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 6

Regular grammatical inference using recurrent neural networks

In this chapter, some contributions on the topic of regular grammatical inference (RGI) using recurrent neural networks (RNNs) are reported together with the results of some experimental studies that have been carried out to assess a variety of connectionist approaches to RGI.

The chapter begins with the presentation of two methods to extract UFSAs from previously trained RNNs, one to be used after learning the next-symbol prediction task from a set of positive examples, and another one to to be used after learning to classify a set of positive and negative strings. Both methods are based on the well-known hierarchical clustering algorithm, and they basically differ in the stop criterion. In the first case, there is no difference between extracting UFSAs or FSAs from the net, and a straightforward implementation of the hierarchical clustering algorithm is done, where the only new feature is in the automatic computation of the distance threshold.

The method proposed for the second case (positive and negative examples) is more interesting and novel, since it takes advantage of the UFSA representation and processing (*merge* operations, which are performed along with the clustering of activation patterns) to guarantee the extraction of a consistent deterministic UFSA in a single run after neural learning[1]. Moreover, the automaton returned by the simplest version of the algorithm is the one with minimal size with respect to the consistent

[1]In the previous FSA extraction methods proposed by Das and Das [DaDa:91], Giles and Omlin [GiOm:93], and Manolios and Fanelli [MaFa:94], it may be necessary to repeat the clustering process several times with different parameters until a DFA consistent with the sample data is obtained (see Section 4.3).

DUFAs extractable by hierarchical clustering from the given sample and network. This version was reported in the ICGI'94 colloquium held in Alicante (Spain) [AlSa:94b]. A more generic version of the extraction algorithm is suggested here to reach a UFSA in the deterministic border set $DBS_{PTU}(S)$ (recall Section 5.2.3) associated with the input sample $S = (S^+, S^-)$.

Next, the choice of the activation function to be used in the RNN units for learning sequential tasks is discussed. Some drawbacks of the commonly used sigmoid function are pointed out and some alternative activation functions are proposed: the antisymmetric logarithm for the recurrent hidden units, and a sinusoidal function for the output units and non-recurrent hidden units (if any). Likewise, the effect of different types of activation functions and RNN architectures on the capability of learning the next-symbol prediction task for the two benchmark Reber grammars is reported. This empirical study includes a comparison between training ASLRNNs by a full-gradient or a truncated-gradient learning algorithm[2]. The results obtained using full-gradient learning algorithms when the sigmoid function was substituted are significantly better than those reported in previous works for the same grammars [ClSM:89, SmZi:89, Fahl:91b]. Partial results of these experiments were already reported in [AlSa:94a, SoAl:94].

The last part of the chapter is devoted to RGI from positive and negative examples by training RNNs to learn the string classification task. An experimental study on the generalization performance of the trained RNNs and the UFSAs extracted from them is reported, which includes both first-order and second-order two-layer ASLRNNs. The same test languages and sparse sample sets used as benchmark in [Dupo:94] and Chapter 5 have been employed for the study. In this way, the results yielded by the connectionist approach, with and without UFSA extraction, can be fairly compared with those displayed in the preceding chapter for pure symbolic methods. Even though the UFSA extraction post-processing is demonstrated to improve the generalization performance of the networks, the average quality of the inference using RNNs is not so good as the one shown by some of the symbolic methods tested previously. The results of this experimental study have been partially reported in [AlSS:97] as well as included in a recently submitted paper [AlSa:97b].

[2]Remember from Chapter 4 that when simple back-propagation is applied to train SRNs or other ASLRNNs, as in the earlier works with Elman's model (e.g. [ClSM:89]), only a truncated gradient is actually computed for gradient descent.

6.1 UFSA extraction from trained RNNs for regular grammatical inference

As it has been observed in different studies [ClSM:89, GiMC:92, ZeGS:93] reviewed in Section 4.3, RNNs with continuous activation functions, typically the sigmoid function, develop an internal state representation in form of clusters in the activation space of the recurrent hidden units when they are faced to a sequential learning task. Therefore, by using clustering techniques, a symbolic description of an "approximately simulated" FSA can be obtained from the network dynamics during or after learning. There are at least three reasons why the extraction of a finite automaton from a recurrent network can be wished:

- it helps to understand the learning process of the RNN and to gain an insight into what is the network doing;
- the ultimate aim of the task is precisely to obtain or learn an FSA;
- it serves to counteract the unstability of network states for long strings which leads to a bad generalization performance of the RNN.

The last two reasons are specially applicable to the problem of regular grammatical inference using RNNs. On the other hand, self-clustering RNNs with discretized activations [ZeGS:93] or adaptive discrete states [DaMo:94] can be employed to ease the FSA extraction or even to fully substitute for an FSA in the learned task, since they perform identically to a DFA.

Several clustering techniques have been proposed by other researchers to extract an FSA from the activation patterns of a trained continuous RNN: dynamic clustering [DaDa:91], search on a regular partition [GiOm:93], k-means algorithm [ZeGS:93], moving markers [MaFa:94] (see Section 4.3). In general, these methods may be used independently of whether the network has been trained to learn a prediction or a classification task. It must be noted, however, that at least one key parameter of each algorithm must be set arbitrarily, such as the minimal inter-cluster distance for dynamic clustering, the cell width for the regular partition search algorithm, and the number of clusters for the k-means and moving markers methods.

The variability caused by the selection of different parameter values may be reduced to some extent by applying a DFA minimization algorithm to the initially extracted automaton; in fact, this step simply groups equivalent DFAs into their minimal canonical representatives. Nevertheless, an NFA could be extracted instead by the above methods, since a transition with a given symbol from two patterns in the same

cluster (state) could lead to patterns in distinct clusters[3].

When the RNN has been trained to classify a set of both positive and negative strings, the preceding FSA extraction methods present a more serious drawback, namely, they cannot guarantee the consistency with the examples of the returned FSA. The two possible causes of the extraction of an inconsistent FSA are the following:

a) an inadequate value has been assigned to some parameter of the clustering algorithm;

b) the RNN itself is not fully consistent with the training set due to a faulty or incomplete learning (e.g. gradient descent may have driven the net to a non-optimal local minimum).

In the first case, a consistent FSA could be finally obtained by repeating one or more times the clustering process with different parameter values, but in the second case, the preceding FSA extraction algorithms may not be able to repair what the RNN has failed to classify, since the natural clusters formed by the net may lead irremediably to an inconsistent FSA.

Two methods based on hierarchical clustering are presented next to extract a consistent deterministic UFSA[4] (or DUFA) from the dynamics of a RNN trained to learn the next-symbol prediction or the string classification task, respectively. However, since both methods share the same basic structure, they will be described jointly in a single algorithm to which a boolean parameter is given to select which non-common operations must be executed in each case. Let *onlypos* be the name of this boolean parameter, that will be assigned to TRUE if the RNN has been trained previously for a prediction task from only positive examples and to FALSE otherwise (the RNN has been trained for a classification task from both positive and negative examples). An important property of the proposed algorithm is that the returned DUFA is **always** a consistent DUFA compatible with the training sample, even if the RNN has not completely learned to classify the supplied strings. Note that in the case of only positive examples, the consistency of the extracted DUFA can be trivially achieved.

[3] In the regular partition search method by Giles and Omlin [GiOm:93], the possibility of reaching an NFA is eliminated by selecting, for each state and symbol, the destination state corresponding to the first transition visited with the given symbol from a pattern in the origin state; as a consequence, different DFAs may be extracted from the same network and set of strings depending on the string presentation order. In the other methods, where the mean of each cluster is stored, one way to avoid the extraction of NFAs is to use the cluster means as the only network states from which the transitions are recorded.

[4] *Unbiased* finite-state automata (UFSA) and their use in grammatical inference from positive and negative data have been discussed in the previous chapter.

The two-stage process of neural learning and UFSA extraction can be regarded as a hybrid approach to RGI, such that the internal state representation developed by the RNN in the former stage is used in the second stage to guide, through hierarchical clustering, a symbolic state merging process which starts on the sample prefix tree. For a review of how to train a RNN for the prediction and classification tasks see Sections 4.2.1 and 4.2.2, respectively. Here, we just assume that a certain RNN with N recurrent hidden units has been trained from a sample S until a stable minimum of the total error function has been reached. If the sample contains both positive and negative strings, $S = (S^+, S^-)$, then it is not required that all the examples in S be correctly classified by the final net. But, in any case, it is expected that the RNN has developed previously its own states as clusters of the hidden unit activation patterns. This information, together with a size-minimization heuristic, is used as inductive bias in the selection of a DUFA U that *consistently-covers* $PTU(S)$, the prefix tree UFSA of the sample. The top-level definition of the UFSA extraction procedure, common to the two methods aforementioned, is given in Algorithm 6.1.

ALGORITHM 6.1: *Consistent DUFA extraction from RNN dynamics*

Inputs:

onlypos	is a boolean parameter that selects the extraction method;
S	is a sample of a language, represented as a list of pairs (*s*, *class*), where *s* is a string and *class* is "+" or "-", depending on whether *s* belongs to the language or not;
net	is a RNN that has been trained previously either to predict (*onlypos*=TRUE) or to classify (*onlypos*=FALSE) the strings in S;
N	is the number of recurrent hidden units in *net*.

Outputs:

U	is a deterministic UFSA that *consistently-covers* $PTU(S)$, which is obtained through hierarchical clustering of the hidden unit activations of *net* over S.

Internal Variables:

PTU	is the prefix tree UFSA of the sample S;
nc	is an integer variable storing the number of clusters;
mean_clus	is an array containing the cluster mean vectors (of dimension N);
npt_clus	is an array containing the number of points in each cluster.

begin_algorithm

cluster_initialization_and_prefixtree_buildup (*onlypos*, S, *net*, N; **returns** *PTU*, *nc*, *mean_clus*, *npt_clus*);

U := hierarchical_clustering_and_state_merging (*onlypos*, N, *PTU*, *nc*, *mean_clus*, *npt_clus*);

end_algorithm

Algorithm 6.1 consists of two operations. In the first one, the prefix tree UFSA is built and, at the same time, single-point clusters are initialized with the hidden

unit activation patterns resulting after each symbol transition of the strings in the sample. In principle, there is a one-to-one correspondence between the initial clusters and the states of the prefix tree UFSA. However, if the learning task has been string classification, one might delete the single-point clusters associated with incorrectly classified strings (under a given tolerance ϵ) while keeping the corresponding states in the prefix tree. This step, that is carried out to remove the activation patterns leading to erroneous decisions from the process of hierarchical clustering, might also be skipped (e.g. by setting $\epsilon = 1$), since it is not needed to ensure the extraction of a consistent UFSA. Its effect on the extracted UFSA is to preserve some tails or branches of the prefix tree that are related to the strings the net has classified incorrectly (if any).

```
procedure cluster_initialization_and_prefixtree_buildup (onlypos, S, net, N;
                    returns PTU, nc, mean_clus, npt_clus)
i := 0;  PTU := ({λ},∅,∅,λ,∅,∅);  nstr := number_of_strings_in_training_set (S);
nc := 1;  mean_clus[0]:= fill_with_reset_activations (net, N);  npt_clus[0]:= 1;
while i < nstr do
    i := i + 1;
    <s,class>:= get_example (S,i); {where s is a string and class is "+" or "-"}
    PTU := expand_prefix_tree (PTU,s,class; returns tpath,tplen,k);
        {where tpath is the path of states of PTU visited by s, tplen is its length,
        and k is the position in tpath of the first new state due to s}
    reset_network_state (net); {initializes the recurrent unit activations}
    for j := 1 to tplen do
        compute_network_activations_from_input (s[j],net);
            {where s[j] is the symbol encoded in the input signals}
        if j ≥ k then
            st := tpath[j];  {where st is an integer identifier of a state}
            mean_clus[st]:= fill_with_rec_hidden_unit_activations (net, N);
            npt_clus[st]:= 1;  nc = nc + 1;
        end_if
    end_for
    if not onlypos then
        p := activation_output_unit (net);  { p ∈ [0,1] }
        if (p < (1 − ε) and class = "+") or (p > ε and class = "-") then
            for j := k to tplen do
                st := tpath[j];  nc = nc − 1;
                npt_clus[st]:= 0; {the cluster is marked as deleted}
            end_for
        end_if
    end_if
end_while
end_procedure
```

In the following function, a hierarchical clustering is carried out, so that the two closest clusters (under a certain distance measure, e.g. euclidean distance between cluster mean vectors) are determined and merged at each step. Each time two clusters are merged, a parallel merge of their associated states is performed in the UFSA representation. This is repeated until a stop condition is satisfied. The stop condition is what actually distinguishes the two UFSA extraction methods:

1) After training a prediction task (*onlypos*=TRUE), the stop condition is met when the distance between the two closest clusters is greater than a predetermined distance threshold *dist_thr*. Although this threshold might be set ad-hoc as a free parameter, an automatic procedure to compute it from the dimensionality N of the hidden activation space is proposed: to set *dist_thr* as a fixed ratio (e.g. 1/3) of the diameter of the associated N-dimensional hypercube.
2) After training a classification task (*onlypos*=FALSE), the stop condition is met when the deterministic merge of the two selected states yields an inconsistent DUFA. Once an inconsistency occurs, any subsequent merge will not remove it, so the clustering process is stopped and the last consistent DUFA is returned. By construction, the result is the minimum-size DUFA which *consistently-covers* $PTU(S)$ among the UFSA extractable from the trained RNN and sample through hierarchical clustering.

function hierarchical_clustering_and_state_merging (*onlypos*,N,PTU,*nc*,*mean_clus*,*npt_clus*)
returns UFSA;
if *onlypos* **then**
$dist\_thr := \sqrt{N}/3$; { for activation functions with range $[0,1]$ }
end_if
$U := PTU$;
repeat
find_closest_clusters (*nc*,*mean_clus*,*npt_clus*; **returns** *cl*1,*cl*2,*distance*);
if *onlypos* **then**
stop := (*distance* > *dist_thr*);
else
stop := **not** consistent (Dmerge($U, cl1, cl2$)); {see Definition 5.28}
end_if
if not *stop* **then**
merge_clusters (*cl*1,*cl*2; **returns** *mean_clus*,*npt_clus*); $nc := nc - 1$;
U := merge ($U, cl1, cl2$); {merges states *cl*1 and *cl*2, see Definition 5.19}
end_if
until *stop*;
return $D(U)$; {$D(U)$ is a consistent DUFA, see Definition 5.28}
end_function

It can be observed that neither the first nor the second method needs the input of any arbitrary clustering parameter, to the contrary of the previously reported FSA extraction approaches, although a distance measure between clusters must be selected in advance (e.g. centroid euclidean distance). In addition, the consistency analysis, that provides the stop criterion in the case of both positive and negative examples, is eased by using UFSAs, and this permits to guarantee the extraction of a consistent deterministic automaton in a single clustering process. Note also that, if the second method were applied to a sample containing only positive examples, then, independently of the RNN and the formed clusters, the extraction algorithm would always return the positive universal UFSA over the given alphabet Σ. Therefore, it is clear that, in the case of only positive examples for RGI, the first extraction method should be used after training the RNN to learn the next-symbol prediction task.

