{R_1}^t$ and $\mathbf{R_2}^t$, for the stack undertop and top respectively, and introduced to the network by means of two sets of $M$ input signals each.

The network consists of a single-layer of $N$ *demon units*, each one receiving the stack reading signals, plus one special *default unit.* Each demon unit is explicitly associated with a particular nonterminal $B \in V_N$, but several demons may be associated with the same nonterminal, each representing a production rule with $B$ in the left hand side. The weights of a given demon unit represent a pattern of two symbols, which correspond to the right hand side of the rule. When a demon unit matches its pattern to the read-head symbols, it fires (maybe partially depending on the matching degree) and produces a (maybe partial) replacement of the symbols under the read-head by the nonterminal associated with that demon. This action corresponds to a string reduction in accordance with the demon's production rule. When none of the demon units fire, the default unit fires and yields a shift of the read-head from left to right. i.e. the first symbol in the input queue is pushed onto the stack. Actually, both the default unit activation and the corresponding push action are not necessarily discrete but continuous (or partial). Hence, the vector of activations of the default and demon units at time $t$, denoted $\mathbf{A}^t$, determines the overall (maybe combined) action that is performed on the scracth pad, which in turn renews the symbols under the read-head.

If a grammatical string is reduced correctly, the final contents of the scratch pad will be an empty queue and a stack of unit total length just containing the start symbol $S$; in practice, $S$ must be the top symbol coded in $\mathbf{R_2}^t$, while the null symbol \$ must be coded in $\mathbf{R_1}^t$. The length of the null symbol is not computed in the stack total length. Therefore, the weights of the network controller can be trained by maximizing an objective function, which depends on both the stack top symbol and total length at the end of a string processing, and also on whether the supplied example string is positive or negative.

The described model, displayed in Fig. 4.8, was actually designed to learn and parse LR(0) grammars, a subclass of CFGs, but Das and Mozer pointed that the architecture can be easily extended to LR($n$) CFGs by including connections from the first $n$ composite symbols in the input queue to the demon units. On the other hand, the neural representation of symbols and production rules in the network presents the following features [DaMo:93]:

- the number of signals $M$ required to represent a symbol is given by $M = |V_T| + |V_N| = |V|$, i.e. the total number of terminal and nonterminal symbols in the target grammar (which must be known a priori);
- the terminal and nonterminal discrete symbols, or non-composite symbols of unit length, are jointly represented using a local unary encoding (by means of $M$-dimensional vectors $\mathbf{S}_c = [s_{c1}, ..., s_{cM}]^T$, $1 \leq c \leq |V|$, where $s_{ci} = 1$ if $c = i$, and $s_{ci} = 0$ otherwise, for $1 \leq i \leq M$);
- a composite symbol is represented by an $M$-dimensional vector $\mathbf{R} = [r_1, ..., r_M]^T$ such that $\sum_{i=1}^{M} r_i = 1$ and each component $r_i$ ($1 \leq i \leq M$) expresses the part (length) of the composite symbol that is assigned to the $i$-th discrete symbol;
- a lower bound on the number of demon units the network may have is given by $N = K|V_N|$, although strictly, the minimum required number is $N = |P|$ if the number of production rules for each nonterminal is known;
- the number of demon units $N$ and the fixed identity of each (i.e. its associated nonterminal) must be specified before learning, but an excess of demon units with respect to $|P|$ does not necessarily degrade the performance of the network, since redundant productions may arise in the demon units.

Now let us describe the operation and training of the model more precisely. Let $\mathbf{X}^t = [x_1(t), ..., x_{2M}(t)]^T$ denote the input vector resulting from the concatenation of the read-head symbol vectors $\mathbf{R_1}^t$ and $\mathbf{R_2}^t$. Each demon unit $D_i$ computes the distance between the input vector $\mathbf{X}^t$ and its weight vector $\mathbf{W}_i$:

$$dist_i(t) = b_i \sum_{j=1}^{2M} (w_{ij} - x_j(t))^2 \quad \text{for } 1 \leq i \leq N, \tag{4.118}$$

where $b_i$ is an adjustable bias associated with the unit, whereas the special default unit $D_0$ just contains an adjustable bias $b_0$, which is used to define $dist_0(t) = b_0$. The activation value of unit $D_i$ at time step $t$, denoted $a_i(t)$, is computed *non-locally* according to a normalized exponential transform

$$a_i(t) = \frac{e^{-dist_i(t)}}{\sum_{k=0}^{N} e^{-dist_k(t)}} \quad \text{for } 0 \leq i \leq N, \tag{4.119}$$

which enforces a competition among the units. Note that the activation of the default unit, which determines the amount of right shift to be made by the read-head, is computed like that of any other demon unit.