The computational cost of Algorithm 6.1 is the sum of the costs of the two subprograms just described. It is easily derived that the time complexity of the procedure "cluster_initialization_and_prefixtree_buildup" is $O(||S|| \cdot N^2)$, where $||S||$ denotes the total length of the sample S and the N^2 factor corresponds to the cost of computing the activations of all the RNN units[5] from the current inputs. On the other hand, the time complexity of the function "hierarchical_clustering_and_state_merging" is at first $O(|PTU(S)|^3 \cdot T(\text{distance\_computation}))$, due to the search of the closest pair of clusters that is carried out at each iteration of the main loop. Nevertheless, if an additional data structure, formed by a heap plus an indexing array, with a total space requirement of $O(|PTU(S)|^2)$, were used to store and maintain the cluster distances in a partial order, then the time complexity of the function would be reduced to $O(|PTU(S)|^2 \cdot (log(|PTU(S)|) + T(\text{distance\_computation})))$. Concerning the cost of computing the distance between two clusters, it will depend on the distance measure selected, but a lower bound of $T(\text{distance\_computation}) = O(N)$ is obtained for the simple euclidean distance between cluster means. Therefore, the time complexity of both UFSA extraction methods described is $O(||S|| \cdot N^2 + |PTU(S)|^3 \cdot N)$, but since the second term cost is larger, the time complexity may be simplified to $O(|PTU(S)|^3 \cdot N)$, and it could reach $O(|PTU(S)|^2 \cdot (log(|PTU(S)|) + N))$ at the expense of increasing the associated space complexity from $O(|PTU(S)|)$ up to $O(|PTU(S)|^2)$.

Figure 6.1 illustrates the whole RGI approach by showing an example of application of the second extraction method to a real case. A first-order ASLRNN with 3 recurrent hidden units and 1 output unit was trained to classify all the binary strings of length ≤ 4 according to the odd-parity predicate. After learning, when the net classified correctly all the strings in the training set, the parallel process of hierarchical clustering in 3-D

[5]Assuming that either an SLRNN or an ASLRNN is used with a total number of weights in the order of N^2. In the case of a second-order SLRNN or ASLRNN, however, the number of weights and the associated cost are actually in the order of N^2M, where M is the number of input signals.

space and state merging was performed, and it led from the sample prefix tree to a 3-state DUFA equivalent to the target (2-state) odd-parity recognizer. Hence, the target UFSA could be obtained by applying a state minimization algorithm to the extracted DUFA.

However, after intensive experimentation, we have realized that, in the most part of the tested cases and specially when the input sample (and the prefix tree UFSA) is large, the stop criterion for the merging process in the second method described is too strict, and consequently, excessively large DUFAs are returned which do not generalize the input data as much as desired. In other words, although the method chooses a consistent DUFA U in $Lat(PTU(S))$, U does not belong usually to the deterministic border set $DBS_{PTU}(S)$ containing the "best" hypotheses (i.e. the deterministic and consistent maximal generalizations of the sample, see Def.5.27). In order to arrive at the deterministic border set, or at least to come close to it, an extension of the UFSA extraction method for positive and negative examples is presented next, that permits to reduce further the size of the extracted DUFA while still guaranteeing its consistency. The price to pay for this improvement is a somewhat higher computational cost and the introduction of a parameter in the extraction algorithm.

The proposed change consists of replacing in Algorithm 6.1 the call to the former function "hierarchical_clustering_and_state_merging" by a call to the new function "extended_hierarchical_clustering_and_state_merging", which is defined hereinafter, after entering a positive integer value for the new argument k. Now, in the method for positive and negative strings ($onlypos = FALSE$), instead of finding the pair of closest clusters, an ordered list with the (at most) k pairs of closest clusters is determined. The first cluster pair in the ordered list that yields a consistent DUFA, after the deterministic merge of the corresponding states, is selected to be actually merged, and the merging process is stopped when all the pairs in the list lead to inconsistent DUFAs.

Hence, the former function is covered by the particular case of $k = 1$. The extracted UFSA is guaranteed to be a consistent DUFA for every $k \geq 1$, but a higher level of generalization is achieved (the returned DUFA is smaller) as a larger value of k is supplied. If $k \geq nc(nc-1)/2$, for the final number of clusters nc, then it is sure that the extracted UFSA belongs to the deterministic border set $DBS_{PTU}(S)$; it could also belong to $DBS_{PTU}(S)$ although the above inequality were not satisfied, but in that case it is not known whether the deterministic border set has been reached. In practice, for relatively small target UFSAs, a value of k in the interval $[50, 100]$ may be enough to obtain a solution in the $DBS_{PTU}(S)$ or close to it, whenever the RNN has reasonably learned the string classification task[6].

[6]Of course, this is only a heuristic figure taken from empirical results.

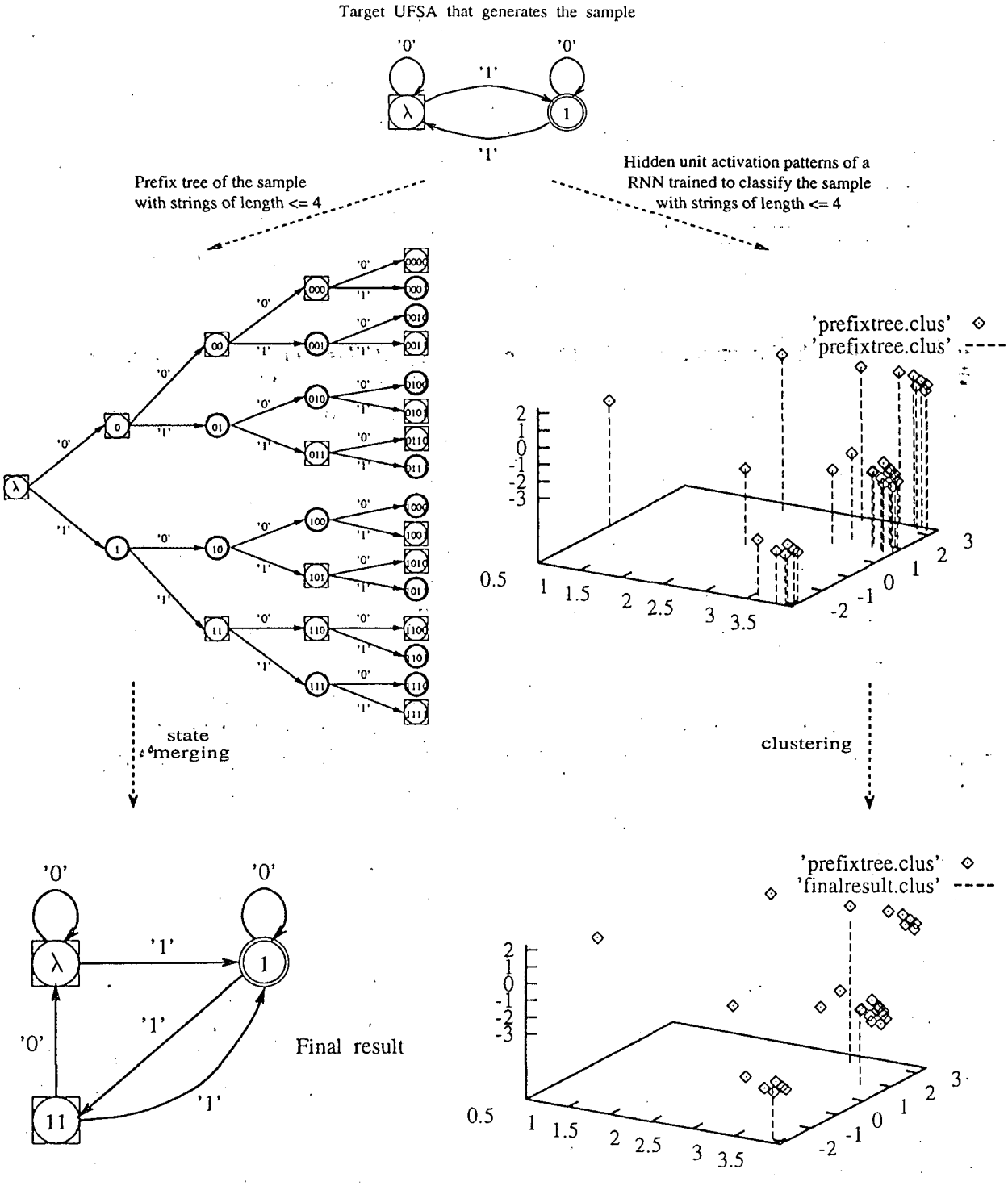

Fig. 6.1 *Inference of an odd-parity recognizer by neural learning and UFSA extraction, using a first-order ASLRNN with 3 recurrent hidden units.*

```
function extended_hierarchical_clustering_and_state_merging
        (k,onlypos,N,PTU,nc,mean_clus,npt_clus) returns UFSA;
if onlypos then
    dist_thr := √N/3;  { for activation functions with range [0,1] }
end_if
U := PTU;
repeat
    if onlypos then
        find_closest_clusters (nc,mean_clus,npt_clus; returns cl1,cl2,distance);
        stop := (distance > dist_thr);
    else
        find_k_closest_cluster_pairs (k,nc,mean_clus,npt_clus;
                                      returns CL1,CL2,DISTANCES);
        i := 1;  found := FALSE;  maxi := min(k, nc(nc − 1)/2);
        while not found and i ≤ maxi do
            cl1 := CL1[i];  cl2 := CL2[i];
            if consistent (Dmerge(U,cl1,cl2)) then found := TRUE;
            else i := i + 1;
            end_if
        end_while
        stop := not found;
    end_if
    if not stop then
        merge_clusters (cl1,cl2; returns mean_clus,npt_clus);  nc := nc − 1;
        U := merge (U,cl1,cl2);  {merges states cl1 and cl2, see Definition 5.19}
    end_if
until stop;
return D(U);  {D(U) is a consistent DUFA, see Definition 5.28}
end_function
```

It must be noted that, for a non small k, the latest cluster merges peformed (when several previous attempts are rejected) may be somewhat artificial or forced, since two distant or well separated clusters may be merged. For this reason, a small value of k should be chosen if the aim is to extract a DUFA that is approximately simulated by the trained network. In that case, an alternative way to reach a DUFA in the $DBS_{PTU}(S)$ is to apply a lexicographically-ordered state merging process (as in Algorithm 5.2) starting on the extracted DUFA. Likewise, the systematic application of a state minimization algorithm to the DUFA returned by the clustering algorithm is recommended to simplify the result, although this does not affect the inferred languages.

Finally, the time computational cost of the UFSA extraction method for a positive and negative sample augments linearly in proportion to k. The worst-time complexity is now of $O(|PTU(S)|^3 \cdot (k+N))$, due to the procedure "find_k_closest_cluster_pairs", and a k factor is also involved in the cost of finding a consistent DUFA derived from the current UFSA during clustering, as the introduction of the inner "while" loop in the new function indicates.

6.2 Effect of different activation functions on learning performance of RNNs

As it has been reviewed in Chapter 4, several RNN architectures have been devised in recent years to deal with tasks involving sequences. Likewise, the learning capabilities of the different models on some benchmark problems of RGI have been reported [ClSM:89, SmZi:89, Fahl:91b, Poll:91, WaKu:92, MiGi:93, ZeGS:93, MaFa:94, DaMo:94].

It is evident that the neural learning algorithm used is a very important component of the connectionist methods (besides the network's topology and activation function) that may have a notable influence both in the learning efficiency and the performance of the trained network. Even though some kind of gradient-based optimization scheme has been proposed as learning algorithm for each one of the RNN architectures devised, some remarkable differences can be noticed. Thus, the reported gradient-based learning techniques can be classified in three groups:

1) those that compute the real gradient of the error function with respect to all the weights of the RNN (the so-called *full gradient* or *complete gradient* algorithms [WiZi:89, WiPe:90, Schm:92, GiMC:92, WaKu:92]);
2) those that compute an approximation of the real gradient for the weights of the recurrent units by truncating the time backward recursion (*truncated gradient* algorithms [Elman:90, Poll:91]); and
3) those that compute the gradient through a differentiable activation function that approximates the discrete activation function actually used in the units (*pseudo-gradient* algorithms [ZeGS:93]).

Normally, the learning algorithms in the first group should work better than those in the other two groups. Indeed, some researchers [SmZi:89, WaKu:92, GiMC:92] have pointed to the use of truncated gradient algorithms as one of the causes of the relatively poor learning capabilities shown in some of the previous studies [SeCM:88, Poll:91]. In that respect, it is proper to remember that the same backpropagation algorithm used for training multi-layer feed-forward nets was proposed as learning rule for Elman's simple

recurrent networks (SRNs) [Elman:90], and it was actually used, despite of its truncated gradient characteristic for the recurrent model, in the former published studies with this architecture [SeCM:88, ClSM:89, SeCM:91, DaDa:91, CaCV:93]. However, as SRNs are a particular case of first-order ASLRNNs, they can be trained as well by the complete gradient learning procedure described in Section 4.1.2.

On the other hand, the topic of the influence of the activation function on the learning performance of RNNs has not been analysed in the preceding works. Thus, the use of a sigmoid activation function, either the one given by Eq.(4.5) or the very similar hyperbolic tangent, has been common and, as far as we know, unquestioned in all the studies reported about RNNs, independently of the specific architecture and learning rule employed. This is quite surprising, since the sigmoid activation function presents some theoretical drawbacks for its usage both in RNNs and multi-layer feed-forward nets, which are explained in the next subsection.

Later on, some experimental results are reported, that have confirmed the conjectures raised about an improvement in the learning capabilities of RNNs if the sigmoid function is replaced by other "more suitable" activation functions. In addition, the complete-gradient learning algorithm for training ASLRNNs has been shown to be clearly superior to the truncated-gradient learning scheme (simple backpropagation).

6.2.1 Alternatives to the use of the sigmoid function as activation function in RNNs

In order to analyse the effect of the activation function on the computational and learning behaviors of RNNs, let us first study the case of the first-order SLRNN architecture. Hence, a fully-connected single-layer of N recurrent units, with first-order connections from a set of M input signals and a set of N previous activation values of the units, is considered. The same notation used in Section 4.1.1 for input and state vectors and network dynamics is followed.

Let us split the weights of the first-order SLRNN in two groups, corresponding to the connections from inputs and recurrent units, respectively. This is, let W_I $(N \times M)$ be the matrix of weights associated with the inputs, and let W_C $(N \times N)$ be the square matrix of weights associated with the (state) recurrent connections, where in both matrices, the weights of the k-th unit are placed in the k-th row, $1 \leq k \leq N$. Now, if $g : \Re \longrightarrow \Re$ denotes the activation function used in all the units, let $G : \Re^N \longrightarrow \Re^N$ denote the corresponding vectorial mapping in the N-dimensional space, which is defined as

$$G([\sigma_1, ..., \sigma_N]^T) = [g(\sigma_1), ..., g(\sigma_N)]^T. \quad (6.1)$$

In this way, starting on an initial network state $\mathbf{S}^1 = [y_1(0), ..., y_N(0)]^T$, the sequence of state vectors $\mathbf{S}^2$,$\mathbf{S}^3$,...,$\mathbf{S}^{t+1}$, computed by the SLRNN in response to a sequence of input vectors $\mathbf{I}^1$,$\mathbf{I}^2$,...,$\mathbf{I}^t$, can be expressed as follows

$$\begin{aligned}
\mathbf{S}^2 &= G(W_I\mathbf{I}^1 + W_C\mathbf{S}^1) \\
\mathbf{S}^3 &= G(W_I\mathbf{I}^2 + W_C G(W_I\mathbf{I}^1 + W_C\mathbf{S}^1)) \\
&\vdots \\
\mathbf{S}^{t+1} &= G(W_I\mathbf{I}^t + W_C G(W_I\mathbf{I}^{t-1} + W_C G(W_I\mathbf{I}^{t-2} + W_C\mathbf{S}^{t-2}))).
\end{aligned}$$

The last equation can be fully expanded in terms of the entered inputs and the initial state:

$$\mathbf{S}^{t+1} = G(W_I\mathbf{I}^t + W_C G(W_I\mathbf{I}^{t-1} + W_C G(\cdots\cdots G(W_I\mathbf{I}^1 + W_C\mathbf{S}^1)\cdots))). \quad (6.2)$$

Therefore, it can be observed that the past history is internally stored in the SLRNN as a state vector resulting from a sum of terms that is recursively filtered by the activation function.

In the particular case of a linear activation function

$$g_{lin}(\sigma) = a\,\sigma, \quad (6.3)$$

the contribution of each term to the network state is clearly distinguished, as given by

$$\mathbf{S}^{t+1} = a^t W_C^t\,\mathbf{S}^1 + \sum_{\tau=1}^{t} a^{t+1-\tau} W_C^{t-\tau} W_I\,\mathbf{I}^\tau. \quad (6.4)$$

However, for a non-linear activation function, the contribution of each one of the past inputs is more difficult to see.

Intuitively, one may expect that the use of a bounded monotonic function like the sigmoid excessively filters the previous inputs, thus possibly cutting information about the sequence which is relevant to perform the task at hand (e.g. for predicting the next input). Consequently, a linear activation function or a non-linear unbounded function would allow a better performance of the SLRNN for complex tasks that require to remember or take into account non-recent inputs, since a "better" record of the entered input sequence would be maintained in the recurrent hidden units.

In order to discuss the above topic more formally, let us proceed to analyse the sensitivity of the state vector $\mathbf{S}^{t+1} = [y_1(t), ..., y_N(t)]^T$ with respect to an input vector $\mathbf{I}^{t-k} = [x_1(t-k), ..., x_M(t-k)]^T$ that has been entered to the SLRNN k time steps before. Let $\partial\mathbf{S}^{t+1}/\partial\mathbf{I}^{t-k}$ denote the corresponding sensitivity matrix, that contains the partial derivatives $\partial y_i(t)/\partial x_j(t-k)$, for $1 \leq i \leq N$, $1 \leq j \leq M$. In addition, let

$\sigma^t = [\sigma_1(t), ..., \sigma_N(t)]^T$ denote the vector of the recurrent unit net-input values at time step t, which are given by Eqs.(4.1) and (4.3).

By applying the chain rule, we have that

$$\frac{\partial \mathbf{S}^{t+1}}{\partial \mathbf{I}^{t-k}} = \frac{\partial \mathbf{S}^{t+1}}{\partial \sigma^t}\frac{\partial \sigma^t}{\partial \sigma^{t-1}} \cdots \frac{\partial \sigma^{t-k+1}}{\partial \sigma^{t-k}}\frac{\partial \sigma^{t-k}}{\partial \mathbf{I}^{t-k}}. \tag{6.5}$$

Assuming that the network weights are not changed during the sequence computation, it turns out that, for every $t \geq 1$,

$$\frac{\partial \mathbf{S}^{t+1}}{\partial \sigma^t} = D(t), \tag{6.6}$$

where $D(t)$ denotes the diagonal matrix

$$D(t) = \begin{pmatrix} g'(\sigma_1(t)) & 0 & \cdots & 0 \\ 0 & g'(\sigma_2(t)) & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & g'(\sigma_N(t)) \end{pmatrix}, \tag{6.7}$$

$$\frac{\partial \sigma^t}{\partial \mathbf{I}^t} = W_I, \tag{6.8}$$

and

$$\frac{\partial \sigma^t}{\partial \mathbf{S}^t} = W_C; \tag{6.9}$$

also, for every $t \geq 2$,

$$\frac{\partial \sigma^t}{\partial \sigma^{t-1}} = \frac{\partial \sigma^t}{\partial \mathbf{S}^t}\frac{\partial \mathbf{S}^t}{\partial \sigma^{t-1}} = W_C\, D(t-1). \tag{6.10}$$

Hence, by substituting conveniently the factors in the right hand side of Eq.(6.5) by the matrices given by the above equations, we obtain the following general expression

$$\frac{\partial \mathbf{S}^{t+1}}{\partial \mathbf{I}^{t-k}} = D(t)\, W_C\, D(t-1) \cdots W_C\, D(t-k)\, W_I, \tag{6.11}$$

that, for the particular case of $k = 0$, is reduced to

$$\frac{\partial \mathbf{S}^{t+1}}{\partial \mathbf{I}^t} = D(t)\, W_I. \tag{6.12}$$

In the same way, it is easily derived that the sensitivity of the state vector $\mathbf{S}^{t+1}$ with respect to the initial state $\mathbf{S}^1$ is given by

$$\frac{\partial \mathbf{S}^{t+1}}{\partial \mathbf{S}^1} = D(t)\, W_C\, D(t-1) \cdots W_C\, D(1)\, W_C. \tag{6.13}$$

Now, if the network weights can be modified at each time step, as in the case of on-line learning using the RTRL algorithm, then Eqs.(6.11), (6.12) and (6.13), are respectively converted in the following ones:

$$\frac{\partial \mathbf{S}^{t+1}}{\partial \mathbf{I}^{t-k}} = D(t)\, W_C(t)\, D(t-1) \cdots W_C(t-k+1)\, D(t-k)\, W_I(t-k), \tag{6.14}$$

$$\frac{\partial \mathbf{S}^{t+1}}{\partial \mathbf{I}^{t}} = D(t)\, W_I(t), \tag{6.15}$$

$$\frac{\partial \mathbf{S}^{t+1}}{\partial \mathbf{S}^{1}} = D(t)\, W_C(t)\, D(t-1) \cdots W_C(2)\, D(1)\, W_C(1), \tag{6.16}$$

where the matrices $W_I(t)$ and $W_C(t)$ contain the network weights at time step t associated with the input and recurrent unit connections, respectively.

Either the weights are changed or not during the computation of the sequence of states, it is clear from the preceding equations that the derivative g' of the activation function plays a fundamental role in the sensitivity of the current state of the first-order SLRNN with respect to the past history, since g' is involved in the multiplying diagonal matrices $D(t), \ldots, D(1)$. A similar conclusion could have been drawn from the analysis of the dynamics of second-order SLRNNs. Moreover, if we take a look to the full-gradient learning algorithms described in Section 4.1.1 both for first- and second-order SLRNNs, we will realize that multiplying factors $g'(\sigma_k(\tau))$, $1 \leq k \leq N$, $1 \leq \tau \leq t$, intervene in the computation of the error gradient $\nabla_{\mathbf{W}} E(t)$, thus affecting the contribution of the past events to the current weight update. The problem resides on the fact that, if many values $g'(\sigma_k(\tau))$ are close to zero, both the current state of the SLRNN and the current weight change for learning a task will depend almost exclusively on the very recent inputs; in other words, the network will have a very short memory of the previous inputs to perform and learn the required task.

Trying to avoid the above problem, Sopena proposed the use of a linear activation function in the recurrent units of a first-order three-layer ASLRNN, within a connectionist approach to natural language parsing [Sope:91]. Thus, if g_{lin} (Eq.6.3) is used in the recurrent layer, then the sensitivity matrix $\partial \mathbf{S}^{t+1}/\partial \mathbf{I}^{t-k}$ for the case of fixed weights is given by

$$\frac{\partial \mathbf{S}^{t+1}}{\partial \mathbf{I}^{t-k}} = a^{k+1} W_C^k W_I, \tag{6.17}$$

and if the slope parameter a is set to 1, then the influence of each of the previous inputs on the network state will only depend on the (trainable) network weights.

On the other hand, the use of a linear activation function bears two drawbacks:

a) only linear mappings from inputs to states (and linear state transition functions) can be implemented[7];
b) un unstable behavior of the network may occur such that the magnitudes of the activation values, as well as the weights during learning, shoot up causing overflow errors[8].

Therefore, a nonlinear activation function that combined the respective advantages of the linear and sigmoid functions while avoiding their drawbacks would be helpful. This basic idea has led us to propose the use of the antisymmetric logarithm as an adequate activation function for recurrent units in RNNs. The antisym-log function is defined as

$$g_{al}(\sigma) = \mathrm{sgn}(\sigma)\ \log(1 + a|\sigma|), \tag{6.18}$$

where $\mathrm{sgn}(\sigma) = +1(-1)$ for $\sigma \geq 0$ ($\sigma < 0$), respectively, log denotes the natural logarithm[9], and a is a positive parameter, the slope at the origin.

The function g_{al} can be thought of as a compromise between the linear function g_{lin} of Eq.(6.3) and the antisymmetric sigmoid function g_{as} ranged in the interval (-1,1), given by

$$g_{as}(\sigma) = \frac{2}{1 + e^{-a\sigma}} - 1, \tag{6.19}$$

as can be observed in the top of Fig.6.2, where $a = 1$ has been set for the three functions. The corresponding first derivatives

$$g'_{lin}(\sigma) = a, \tag{6.20}$$

$$g'_{al}(\sigma) = \frac{a}{1 + a|\sigma|}, \tag{6.21}$$

$$g'_{as}(\sigma) = 2a\ g_{as}(\sigma)\ (1 - g_{as}(\sigma)), \tag{6.22}$$

are displayed in the bottom of Fig.6.2 (also for $a = 1$). It can be seen that the derivative of the sigmoid is only significant for a small domain centered at 0, whereas the derivative of the antisym-log is always greater and decays less abruptly, being significant for a much wider interval. Hence, for the aforementioned reasons, the learning performance of an SLRNN should improve by using g_{al} instead of g_{as} (or g_s) as activation function.

[7]An SLRNN with a linear activation function would implement a linear sequential machine [Booth:67] in the case that only a finite set of state vectors could be reached at any time step from a given finite set of input vectors.

[8]This problem can be somewhat alleviated in practice by setting a small positive $a < 1$, e.g. $a = 1/N$ or $a = 1/(N + M)$, though the smaller the value of a, the lesser sensitivity of the state with respect to the previous inputs, as can be seen in Eq.(6.17).

[9]Other logarithmic bases (e.g. 2, 3, ...) might be chosen as well.