Let $\mathbf{A}^t = [a_0(t), a_1(t), ..., a_N(t)]^T$ denote the action vector that is output by the single-layer network at time step $t$. Reduction of a pair of symbols on the scratch pad corresponds to *popping* the top two symbols from the stack and *pushing* the nonterminal associated with the demon unit that matched the symbols. On the other hand, the left-to-right shift of the read-head, forced by the default unit, is achieved by moving the next symbol from the input queue on the top of the stack. However, since both the default and demon units can be partially active, reduction and shift actions need to be performed partially and simultaneously on the scratch pad. The continuous stack allows this type of combined operation while providing a differentiable function that permits to back propagate error through the stack during learning. At every time step $t$, the action vector $\mathbf{A}^t$ performs two basic operations on the continuous stack [DaMo:93]:

- **pop:** the length of all items on the stack contributing to the top two composite symbols is multiplied by $a_0(t)$; when $a_0(t) = 0$ (meaning that one or more demon units are strongly active) the top two symbols will be popped, and when $a_0(t) = 1$ no pop operation will take place;
- **push:** the symbol pushed onto the stack is the composite symbol given by the linear combination $\sum_{k=0}^{N} a_k(t)\mathbf{S}_k$, where $\mathbf{S}_k$, $1 \leq k \leq N$, is the vectorial representation of the discrete nonterminal associated with demon unit $D_k$, and $\mathbf{S}_0$ is defined to be the first (maybe composite) symbol of unit length in the input queue, where a fraction $a_0(t)$ of the $\mathbf{S}_0$ symbol is removed from the input queue and pushed onto the stack.

The system is trained using a sample $(S^+, S^-)$ of positive and negative example strings from a CFL. The task to be learned is to classify each string in the training set correctly. Let us define

$$p_s^{start} = e^{-||\mathbf{R}_2{}^{t_f(s)} - \mathbf{S}_1||^2} \tag{4.120}$$

as the probability that the stack top contains the start symbol $S$ after presentation of the string $s$, where $\mathbf{S}_1$ is the vectorial unary representation of nonterminal $S$; and let us define

$$p_s^{length} = e^{-c(L(t_f(s))-1)^2} \tag{4.121}$$

as the probability that the stack total length $L(t_f(s))$ after processing $s$ be 1, where $c$ is a constant. Then, the product $p_s = p_s^{start} p_s^{length}$ is a measure of how well the input string $s$ has been parsed by the model. The value of $p_s$ should be close to 1 for positive examples and close to 0 for negative examples. Hence, a string $s$ may be accepted if $p_s > 1 - \epsilon$ and rejected if $p_s < \epsilon$, where $\epsilon$ is a tolerance such that $0 < \epsilon \leq 0.5$.

Furthermore, a likelihood objective function can be defined [DaMo:93] as

$$H = \prod_{s \in S^+} p_s^{start} p_s^{length} \prod_{s \in S^-} (1 - p_s^{start} p_s^{length}) \tag{4.122}$$

and the logarithm of this function can be taken as the objective function to be maximized during training:

$$Z = \log H = \sum_{s \in S^+} \log (p_s^{start} p_s^{length}) + \sum_{s \in S^-} \log (1 - p_s^{start} p_s^{length}). \tag{4.123}$$

The gradient $\nabla_{\mathbf{W}} Z$ is just the sum of the gradients $\nabla_{\mathbf{W}} Z(t_f(s))$ for all $s \in (S^+, S^-)$, where

$$Z(t_f(s)) = \begin{cases} \log (p_s^{start} p_s^{length}) & \text{if } s \in S^+ \\ \log (1 - p_s^{start} p_s^{length}) & \text{if } s \in S^- \end{cases} \tag{4.124}$$

and $Z(t) = 0$ for the rest of time steps during the processing of the training set. The gradients $\nabla_{\mathbf{W}} Z(t_f(s))$ can be computed using an adaptation of the BPTT algorithm, which involves back-propagating through the stack. Then, the weights of the network (including unit biases) can be updated at the end of each string presentation during training using the gradient-ascent rule

$$\triangle \mathbf{W} = \alpha \nabla_{\mathbf{W}} Z(t_f(s)) \tag{4.125}$$

where $\alpha$ is the learning rate.