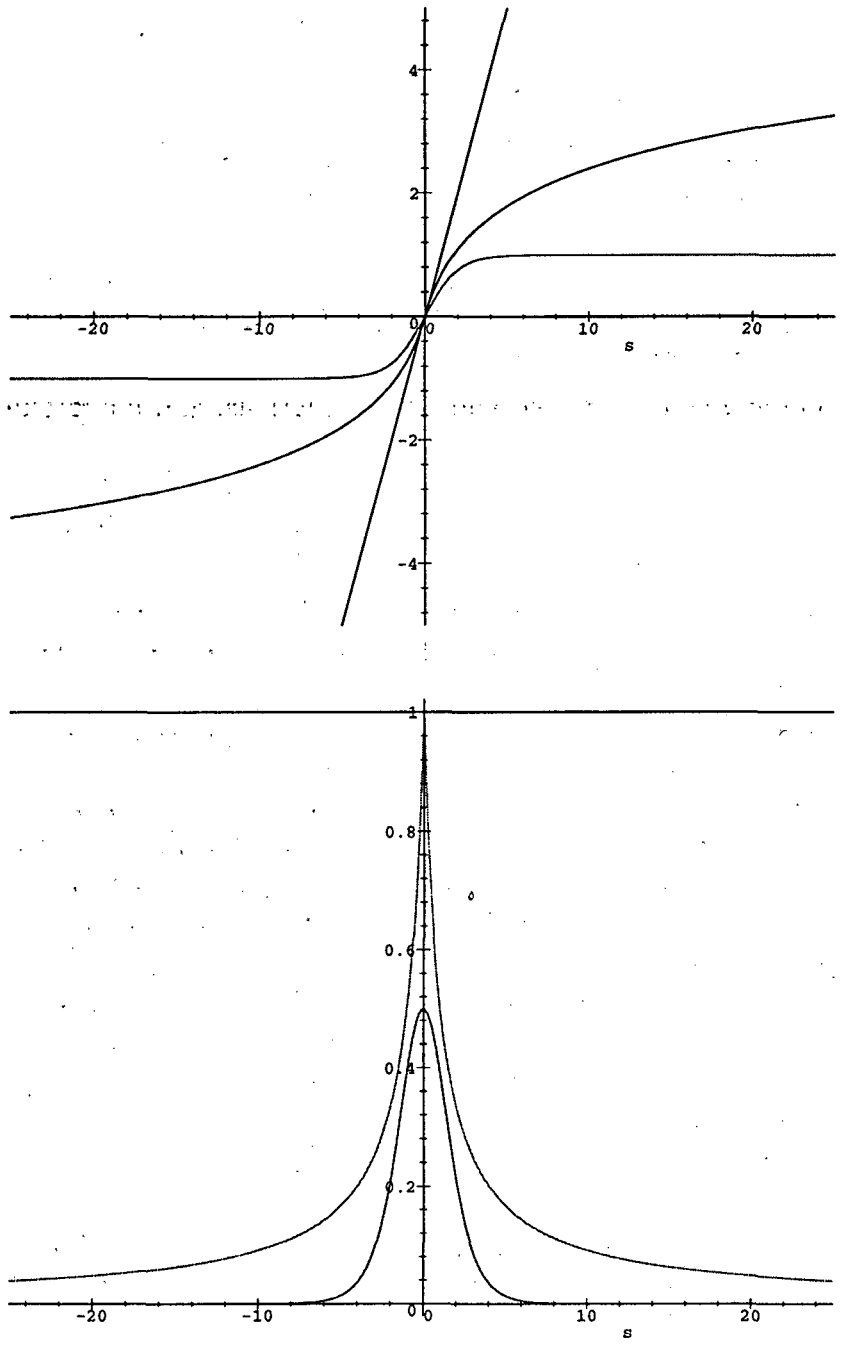

Fig. 6.2 *The linear, antisym-log and antisym-sigmoid functions (top) and their corresponding first derivatives (bottom).*

Moreover, the antisym-log function possesses other interesting properties:

a) although it is unbounded like the linear function, it does not cause unstable behaviors in RNNs, since in the case of large weights and net-input values, the successive activation values of the units in the recurrent layer do not shoot up;
b) a multilayer feedforward (MLFF) network with the antisym-log activation function in the hidden units is a universal approximator, since the antisym-log is a nonpolynomial function [LeLP:93];
c) the antisym-log can also be used as activation function in the output units of a universal approximator MLFF network (instead of a linear function), since it is an invertible function [Funa:89];
d) it is monotonic, like the sigmoid and linear functions, and monotonic activation functions have been recommended to be used in networks trained by the generalized delta rule (backpropagation) in order to avoid a certain type of local minimum problems typical of some nonmonotonic functions [RuHW:86,DaSc:92].

Due to the above features, the antisymmetric logarithm can be proposed as a kind of "standard" nonlinear unbounded activation function for neural networks.

Now, let us turn our attention to the ASLRNN architectures, in which one or more layers of feedforward units with first-order connections and bias weight are added to an SLRNN, with the purpose of improving the capability of learning and implementing output functions [GoGi:93]. In these cases, the recurrent layer is mainly involved in the inference of some state transition function for the task, for which the net builds its own state representation, whereas the additional feedforward layer(s) contribute to the approximation of a mapping from current input and state to target output. What types of activation function are adequate for the units of the feedforward layers in ASLRNNs? The logical but indirect answer is "those that are adequate for the units in MLFF networks". Hence, it is worthwhile to recall some theoretical results about MLFF networks and their approximation abilities.

If a sigmoid activation function is used in the hidden units and a linear activation function is used in the output units, an MLFF network with two hidden layers and sufficiently many hidden units per layer can approximate any function arbitrarily well [HeKP:91]. This can be shown by noting that any "reasonable" function can be represented by a linear combination of localized *bumps*, which are each non-zero only in a small region of the domain, and such *bumps* can be constructed with two hidden layers of sigmoid units [LaFa:88]. It has also been proved that only one hidden layer with sufficiently many sigmoid units is enough to approximate any *continuous* function [Cybe:89,HoSW:89], and this result has been generalized to bounded continuous activation functions that are nonconstant [Horn:91] and later

to locally bounded piecewise continuous activation functions that are nonpolynomial [LeLP:93]. In all cases, the hidden units are supposed to include a bias weight (which is often called threshold).

In the radial basis function (RBF) networks [HaKK:90], the hidden units themselves have a localized bump-like response, each becoming activated only for inputs in some small region of the input space, so that only one hidden layer of such units and an output layer of linear units are needed to represent any reasonable function. The hidden units in RBF networks are characterized by a Gaussian activation function

$$g_G(\sigma) = e^{-\pi\sigma^2}, \tag{6.23}$$

where the net-input σ is defined as the distance between the input vector and the weight vector of the unit, which represents a point in the input space.

For both the RBF and MLFF nets that are universal approximators of mappings from $\Re^M$ to $\Re^P$, the linear activation function in the output units can be replaced by a nonlinear invertible activation function [Funa:89]. Furthermore, if the range of the possible output values is bounded, e.g. for mappings from $\Re^M$ to $[0,1]^P$, then the activation function in the output units needs only to be invertible for the given range.

Although the universal approximation capability is desirable, the utility of these theoretical results for practical purposes is somewhat limited, since the number of hidden units that are necessary is not known in general, and in many cases it may grow exponentially with the number of input units (as occurs in the case of the general Boolean functions [HeKP:91]). Also, the bump-based constructions say nothing about learning or generalization, and it is possible that some functions are representable but extremely difficult to learn with two hidden layers of sigmoid units, perhaps because of local minima. In fact, the use of alternative activation functions may reduce considerably the number of required units or may speed up learning [DaSc:92].

Moreover, when discrete pattern classification, rather than approximation of continuous real mappings, is the goal, then the computational power of the network rests with how the units carve up an input space into arbitrary regions [Lipp:87]. In this sense, MLFF networks with different types of activation functions and RBF networks are equivalent in that networks of one type can emulate how networks of another type carve the input space up. However, for a given classification problem, the size of a specific network solution and the easiness of learning such a solution from the training set may vary notably from one type of network to another.

Consider, for instance, the well-known XOR problem. As it is displayed in Fig.6.3, a single sigmoid unit (or a single threshold unit) cannot approximate the XOR function, always failing at least in one of the four points, whereas a single unit

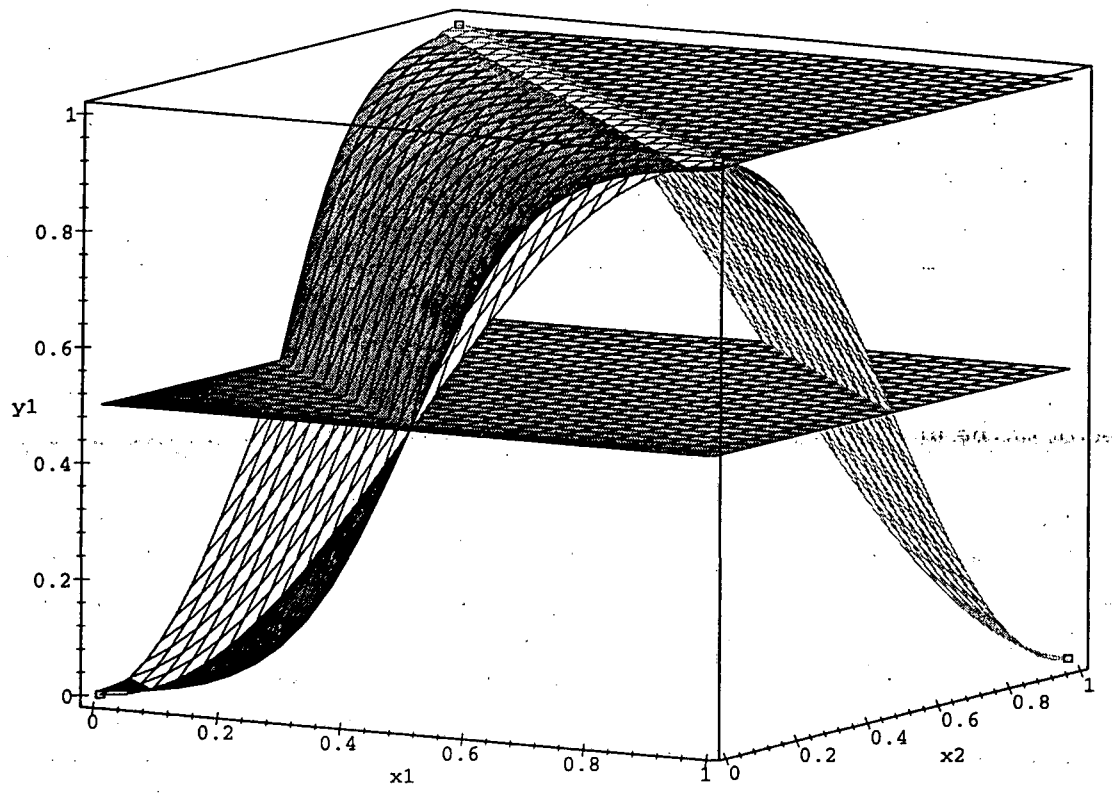

Fig. 6.3 *The XOR function approximated by a single unit: a sigmoid unit cannot represent the function, whereas a unit with a sinusoidal activation function can perfectly do it.*

with a nonmonotonic activation function like a sinusoidal or a Gaussian[10] function can represent it. Of course, this is due to the fact that a unit with a monotonic activation function like a sigmoid or a step function is restricted to make linearly separable discriminations. In order to build or learn an MLFF network with sigmoid units that solves the XOR problem, two hidden unit and one output unit are needed at least; to learn such a solution using backpropagation typically requires hundreds of epochs [HeKP:91]. On the other hand, just a few epochs are enough to learn the XOR function using a single sinusoidal unit.

Dawson and Schopflocher [DaSc:92] proposed the use of the Gaussian activation function given by Eq.(6.23) in the output and hidden units of MLFF networks for classification tasks, where the net-input σ of a Gaussian unit is not a distance as in

[10]Not as it is used in RBF networks, but simply applied to the net-input resulting from the sum of the bias plus the inner product of the input and weight vectors.

RBF nets, but a weighted sum of inputs plus a bias weight (as usually in MLFF nets). The difference between hidden RBF units and Gaussian units in MLFF nets is better appreciated by comparing their respective receptive fields. A hidden unit in an RBF network has a symmetric, Gaussian shaped receptive field in input space centered at the point represented by the unit weights, and the decision region that is obtained by setting a threshold on the unit activation value is a hypersphere. In contrast, the receptive field of a Gaussian unit is a Gaussian-contoured "hyperhill" that cuts through the input space at an orientation that is perpendicular to the unit weight vector, and therefore, the corresponding decision region is bounded by two parallel hyperplanes. In addition, a Gaussian unit in an MLFF net has a nonmonotonic activation function with respect to the net-input σ, whereas the activation function of an RBF hidden unit is actually decreasing monotonic, since σ is a distance in RBF units and thus always non-negative.

Dawson and Schopflocher performed some experiments comparing the ability of MLFF nets with sigmoid and Gaussian units, respectively, to learn some pattern classification problems, including the XOR. The results reported showed that MLFF nets with Gaussian units learn pattern discriminations more quickly than standard MLFF nets with sigmoid units. Furthermore, because the Gaussian activation function is nonmonotonic, fewer units are required in general in Gaussian-based MLFF nets to make the discriminations that solve the classification tasks [DaSc:92].

However, in order to obtain good learning results with the Gaussian units, Dawson and Schopflocher needed to change the cost function used in the backpropagation algorithm, in such a way that two terms were involved: the first one measuring the square error between observed and desired output, which is normally the only one, and the second one measuring how far from the center of the Gaussian is the observed net-input (of an output unit) when the desired output is 1. This modification indeed requires that the target output of a unit be either 0 or 1, i.e. it restricts the possible tasks to mappings from $\Re^M$ (or some subset of $\Re^M$) to $\{0,1\}^P$. The introduction of the second term in the error function is aimed at avoiding to fall during learning in a type of local minima, which are often reached if only the first term is included, and which are characterized by pulling all the net-inputs towards either negative or positive infinity [DaSc:92].

We claim that using a sinusoidal activation function in the units of an MLFF network should lead to even better results in terms of learning time and network size, without the need of changing the standard cost function nor restricting the target outputs to binary values. Let us define

$$g_{sin}(\sigma) = \frac{1}{2}(1 + \sin(a\sigma)), \tag{6.24}$$

where a is a constant, as a sinusoidal activation function ranged in $[0,1]$.

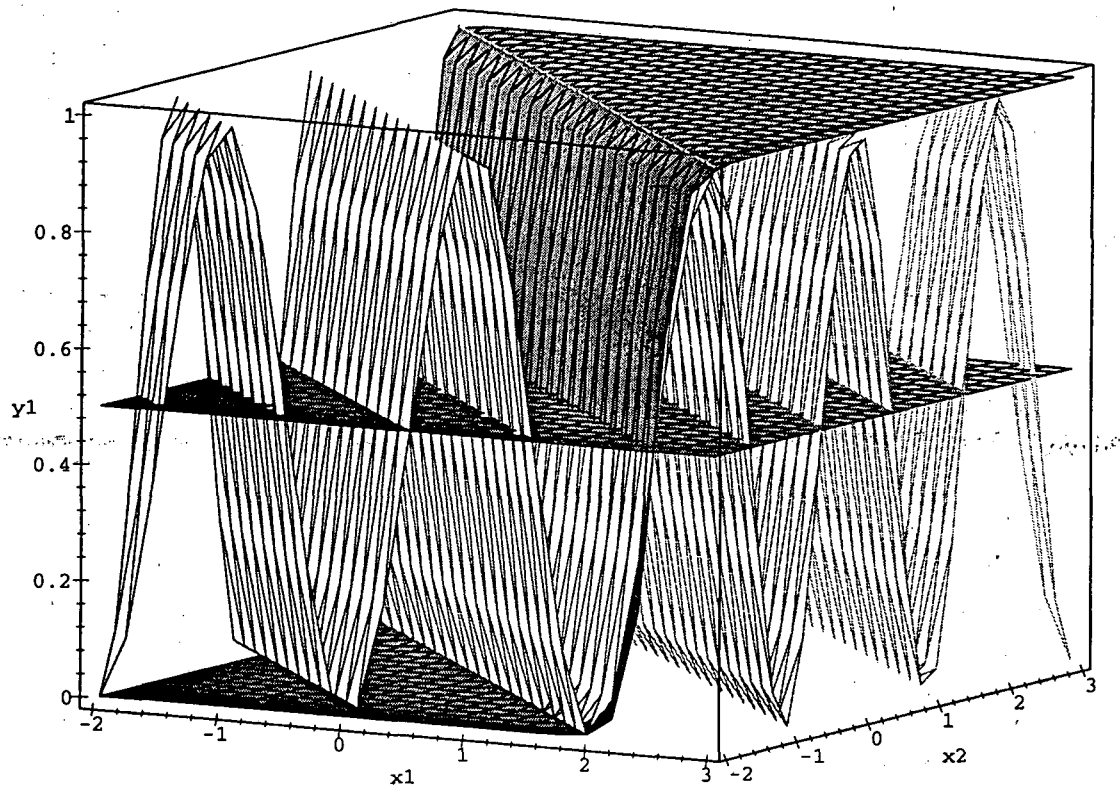

Fig. 6.4 *The receptive fields, in two-dimensional input space, of a sigmoid unit and a sinusoidal unit.*

Some arguments can be given to prefer g_{sin} (6.24) instead of the sigmoid g_s (4.5) or the Gaussian g_G (6.23), which are explained next.

First, a sinusoidal activation function is more powerful to make discriminations and carve up a pattern space, and this should allow the existence of learnable solutions for classification tasks using MLFF networks with fewer units. The receptive field of a single unit with g_{sin} as activation function, which is displayed in Fig.6.4 for a 2D input space, is a "sine-contoured" (infinite) succession of parallel "hyperhills" and "hypervalleys" of adjustable width that cut through the input space at an orientation that is perpendicular to the unit weight vector; the corresponding decision region for binary classification is given by a succession of alternate ON/OFF "hyperslices" of adjustable width limited by a succession of parallel hyperplanes. In contrast, the decision region for a sigmoid unit is just the semi-space defined by one hyperplane (Fig.6.4), what is obvious if a sigmoid is regarded as a smoothed step function. Also, the decision region for a Gaussian unit corresponds to just one of the parallel ON hyperslices obtained using a sinusoidal unit.

Second, MLFF nets with sinusoidal hidden units should also be excellent universal approximators of continuous mappings, since they can be seen as generalized discrete Fourier series with adjustable frequencies [LaFa:87]. Consider an MLFF net with a single hidden layer of sinusoidal units and an output layer of linear units. Then it is not hard to see that the weights of the output layer act like Fourier amplitudes, the weights of the hidden layer act like frequencies (in addition to determine the orientation of the receptive field of each hidden unit), and the bias weights of the hidden units act like phases. Note however that in contrast to a common Fourier decomposition in which the values of the frequencies are fixed, the MLFF net has the ability to adjust the values of the frequencies to obtain the minimum least mean square error, where the number of adjustable frequencies is given by the number of hidden units. Once this number is specified, the net adjusts the numerical value of these variable frequencies, together with the amplitudes and phases, to produce a best fit. It is expected that by adding more hidden units (i.e. more adjustable frequencies) the accuracy of the approximation will improve. Lapedes and Farber termed this kind of mode decomposition performed by the MLFF net as "generalized Fourier decomposition", where "generalized" refers to the ability to adjust frequencies, because in conventional Fourier mode analysis only the amplitudes are adjusted whereas the frequencies are fixed [LaFa:87].

The above analogy is perfect in the case of a single input unit with real values. If a multidimensional input is involved, i.e. a vector in $\Re^M$, then each hidden unit is associated not only with a frequency but also with an adjustable hyperplane that determines the orientation of a wave front. Further generalization is achieved by allowing two or more hidden layers of sinusoidal units, although in this case the geometry of the approximation is more difficult to understand. Likewise, a sinusoidal activation function g_{sin} may also replace the linear function in the output units of an MLFF net whenever the range of the mapping is restricted to a hypercube $[0,1]^P$.

Third, MLFF nets with sinusoidal units should learn more quickly (less training epochs) than the MLFF nets with sigmoid or Gaussian units, due to the fact that the derivative of the sinusoidal activation function is more suitable to backpropagate error information during training. This argument is quite similar to the one that has been given in the case of training SLRNNs, with the difference that in SLRNNs the net is unfolded in time for backpropagation. By analysing the sensitivity of MLFF network outputs with respect to the weights of the different layers using the chain rule, in a similar way to that described for SLRNNs, it can be seen that a poor sensitivity will be obtained in general if the derivative of the activation function used in the units of the net is (practically) zero for most of the possible net-input values. Moreover, the sensitivity becomes poorer for the weights in more distant layers. Again, while the first derivatives of the sigmoid and Gaussian functions are practically null for the most part of the domain, the derivative of the sine (i.e. the cosine) takes significant values for all points except those very close to the extremes of each period.

From a different point of view, it may be expected that the landscape of the error function with respect to the weights of an MLFF net be much smoother if a sinusoidal activation function is used, thus easing gradient-descent learning, since the plateau regions that are typical in error landscapes of sigmoid-based or Gaussian-based MLFF nets may be avoided. Note that plateau regions in error landscapes basically correspond to situations where all or most of the units are "saturated" (due to large positive or negative net-inputs) for some region of the weight space and a given set of inputs, so that changing the weights in a local neighborhood does not affect significantly to the network outputs and errors. For this reason, the type of modification in the error function that was recommended for Gaussian units to pull net-input values to the Gaussian centers [DaSc:92] is not required in the case of sinusoidal units.

Nevertheless, Lapedes and Farber pointed out that using g_{sin} sometimes leads to numerical problems and nonglobal minima in the learning stage [LaFa:87]. A numerical problem may arise if high frequencies are reached in the units, due to a large magnitude of the unit weight vector. In this case, small weight changes during gradient descent may produce jumps of the net-input value for the same input between different periods of the sine function, thus difficulting a smooth local adjustment of the unit activation value to reduce error. A possible way to alleviate this problem is to "normalize" the net-input value with respect to the unit fan-in, e.g. by setting the constant a in Eq.(6.24) to $\pi/fan\text{-}in$. Another approach is to add a regularization term to the cost function being minimized, which impedes the weights to grow excessively.

Finally, it should be noted that even though an MLFF net with sinusoidal units may learn a trained task more quickly than another MLFF net with sigmoid units and same topology, the sinusoidal-based net may generalize worse than the sigmoid-based net. This is because the MLFF net with sinusoidal units can be more powerful in mapping approximation capability, due to the reasons aforementioned, and therefore, an overfitting of the training set may sometimes occur.

In summary, the learning performance and memory capacity of both SLRNNs and ASLRNNs may improve if the commonly used sigmoid activation function is replaced by other functions. In particular, it has been argued that the antisymmetric logarithm g_{al} may be more adequate for the recurrent hidden units, where the net represents state information, and that the sinusoidal function g_{sin} may be more adequate for the rest of units, which are aimed at yielding some output function from inputs and states. In order to test these conjectures in a benchmark GI problem, some RNNs were trained to learn the next-symbol prediction task for the two Reber grammars studied in some previous works [ClSM:89, SmZi:89, Fahl:91b], using different combinations of activation functions. In addition, both complete and truncated gradient learning algorithms were tried for ASLRNNs to assess the relevance of this choice. The results of this experimental study, which are reported next, confirmed the conjectures raised.

6.2.2 An empirical study on the next-symbol prediction task

6.2.2.1 Architectures and learning algorithms used in the tests

The three types of RNN architectures shown in Fig.6.5 were used for the experimental study. The one on the top left corner corresponds to a fully-connected first-order SLRNN, where some neurons are trained to predict the next-symbol (output units) and the rest are not trained (hidden units). The one on the bottom left corner is a two-layer first-order ASLRNN, which is similar to Elman's SRN, except in the fact that it is trainable with a true gradient-descent learning algorithm. Finally, the one on the right is a three-layer first-order ASLRNN, which is more powerful than the previous ones for learning output functions. Four different combinations of activation functions were tested for each architecture: (sigmoid, sigmoid), (linear, sigmoid), (antisym-log, sigmoid) and (antisym-log, sinusoidal), where, for each pair, the first function was used in the recurrent hidden units, and the second one in the rest of the network units, including the output units. Sigmoid and sinusoidal functions were ranged in $[0, 1]$, i.e. the functions g_s (Eq.(4.5)) and g_{sin} (Eq.(6.24)) were used. The constant parameter a was set as follows: $a = 1$ for g_s and g_{lin}, $a = 2$ for g_{al}, and $a = \pi/fan\text{-}in$ for g_{sin}, respectively.

First-order SLRNNs were trained using the Schmidhuber's efficient RTRL algorithm [Schm:92] described in Section 4.1.1.1, which computes the true error gradient. For first-order ASLRNNs, two learning schemes were tried:

1) a complete gradient-descent algorithm (described in Section 4.1.2), such that backpropagation is used to update the weights of the higher layer units and supply the error values $e_i(t)$ for the recurrent hidden units, and Schmidhuber's RTRL is applied to the first layer; and
2) a truncated gradient-descent algorithm, which simply consists of applying backpropagation to all layers as in Elman's SRNs [ClSM:89], by considering the activation values of the recurrent units in the previous time step as a set of additional input signals (usually called context units [Elman:90]).

In order not to introduce additional variability, the learning rate α and the momentum parameter β were fixed for all the tests: $\alpha = 0.025$, $\beta = 0$.

6.2.2.2 Procedure and results

The two regular grammars displayed in Fig.4.5, the Reber grammars, that had been employed as benchmark in some previous studies on the next-symbol prediction learning task [ClSM:89, SmZi:89, Fahl:91b] were chosen for the tests. First, the simple

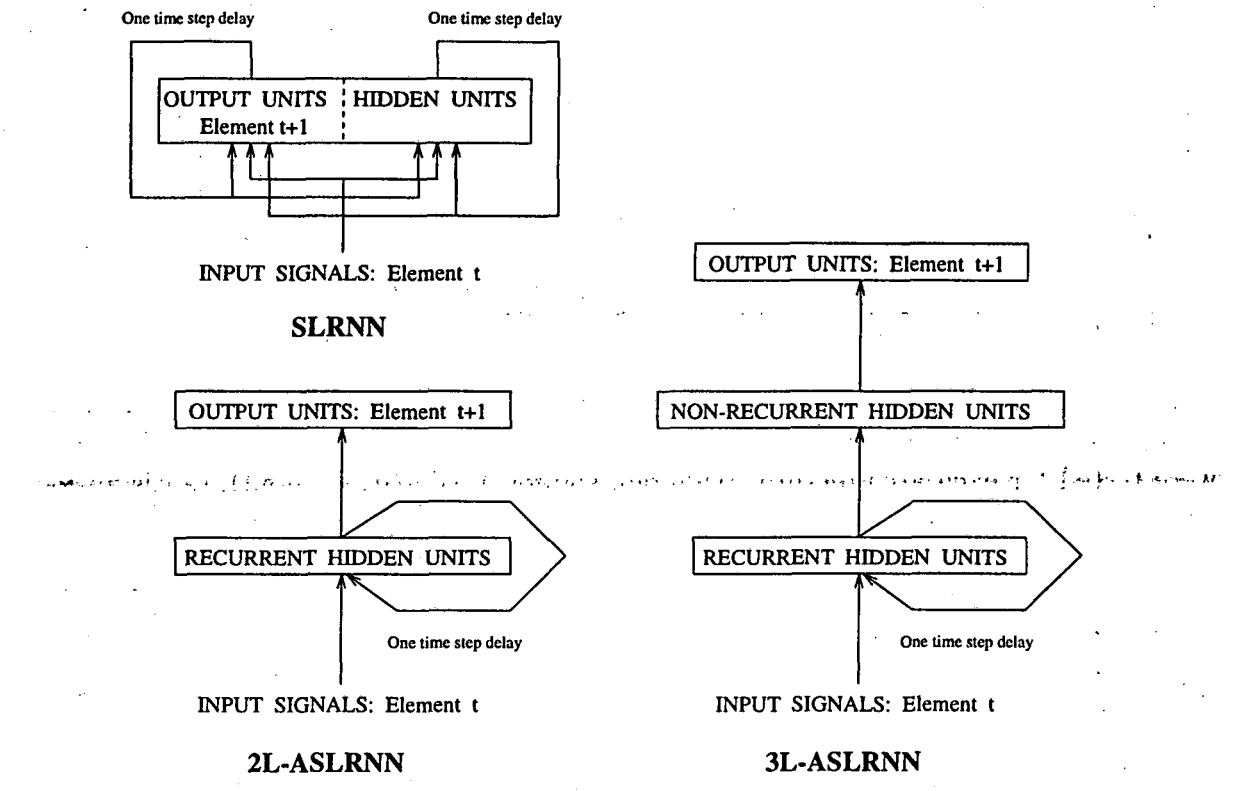

Fig. 6.5 *The three layered RNN architectures (SLRNN, 2L-ASLRNN, and 3L-ASLRNN) used in the empirical study on RGI through next-symbol prediction.*

Reber grammar (Fig.4.5a) was selected to generate random sequences of valid strings, by distributing a uniform probability among the successor symbols of each state.