Das and Mozer employed the connectionist approach described to infer the following simple target CFGs:

1) the balanced parenthesis grammar with $P = \{S \rightarrow ()|(X)|SS,\ X \rightarrow S)\}$
2) the $a^n b^n$ grammar with $P = \{S \rightarrow ab|aX,\ X \rightarrow Sb\}$
3) the postfix grammar with $P = \{S \rightarrow aX|SX,\ X \rightarrow b + |S+\}$
4) the pseudo-nlp grammar with $P = \{S \rightarrow Nv|nV,\ V \rightarrow vn,\ N \rightarrow an\}$

from short positive and negative strings selected by hand [DaMo:93]. Since, for a given grammar, the number of negative examples was much greater than the number of positive examples, the positive strings were repeated in the training set to constitute half of the total training examples. The number of demon units $N$ and the fixed identity of each demon was specified in advance of learning. The initial weights $w_{ij}$ were selected from a uniform distribution over the interval $[0.45, 0.55]$ and the unit biases $b_i$ were initialized to 1. Das and Mozer reported a network generalization performance of 100% for the first three grammars above. Finally, it was noted that the learned weights can be interpreted as right hand sides of symbolic production rules, with one production per demon unit, and therefore, an explicit representation of the CFG can be extracted from the trained network [DaMo:93].

## 4.5 Concluding remarks

In the latest years, recurrent neural networks (RNNs), as well as some other connectionist models, have been investigated as an alternative approach to learn grammars or acceptors from example strings. A variety of architectures (both first-order [Elman:90, SmZi:89, Fahl:91, MaFa:94, SoAl:94] and higher-order [Poll:91, GiMC:92, SuGC:93, DaMo:94]) and related learning algorithms (RTRL [WiZi:89, GiMC:92], BPTT [WiPe:90], time-block based RTRL [Schm:92], ...) have been proposed. Some works have focused on training the network to predict the next input symbol at each step of a positive string sequential presentation (*next-symbol prediction task*) [ClSM:89, SmZi:89, Fahl:91, SoAl:94]. Other works have focused on training the network to respond, after a whole string presentation, whether the input string belongs to a target language or not (*classification task*) [GiMC:92, WaKu:92, MiGi:93, MaFa:94, SuGC:93].

The capability of neural networks to learn some simple regular and context-free grammars (or their corresponding automata) have been shown. However, some problems have appeared altogether: bad generalization for long strings caused by state unstability, possible learning failure due to local minima of the error function and/or an inadequate number of units, difficulty of biasing and controlling the inference process, etc. Moreover, the performance of the reported neural models in learning large grammars is still unknown.

On the other hand, some clustering methods have been proposed to extract symbolic representations (both FSAs [SeCM:88, DaDa:91, GiOm:93, MaFa:94] and PDAs [SuGC:93]) from trained networks with a continuous state space. Likewise, quantization techniques have been applied to use RNNs as discrete machines both during [ZeGS:93, DaMo:94] and after learning [DaDa:91, SuGC:93]. In general, the extracted automata and the quantized networks outperform the trained continuous networks in classifying unseen data (generalization performance). Nevertheless, the conditions that must be met by the network and the training set in order to identify a target grammar or automaton are still not well understood.

In Chapters 6 and 7 some techniques will be proposed to improve the learning performance of RNNs, as well as to extract and insert FSAs in RNNs, which attempt to overcome some of the deficiencies aforementioned about the application of RNNs to grammatical inference (e.g. the difficulty in controlling and biasing the inference process carried out by the network). The FSA extraction and insertion methods that will be proposed also improve and extend the ones reported previously by other researchers.