Three specific network configurations were trained to predict the next symbol in a sequence of legal transitions: network **SLRNN-a**, that consisted of 6 input signals (one for each grammar symbol, including an end-of-string symbol \$) and 9 fully-connected recurrent units (3 hidden and 6 output units, again one for each symbol); network **2L-ASLRNN-a**, that consisted of 6 input signals, 6 recurrent hidden units, and 6 non-recurrent output units; and network **3L-ASLRNN-a**, that had an additional layer of 6 feed-forward hidden units between the recurrent and the output layers. As usual, a local encoding was used to represent the language symbols both in the input signals and the output units, and pairs of consecutive symbols in a sequence of positive strings, each ended with \$, were given for training (see Section 4.2.1).

Likewise, as in previous studies [ClSM:89,SmZi:89], a prediction was considered correct when the activation values of output units associated with the possible successors were above some threshold whereas the rest were below it; the prediction

threshold was set to 0.25. Valid strings were continuously generated for on-line training, but the recurrent-unit activations were reset to 0.1 after the end of each string. Learning was regularly stopped after groups of 2,000 training transitions, when a sequence of 1,000 transitions (around 143 strings in average) were tested for next-symbol prediction. Training ended either when all predictions in a test phase were correct, or when a maximum number of 2 million training transitions (approximately 285,700 strings) were reached without success. Ten trials, with different random initial weights in the interval [-0.25,0.25], were run for each model and each pair of activation functions.

The results of learning the prediction task for the simple Reber grammar are shown in Tables 6.1 and 6.2. Three performance figures are placed inside each entry, from top to bottom: **a)** number of successful runs (up to 10), **b)** mean and standard deviation (over the successful runs), and **c)** minimum and maximum, of the number of training transitions (expressed in thousands) required to reach 0% prediction error.

The results corresponding to the ASLRNNs trained with a truncated gradient approach (backpropagation) are displayed in Table 6.1. We first observe that some of the runs performed using **2L-ASLRNN-a** with only sigmoid units did not converge to a solution (after 2 million transitions). Some of the runs performed using a linear activation function in the recurrent units did not converge either, but in this case, it was due to network unstability during training. The (antisym-log, sigmoid) configurations learnt the task in all runs and around 5 times faster than the networks with only sigmoid units. Furthermore, the (antisym-log, sin) nets learnt 8 and 49 times faster than the all-sigmoid nets, for **2L-ASLRNN-a** and **3L-ASLRNN-a**, respectively. The best result using the truncated gradient computation was obtained by the **3L-ASLRNN-a** net with the (antisym-log, sin) pair of activation functions: an average of 1,970 training strings (13,800 transitions) to reach a solution.

On the other hand, all the 120 runs performed using a true gradient computation were successful for the simple Reber grammar (Table 6.2). Likewise, the number of required training transitions dropped, with respect to the former results, for 2L- and 3L-ASLRNNs and all four combinations of activation functions; this improvement was specially remarkable in the case of the two-layer nets. Here, the (antisym-log, sin) configurations learnt 3, 16 and 20 times faster than the all-sigmoid nets, for **SLRNN-a**, **2L-ASLRNN-a** and **3L-ASLRNN-a**, respectively. The (antisym-log, sigmoid) nets performed rather similarly to the (antisym-log, sin) ones for the SLRNN and two-layer ASLRNN architectures, a little faster learning with **SLRNN-a** and a little slower learning with **2L-ASLRNN-a**; however, a quite large upgrade was obtained again by the sigmoid to sinusoidal replacement in the non-recurrent units of the three-layer ASLRNN (yielding 5 times faster learning). The nets with linear units in the recurrent layer learnt more quickly than the ones with sigmoid units but more slowly

than the ones with the antisym-log activation function. Finally, the fact that 2L-ASLRNNs learnt more quickly than 3L-ASLRNNs can be attributed to the simplicity of the target prediction task, so that a correct generalization of the examples was more rapidly accomplished with a lesser number of neurons.

Rec.hidden-unit Act.Func. / Other-unit Act.Func.	Sigmoid / Sigmoid	Linear / Sigmoid	Antisym-Log / Sigmoid	Antisym-Log / Sinusoidal
Network **2L-ASLRNN-a** 6 i.+ 6 r.h.u.+ 6 o.u.	6 **997.0** ± 451.8 [532 – 1742]	5 **570.0** ± 663.8 [114 – 1700]	10 **205.6** ± 212.3 [22 – 504]	10 **120.4** ± 168.1 [18 – 580]
Network **3L-ASLRNN-a** 6 i.+ 6 r.h.u.+ 6 h.u.+ 6 o.u.	10 **673.4** ± 440.4 [230 – 1668]	8 **98.8** ± 85.9 [44 – 286]	10 **119.8** ± 173.6 [30 – 574]	10 **13.8** ± 5.6 [10 – 28]

Table 6.1 *Learning performance figures for the simple Reber grammar using ASLRNNs trained with backpropagation (truncated gradient).*

Rec.hidden-unit Act.Func. / Other-unit Act.Func.	Sigmoid / Sigmoid	Linear / Sigmoid	Antisym-Log / Sigmoid	Antisym-Log / Sinusoidal
Network **SLRNN-a** 6 i.+ 3 r.h.u.+ 6 r.o.u.	10 **44.0** ± 6.3 [36 – 56]	10 **18.4** ± 2.8 [14 – 24]	10 **12.4** ± 2.8 [8 – 18]	10 **15.4** ± 2.7 [12 – 20]
Network **2L-ASLRNN-a** 6 i.+ 6 r.h.u.+ 6 o.u.	10 **116.6** ± 18.6 [90 – 146]	10 **11.2** ± 2.7 [8 – 16]	10 **9.0** ± 4.0 [6 – 18]	10 **7.2** ± 3.7 [4 – 16]
Network **3L-ASLRNN-a** 6 i.+ 6 r.h.u.+ 6 h.u.+ 6 o.u.	10 **195.0** ± 71.4 [138 – 388]	10 **90.0** ± 74.1 [36 – 284]	10 **49.8** ± 22.6 [18 – 78]	10 **9.4** ± 3.3 [6 – 14]

Table 6.2 *Learning performance figures for the simple Reber grammar using full-gradient computation.*

Rec.hidden-unit Act.Func. / Other-unit Act.Func.	Sigmoid / Sigmoid	Linear / Sigmoid	Antisym-Log / Sigmoid	Antisym-Log / Sinusoidal
Network **SLRNN-b** 6 i.+ 12 r.h.u.+ 6 r.o.u.	0 – –	0 – –	0 – –	6 **909.0** ± 601.4 [362 – 1844]
Network **2L-ASLRNN-b** 6 i.+ 12 r.h.u.+ 6 o.u.	1 **1970.0** [1970]	0 – –	9 **238.4** ± 81.4 [120 – 402]	10 **235.0** ± 122.8 [122 – 480]
Network **3L-ASLRNN-b** 6 i.+ 12 r.h.u.+ 12 h.u.+ 6 o.u.	6 **1487.3** ± 373.0 [980 – 1920]	7 **294.6** ± 56.2 [200 – 358]	10 **265.2** ± 232.3 [102 – 846]	10 **144.0** ± 33.5 [80 – 192]

Table 6.3 *Learning performance figures for the symmetric Reber grammar using full-gradient computation.*

From Tables 6.1 and 6.2, we see that the best result for the simple Reber grammar was achieved by the **2L-ASLRNN-a** net with the (antisym-log, sin) activation functions trained by the true gradient-descent algorithm: an average of 1,030 training strings (7,200 transitions) for fully successful performance. It is interesting to compare this result with those reported elsewhere for the same task and grammar: Cleeremans *et al.* [ClSM:89] reported a best result of 60,000 training strings with 3 hidden units, and 20,000 with 15 hidden units, using a sigmoid-based SRN trained by backpropagation. Smith and Zipser [SmZi:89] reported to learn the task after 19,000 to 63,000 string presentations using a sigmoid-based SLRNN of 2 hidden units and RTRL; Fahlman [Fahl:91b] reported an average of 25,000 string presentations using the constructive recurrent cascade-correlation approach with sigmoid activation functions. In contrast, our two-layer ASLRNN with 6 hidden units, trained by Schmidhuber's RTRL in the recurrent layer, needed an average of 17,000 strings with the sigmoid function, but only 1,030 with the (antisym-log, sin) pair!

Afterwards, the prediction task for the more complex symmetrical Reber grammar (Fig.4.5b) was tested using three larger net structures: network **SLRNN-b**, that consisted of 6 input signals and 18 fully-connected recurrent units (12 hidden and 6 output units); network **2L-ASLRNN-b**, that had the same number number of inputs, hidden and output units than **SLRNN-b**, but with the neurons organized in two layers, a recurrent layer of hidden units and a non-recurrent output layer; and network **3L-ASLRNN-b**, that had an additional layer of 12 feed-forward hidden units.

In order to generate random sequences of positive strings, a uniform probability distribution among the successor symbols of each state was set. This implies that the two embedded simple Reber grammars are completely identical, thus difficulting the prediction task [ClSM:89]. The training and test procedure was the same as before. The first outstanding result was that, using the truncated-gradient computation, the prediction task for the symmetrical Reber grammar could not be learned in any of the 80 runs performed with 2L- and 3L-ASLRNN architectures and distinct activation functions. This demonstrates that using just backpropagation, as in the original Elman's SRN, is a bad choice when the RNN must learn a task which requires to remember an event occurred several time steps ago.

Table 6.3 displays the learning performance measures that were obtained for the symmetrical Reber grammar using the true-gradient computation. Even though larger nets were employed, it can be observed, by comparing with Table 6.2, that the next-symbol prediction task for this grammar was much harder to learn. Indeed, the **SLRNN-b** and **2L-ASLRNN-b** nets with sigmoid or linear activation function in the recurrent hidden units were not able to converge after 2 million training transitions, except for a single run with **2L-ASLRNN-b** and sigmoid units.

Curiously, the (antisym-log, sigmoid) function pair worked quite well for the 2L-ASLRNN architecture, with only 1 failure, but not for the SLRNN, where the 10 runs failed to converge. However, by replacing the sigmoid by a sinusoidal function in the output units of the SLRNN, a 60% success rate was achieved. These and the previously commented results show the bad effect on learning performance that was caused by using a sigmoid activation function in all or part of the recurrent units. It is important to note also that the 2L-ASLRNNs learnt much better and faster than the SLRNNs, even though both models had the same number of neurons, in the cases where the antisym-log activation function was used in the hidden units.

In contrast with Table 6.2, the results of the 3L-ASLRNN architecture for the complex symmetrical Reber grammar outperformed those of the 2L-ASLRNN. For instance, 6 of the 10 runs carried out using **3L-ASLRNN-b** with only sigmoid units converged to a solution, requiring an average of 165,250 strings. Finally, and in accordance with the expectations derived from the theoretical discussion stated in Subsection 6.2.1, the (antisym-log, sin) function pair contributed to the best learning performance for the three architectures tested: 100% success, after an average of 16,000 and 26,000 string presentations, with **3L-ASLRNN-b** and **2L-ASLRNN-b** nets, respectively; and 60% success, after an average of 101,000 string presentations, with **SLRNN-b**. To compare with, the following results were achieved in the previous studies: Cleeremans *et al.* [ClSM:89] reported failure in learning the prediction task on the symmetrical grammar using an SRN with sigmoid function and backpropagation, even with 15 hidden units and 250,000 training strings; Smith and Zipser [SmZi:89] reported that sigmoid-based SLRNNs trained by RTRL learnt the task in some (unspecified) fraction of attempts (best result: 25,000 training strings with 12 hidden units); Fahlman [Fahl:91b] needed an average of 182,000 string presentations using recurrent cascade correlation ($7-9$ hidden units), and perfect learning was achieved in just half the trial runs.

In summary, both for the simple and the symmetrical Reber grammars, the learning performance obtained by using first-order ASLRNNs trained by a true gradient-descent algorithm, with the antisym-log activation function in the recurrent hidden units and a sinusoidal activation function in the rest of the units, greatly improved the results reported in the previous studies by other researchers, where different RNN architectures with sigmoid units were employed.

On the other hand, the FSA extraction algorithm based on hierarchical clustering for only positive examples, that has been described in Section 6.1, was applied to some of the networks reached as solution in the preceding study. In the most part of the trials, the target DFAs of Fig.4.5 were returned. In the rest of cases, similar DFAs were extracted containing one more or one less state. This result may be attributed to the difficulty of setting a distance threshold to stop clustering suitable for every run.

6.3 RGI from positive and negative examples by training RNNs to learn the string classification task: experimental assessment

In this section, the ability of RNNs for RGI from positive and negative examples is investigated. As in several previous works [WaKu:92, MiGi:93, ZeGS:93, MaFa:94, DaMo:94], RNNs were trained to learn the string classification task from sparse samples of some target regular languages. In addition, the UFSA extraction method described in Section 6.1 was applied to each one of the trained networks to obtain a symbolic description of the inference outcome.

The results of the former study on the next-symbol prediction task were taken into account to select the types of RNNs and activation functions used. However, in this new experimental study, the focus was not on comparing the convergence speed of different RNNs to learn the training set, but on assessing the generalization performance of the trained nets on unseen test strings and the corresponding classification rates of the extracted UFSAs.

Although the Tomita's languages [Tomi:82] had been chosen as target regular languages in all the empirical studies reviewed in Section 4.2.2, both the given training sets (samples) and the parameters of the training procedure had varied widely from one work to another. Hence, a precisely defined benchmark was not available to compare with the previous connectionist works on RGI from positive and negative examples. Moreover, as far as we know, the RGI methods based on RNNs had not been confronted yet with symbolic RGI algorithms on common data.

Due to the above reasons, the same 15 test languages, which include Tomita's languages, and sparse samples used as benchmark in Chapter 5 (and in some other studies with symbolic and genetic RGI methods [MiGe:94, Dupo:94]) were selected as input data. In this way, the results produced by the proposed symbolic and connectionist approaches, both with and without UFSA extraction, could be compared.

6.3.1 Architectures and algorithms used in the tests

From the analysis of the results reported in Section 6.2.2, first-order ASLRNNs with the antisym-log activation function in the recurrent units were preferred with respect to first-order SLRNNs or other ASLRNNs with sigmoid units in the recurrent layer. Also, it was decided to use 2L-ASLRNNs instead of more powerful 3L-ASLRNNs to avoid sample overfitting, i.e. to promote the generalization of the given classified examples.

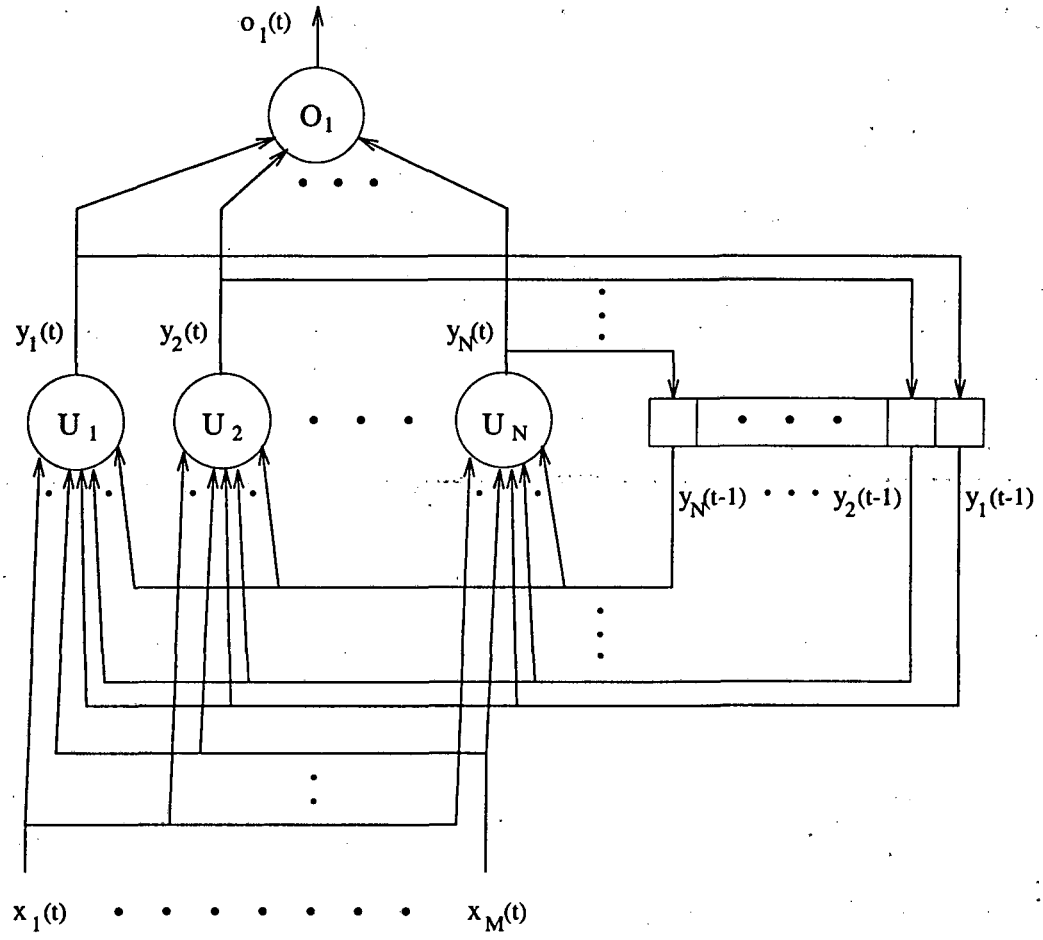

Fig. 6.6 *A generic 2L-ASLRNN architecture for RGI through classification of positive and negative examples.*

More precisely, since a single output unit is enough for the string classification task (recall Section 4.2.2), 2L-ASLRNNs with only one output unit were chosen for the study (see Fig.6.6).

Both sigmoid and sinusoidal activation functions were tested for the output unit, because both (antisym-log, sigmoid) and (antisym-log, sinusoidal) 2L-ASLRNN configurations had learned nicely in the former study on the Reber grammars, and we wanted to assess their respective generalization abilities[11]. Finally, since second-order RNNs had been used by other researchers with rather good results for the same task

[11]Actually, the (antisym-log, sinusoidal) nets had learned somewhat faster, but it was unknown whether they would generalize better or worse than the (antisym-log, sigmoid) nets on the classification task.

[GiMC:92, MiGi:93, DaMo:94], it was also decided to test both first- and second-order 2L-ASLRNNs. The dynamic behavior of 2L-ASLRNNs is described by Eqs. (4.63) and (4.64), together with Eq.(4.65) for first-order type or Eq.(4.66) for second-order type. Note that the 2L-ASLRNN displayed in Fig.6.6 is intentionally ambiguous with regards to the weights of the recurrent layer, so that the figure may represent both first-order and second-order architectures in a generic way.

To sum up, four types of 2L-ASLRNNs, with some number N of recurrent hidden units and 1 output unit, were tested corresponding to first- and second-order 2L-ASLRNNs with the antisym-log activation function g_{al} in the recurrent layer and either a sigmoid g_s or a sinusoidal g_{sin} activation function in the output layer. The constant a in Eqs. (6.18), (4.5), and (6.24), was set respectively to $a = 2$ for g_{al}, $a = 1$ for g_s, and $a = \pi/(N+1)$ for g_{sin}.

A complete gradient-descent learning algorithm was used to train the 2L-ASLRNNs for the classification task, such that backpropagation was applied to update the weights of the output unit and supply the error values $e_i(t)$ $(1 \leq i \leq N)$ for the hidden units, and either the first-order (Section 4.1.1.1) or the second-order (Section 4.1.1.2) version of Schmidhuber's RTRL was applied to train the recurrent layer. The learning rate α and the momentum parameter β were fixed for all the runs: $\alpha = 0.025$ and $\beta = 0.5$. The truncated gradient-descent algorithm for ASLRNNs [Elman:90] was not tried in this case, since the results of the former study clearly showed its inferiority with respect to the complete gradient-descent method.

In order to extract a consistent DUFA from a 2L-ASLRNN previously trained to classify a sample $S = (S^+, S^-)$, the version of Algorithm 6.1 with the "extended_hierarchical_clustering_and_state_merging" function (described in Section 6.1) was used with *onlypos*=FALSE and $k = 50$, followed by a DUFA state minimization. In this way, it was expected that a DUFA in the deterministic border set $DBS_{PTU}(S)$ would be obtained in most cases. For $k = 50$, the extracted DUFA is guaranteed to meet this property if the final number of clusters is ≤ 10, though it could also belong to $DBS_{PTU}(S)$ for a larger number of final clusters.

6.3.2 Procedure and results

The set of 15 regular languages selected by Dupont [Dupo:94, MiGe:94], which include the well-known Tomita's seven languages [WaKu:92, MiGi:93, DaMo:94], were chosen again as target languages. A compact description of each one of these languages has been given in Section 5.5, together with the associated minimal-size DUFA that accepts it and rejects its complement (Figure 5.2).

	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9	L_{10}	L_{11}	L_{12}	L_{13}	L_{14}	L_{15}
Neurons	3	3	4	3	3	3	4	3	4	4	3	3	3	3	4
Pos.Ex.	4	4	31	19	10	9	41	4	15	10	13	6	8	5	11
Diff.P.Ex.	4	4	24	15	10	9	31	4	10	8	13	4	8	5	8
Neg.Ex.	12	22	48	14	28	18	18	17	54	79	20	38	10	27	65
Diff.N.Ex.	12	17	46	14	23	15	17	14	44	63	17	31	9	22	55
Tot.Ex.	16	26	79	33	38	27	59	21	69	89	33	44	18	32	76
Diff.T.Ex.	16	21	70	29	33	24	48	18	54	71	30	35	17	27	63

Table 6.4. *Number of recurrent hidden units in the trained ASLRNNs and average number of examples for each one of the test languages.*

Likewise, the same 10 samples of each language that were used in the experiment reported in Section 5.5.1 were also employed here. Each sample was structurally complete with respect to the corresponding minimal-size DUFA. To this end, the original benchmark samples generated by Dupont were slightly modified just to include the empty string λ in S^+ or S^- appropriately. In addition, ten negative samples were generated for language $L_1 = a^*$, since only positive samples had been defined for L_1 by Dupont[12] [Dupo:94].

Table 6.4 shows the average number of positive, negative, and total examples in the training sets of each test language. It also displays the average number of different positive, negative, and total examples, since most of the benchmark samples contained repeated strings, due to the random way they were generated [Dupo:94]. Larger training sets had been used in most of the previous works with Tomita's languages and RNNs [MiGi:93, ZeGS:93, DaMo:94], whereas the Tomita's original sets of strings used in [Poll:91, WaKu:92] were even smaller (specially for languages L_3 and L_7).

In addition, Table 6.4 shows the number N of recurrent hidden units that were included in both the first- and second-order networks for each target language. A small N was preferred to favour generalization, and thus, N was set to 3 or 4, depending on the size of the minimal DUFA associated with the language (see Figure 5.2). Two trials, with different random initial weights in the interval [-0.25,0.25], were run for each training set and each type of 2L-ASLRNN tested, so that a total number of 20 runs were performed with each of the four network models for each of the fifteen test languages.

Each network was trained to classify the strings in the given training set, following the normal procedure described in the beginning of Section 4.2.2. Thus, a set of

[12]Because he considered L_1 to be defined over a unary alphabet, to the contrary of Tomita's L_1, which was defined over a binary alphabet.

string-response pairs were given to the net at each training epoch, where the response corresponds to the target value of the output unit at the last time step of the string presentation, which was defined as $T_{accept} = 1$ for positive strings and $T_{reject} = 0$ for negative strings. As in the previous studies, a string was considered to be correctly classified by the net if the absolute difference between the desired response and the final activation value of the output unit was less than a tolerance threshold ϵ.

A local encoding was used to represent the language symbols in the input signals. The recurrent-unit activations were reset to 0.1 at the start of each string presentation, and no end-of-string symbol was used. The training phase ended when all the strings in the training set were correctly classified, using $\epsilon = 0.1$ as tolerance threshold, and a minimum number of 1,000 epochs were performed. The requirement of a minimum number of training epochs was aimed at giving the net enough time to contract the clusters formed in the hidden unit activation space [ZeGS:93, MaFa:94]. Then, a consistent DUFA was extracted from the final net dynamics on the given sample by applying the clustering algorithm aforementioned.

The same test samples that had been used in the experiment of Section 5.5.1 (and in [Dupo:94]) were again employed. This is, for each run, all the strings up to length l but the given training examples were included, where l was set to 9 or 7 for target languages over $\{a, b\}$ or $\{a, b, c\}$, respectively. After training, the test strings were classified both by the final net and the extracted DUFA. A test string was accepted by a trained net if the final activation value of the output unit after string presentation was greater than 0.5, and it was rejected otherwise (i.e. $\epsilon = 0.5$ was used in this step). On the other hand, a test string was accepted by an extracted DUFA if it drove the automaton to a positive final state, and it was rejected otherwise. For both 2L-ASLRNNs and extracted DUFAs, the correct classification rates on the three sets of positive, negative, and all test strings were computed.

For each language and each type of inference, the averages of the above rates over the 20 runs were calculated, and they are displayed in the former three wide columns of Tables 6.5 to 6.8. The fourth wide column refers to the arithmetic mean of the positive and negative classification rates. In Tables 6.5 and 6.6, the fifth wide column shows, for each language and type of 2L-ASLRNN, the percentage of times the whole test sample was correctly classified by the trained net (success rate); whereas in Tables 6.7 and 6.8, it shows the percentage of times the target DUFA was extracted (identification rate). The bottom row of the tables displays the averages of these five features over the 15 test languages.

F.O. nets	Pos.class		Neg.class		Tot.class		Av.class		Success	
	Sigm	Sin	Sigm	Sin	Sigm	Sin	Sigm	Sin	Sigm	Sin
L_1	90.0	70.2	99.6	91.1	99.5	90.8	94.8	80.7	45.0	25.0
L_2	92.1	90.5	92.6	84.0	92.6	84.1	92.4	87.3	20.0	0.0
L_3	53.2	58.1	79.3	76.4	68.8	68.9	66.3	67.3	0.0	0.0
L_4	66.6	67.2	67.6	72.9	67.0	69.5	67.1	70.1	0.0	0.0
L_5	49.0	41.8	74.1	76.7	69.9	70.9	61.6	59.3	0.0	0.0
L_6	37.1	42.1	63.8	58.4	54.9	53.0	50.5	50.3	0.0	0.0
L_7	59.2	57.1	60.7	71.1	60.1	66.0	60.0	64.1	0.0	0.0
L_8	82.3	68.6	89.6	78.3	89.5	78.1	86.0	73.5	10.0	0.0
L_9	87.7	81.7	92.9	86.4	92.9	86.4	90.3	84.1	0.0	0.0
L_{10}	46.0	51.3	93.7	90.6	92.7	89.8	69.9	71.0	0.0	0.0
L_{11}	69.2	60.7	55.8	51.9	60.3	54.8	62.5	56.3	0.0	0.0
L_{12}	94.3	76.2	96.4	95.3	96.3	95.1	95.4	85.8	0.0	0.0
L_{13}	50.7	50.7	51.1	60.3	50.9	55.5	50.9	55.5	0.0	0.0
L_{14}	39.1	45.7	85.2	78.3	83.7	77.2	62.2	62.0	0.0	0.0
L_{15}	58.0	43.3	89.7	83.4	89.5	83.1	73.9	63.4	0.0	0.0
Mean	65.0	60.3	79.5	77.0	77.9	74.9	72.3	68.7	5.0	1.7

Table 6.5. *Classification results of trained first-order ASLRNNs, with an antisymmetric logarithm activation function in the recurrent units and either a sigmoid or a sinusoidal activation function in the output unit.*

S.O. nets	Pos.class		Neg.class		Tot.class		Av.class		Success	
	Sigm	Sin	Sigm	Sin	Sigm	Sin	Sigm	Sin	Sigm	Sin
L_1	86.0	68.6	96.5	86.1	96.4	85.9	91.3	77.4	35.0	10.0
L_2	84.5	85.9	89.7	82.0	89.7	82.0	87.1	84.0	10.0	0.0
L_3	78.5	63.9	72.9	75.1	74.9	70.6	75.7	69.5	0.0	5.0
L_4	74.7	52.1	58.8	78.3	68.0	63.0	66.8	65.2	0.0	0.0
L_5	52.4	39.7	76.6	64.2	72.5	60.1	64.5	52.0	5.0	0.0
L_6	44.0	46.4	60.4	61.3	54.9	56.3	52.2	53.9	0.0	0.0
L_7	77.4	67.9	45.5	54.8	57.0	59.5	61.5	61.4	0.0	0.0
L_8	84.6	65.1	72.0	74.3	72.2	74.1	78.3	69.7	0.0	5.0
L_9	90.0	73.9	81.1	80.1	81.2	80.1	85.6	77.0	0.0	0.0
L_{10}	53.7	53.0	93.4	85.4	92.6	84.8	73.6	69.2	0.0	0.0
L_{11}	76.0	60.5	70.2	52.2	72.1	55.0	73.1	56.4	5.0	0.0
L_{12}	88.3	90.7	90.7	90.7	90.6	90.7	89.5	90.7	5.0	15.0
L_{13}	81.1	60.0	81.1	62.5	81.1	61.3	81.1	61.3	45.0	5.0
L_{14}	65.3	56.3	79.7	76.5	79.2	75.8	72.5	66.4	0.0	0.0
L_{15}	75.9	69.1	84.5	80.7	84.4	80.6	80.2	74.9	0.0	0.0
Mean	74.2	63.5	76.9	73.6	77.8	72.0	75.5	68.6	7.0	2.7

Table 6.6. *Classification results of trained second-order ASLRNNs.*

F.O. DUFA	Pos.class		Neg.class		Tot.class		Av.class		Identif.	
	Sigm	Sin	Sigm	Sin	Sigm	Sin	Sigm	Sin	Sigm	Sin
L_1	100.0	95.0	100.0	99.6	100.0	99.5	100.0	97.3	100.0	90.0
L_2	96.7	96.6	96.4	91.5	96.4	91.6	96.6	94.1	55.0	15.0
L_3	47.3	52.0	89.5	82.0	72.8	70.0	68.4	67.0	0.0	0.0
L_4	55.6	60.2	84.5	78.0	67.6	67.6	70.1	69.1	5.0	5.0
L_5	38.3	38.9	82.4	82.0	75.0	74.9	60.4	60.5	0.0	0.0
L_6	31.0	33.9	70.8	69.9	57.5	57.9	50.9	51.9	0.0	0.0
L_7	53.8	53.8	71.3	77.7	64.9	69.0	62.6	65.8	0.0	5.0
L_8	100.0	88.4	96.4	91.0	96.5	90.9	98.2	89.7	70.0	45.0
L_9	96.0	97.7	95.6	92.2	95.6	92.2	95.8	95.0	5.0	0.0
L_{10}	54.2	62.7	93.5	92.4	92.7	91.7	73.9	77.6	0.0	0.0
L_{11}	59.8	48.2	62.9	66.9	61.8	60.7	61.4	57.6	0.0	0.0
L_{12}	100.0	92.7	97.6	97.9	97.6	97.9	98.8	95.3	40.0	40.0
L_{13}	41.3	40.5	71.4	74.5	56.3	57.5	56.4	57.5	10.0	5.0
L_{14}	39.3	50.5	91.1	86.2	89.5	85.0	65.2	68.4	0.0	0.0
L_{15}	72.1	63.0	93.2	91.3	93.0	91.1	82.7	77.2	0.0	0.0
Mean	65.7	64.9	86.4	84.9	81.1	79.8	76.1	74.9	19.0	13.7

Table 6.7. *Classification results of DUFAs extracted from trained first-order ASLRNNs for the fifteen test languages.*

S.O. DUFA	Pos.class		Neg.class		Tot.class		Av.class		Identif.	
	Sigm	Sin	Sigm	Sin	Sigm	Sin	Sigm	Sin	Sigm	Sin
L_1	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
L_2	93.7	85.0	94.4	88.5	94.4	88.4	94.1	86.8	25.0	5.0
L_3	59.4	51.6	85.3	85.5	75.0	72.1	72.4	68.6	5.0	5.0
L_4	53.8	44.3	82.4	79.8	65.7	59.1	68.1	62.1	10.0	0.0
L_5	42.4	27.1	82.8	77.5	76.1	69.1	62.6	52.3	15.0	0.0
L_6	31.6	34.7	76.1	73.3	61.2	60.4	53.9	54.0	0.0	5.0
L_7	61.4	59.9	71.0	75.8	67.4	70.0	66.2	67.9	10.0	5.0
L_8	90.0	89.5	81.1	87.6	81.2	87.6	85.6	88.6	10.0	30.0
L_9	91.7	89.4	88.8	88.2	88.8	88.2	90.3	88.8	0.0	0.0
L_{10}	55.6	49.9	94.7	93.4	93.9	92.4	75.2	71.7	0.0	0.0
L_{11}	65.3	48.2	75.8	66.9	72.3	60.6	70.6	57.6	20.0	5.0
L_{12}	96.5	100.0	93.9	96.2	93.9	96.2	95.2	98.1	30.0	40.0
L_{13}	78.7	57.9	87.0	78.0	82.9	68.0	82.9	68.0	65.0	30.0
L_{14}	61.8	56.3	83.7	87.8	83.0	86.8	72.8	72.1	0.0	10.0
L_{15}	75.3	78.2	93.4	92.6	93.2	92.5	84.4	85.4	0.0	0.0
Mean	70.5	64.8	86.0	84.7	81.9	79.4	78.3	74.8	19.3	15.7

Table 6.8. *Classification results of DUFAs extracted from trained second-order ASLRNNs for the fifteen test languages.*

The results of the generalization tests for the trained first- and second-order 2L-ASLRNNs are presented in Tables 6.5 and 6.6, respectively. It can be observed that both architectures performed rather similarly. On the other hand, the nets with the sigmoid output unit (left) generalized better than those with the sinusoidal output unit (right): a 3% and a 7% better average classification rate for first- and second-order nets, respectively. Note also that, even for the model with best results (the second-order ASLRNNs with sigmoid output unit), all global classification rates were below the 80%; furthermore, the success rate was very poor (7%), since for most of the languages, a perfect classification of the test sample was not achieved in any of the 20 runs.

The classification results of the DUFAs extracted from first- and second-order 2L-ASLRNNs are presented in Tables 6.7 and 6.8, respectively, together with the identification rates of the target DUFAs. In general, the inferred DUFAs performed better than the trained nets from which they were obtained, improving the global average classification rates between a 3% and a 6%. The best results were yielded by the DUFAs extracted from second-order 2L-ASLRNNs with sigmoid output unit, with 78.3% average correct classification and 19.3% identification rate. The quality of the extracted DUFAs was similar for first- and second-order nets, and somewhat better for the nets with a sigmoid output unit than for the nets with a sinusoidal unit. For all the test languages except L_{10} and L_{15}, the target DUFA was inferred in at least one run for some of the tested ASLRNNs, but the identification rates were poor for most of the languages.

For comparison purposes, Table 6.9 shows the summary of the results obtained by the eight "connectionist" RGI methods tested in the experiment (i.e. the four pure connectionist methods returning a 2L-ASLRNN, and the associated four hybrid methods returning a DUFA), together with those obtained by the four symbolic RGI methods tested in Section 5.5. As can be observed, both with and without DUFA extraction, the inference quality shown by the connectionist methods was notably worse than the one shown by three of the four symbolic methods, where the exception was the incremental SM_2 algorithm (split and merge with minimal splitting). The average correct classification rates were also worse than the ones reported by Dupont for his genetic RGI approaches (85.4% and 94.4% for the non-incremental and semi-incremental methods, respectively) [Dupo:94].

The best results in Table 6.9 correspond to the symbolic RPNI algorithm [OnGa:92b] (which is labeled "non-incremental max.pos." in the table) with total and average correct classification rates of 96.0% and 94.7%, respectively, and a high identification rate of 78.0%. However, it might be argued that the benchmark samples were specially suitable to the RPNI method, since they often included the representative samples that allowed the identification of the target automata by this algorithm. On the other hand, the small number of examples in the training sets (an

RGI METHOD	Pos.class	Neg.class	Tot.class	Av.class	Success
F.O. ASLRNNs (An.log, Sigm.)	65.0	79.5	77.9	72.3	5.0
S.O. ASLRNNs (An.log, Sigm.)	74.2	76.9	77.8	75.5	7.0
F.O. ASLRNNs (An.log, Sin)	60.3	77.0	74.9	68.7	1.7
S.O. ASLRNNs (An.log, Sin)	63.5	73.6	72.0	68.6	2.7
DUFAs F.O.nets (An.log, Sigm.)	65.7	86.4	81.1	76.1	19.0
DUFAs S.O.nets (An.log, Sigm.)	70.5	86.0	81.9	78.3	19.3
DUFAs F.O.nets (An.log, Sin)	64.9	84.9	79.8	74.9	13.7
DUFAs S.O.nets (An.log, Sin)	64.8	84.7	79.4	74.8	15.7
Non-incremental Max.Pos.	91.8	97.5	96.0	94.7	78.0
Non-incremental Max.P-N	90.7	90.2	90.0	90.5	40.7
Incremental SM_1 Max.P-N	90.6	90.1	89.9	90.4	40.0
Incremental SM_2 Max.P-N	72.6	74.6	72.5	73.6	10.0

Table 6.9. *Summary of results for the RGI experiments from positive and negative sparse samples.*

average of 37 different strings) might be insufficient, from the statistical point of view, to allow the networks generalize correctly, this being probably the cause of the middling results obtained.

The preceding argument would be in agreement with the results reported by Watrous and Kuhn for the inference of Tomita's languages from the small Tomita's sets that contained positive and negative examples (around 20 strings in each set) [WaKu:92]. Using a similar network size (3 hidden units and 1 output unit) in their second-order SLRNN architecture, a sigmoid activation function in all units, and a lower tolerance threshold ($\epsilon = 0.1$) for classification, the trained nets achieved only an average of 56.1% total correct classification rate for the seven languages, ranging from 23.2% for L_6 to 91.2% for L_2. Moreover, Watrous and Kuhn presented further results showing that by increasing the number of strings in the training set from 20 to 200, the total correct classification rate for L_4 augmented from 42.5% to 98.2% [WaKu:92]. We conjecture that a similar high generalization performance might be achieved by the 2L-ASLRNNs tested here by increasing enough the size of the training set.

This hypothesis is supported by other experimental results reported in some previous studies on RGI using RNNs [MiGi:93, DaMo:94]. Miller and Giles [MiGi:93] reported that second-order SLRNNs with $N = 4$ units (including the output unit) learned to classify the $1,023$ binary strings up to length 9 used as training sets for the 7 Tomita's languages, and then they achieved a 99.9% total correct classification rate on the strings of lengths 10 to 15, using $\epsilon = 0.5$ as tolerance threshold in the generalization test. The first-order SLRNNs with $N = 4$ did not learn the training set in some of the runs, but the networks that converged also reached a 99.9% generalization performance. Similar results were obtained for $N = 5, .., 9$ for both architectures [MiGi:93].