Recurrent Neural Networks Can Learn Simple, Approximate Regular Languages*

Mikel L. Forcada and Antonio M. Corbí-Bellot,
{mlf,acorbi}@dlsi.ua.es
Departament de Llenguatges i Sistemes Informàtics,
Universitat d'Alacant, E-03071 Alacant, Spain.

and

Marco Gori and Marco Maggini,
marco@dii.unisi.it, maggini@ing.unisi.it
Dipartimento di Ingegneria dell'Informazione,
Università degli Studi di Siena,
via Roma, 56, I-53100 Siena, Italy.

Abstract

A number of researchers have shown discrete-time recurrent neural networks (DTRNN) to be capable of inferring deterministic finite automata (DFA) from sets of example and counterexample strings; however, discrete algorithmic methods are much better at this task and clearly outperform DTRNN in terms of space and time complexity. We show, however, how DTRNN may be used to learn not the *exact* language that explains the whole learning set but an *approximate* and much simpler language that explains a great majority of the examples by using simpler rules. This is accomplished by gradually varying the error function in such a way that the DTRNN is eventually allowed to classify clearly but incorrectly those strings that it has found to be difficult to learn, which are treated as exceptions. The results show that, in this way, the DTRNN usually manages to learn a simplified approximate language.

1 Introduction

Discrete-time recurrent neural networks (DTRNN) may be trained to to recognize regular languages from sets of example and counterexample strings [3, 11, 5, 13, 7, 4], under the intuitive assumption that, being state machines themselves, DTRNN can emulate deterministic finite automata (DFA), that is, regular-language recognizers. DTRNN can be indeed trained to behave as simple DFA but only at a considerable expense of resources, and their learning behavior does not nicely scale to larger DFA. On the other hand, very efficient discrete algorithms exist [10] that are capable of performing the same task in polynomial time.

In view of this, one might contend that there is no use in pursuing the study of DTRNN as adaptive language recognizers. But if one is interested in obtaining not an *exact* DFA that recognizes all the strings in the learning set, which may be quite complex, but a small, *approximate* DFA which explains all but a few examples, then the poor scaling behavior of DTRNNs and their bias toward very simple representations, which could be otherwise considered as weaknesses, may be used advantageously to accomplish the task of obtaining a simple and approximate language.

This approach has already been explored by Gori and coworkers [6], who view the training set as a *noisy* or *corrupted* version of a hypothetical *clean* learning set for a very simple language; this "noise" only affects the membership of the strings: a few counterexamples are turned into examples and vice versa. When a DTRNN learns to behave as a DFA, its state space tends to organize in clusters of low dimensionality that correspond to the states of the DFA: a clustering algorithm is used in [6] to define these states and extract the learned DFA. A promising alternative approach is described in [8]: an additional term is added to the error function to enforce weight configurations that ensure correct DFA behavior of the DTRNN.

*Work supported through grants TIC97-0941 and HI1996-0055 of the Spanish government.

In this work, instead of focusing on the extraction algorithm, we propose the use of an error function that varies during the learning process in such a way that the net is eventually allowed to classify clearly but incorrectly those strings that are difficult to learn: these strings are treated as exceptions.

2 Definitions and methods

Regular languages and finite automata: A *language* is a set of finite-length strings over a finite alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$. The length of a string w will be denoted $|w|$. A *deterministic finite automaton* (DFA) is defined as a 5-tuple $M = (Q, \Sigma, \delta, q_I, F)$, where Q is a finite number of *states*, Σ is the input alphabet, δ is the next-state function $\delta : Q \times \Sigma \to Q$ which defines which state $q' = \delta(q, \sigma)$ is reached by the automaton after reading symbol σ when in state q, $q_I \in Q$ is the initial state of the automaton (before reading any string) and $F \subseteq Q$ is the set of accepting states of the automaton. The language $L(M)$ accepted by the automaton contains all the strings that bring the automaton to an accepting state. The languages accepted by DFA are called *regular languages*.

Second-order DTRNN as string processors: We have used a second-order DTRNN identical to those used in [5] and [4], having n_U inputs and n_X state units. The value of the k-th input at time t is denoted by $u_k[t]$; the state of the i-th state unit *after* processing the input at time t is denoted by $x_i[t]$. The dynamics of the network is defined by

$$x_i[t] = g\left(\sum_{j=1}^{n_X}\sum_{k=1}^{n_U} W_{ijk} x_j[t-1] u_k[t]\right) \qquad (1)$$

where $i = 1, \ldots, n_X$ and $g(x) = 1/(1+\exp(-x))$. When a DTRNN is used to process strings, the number of input units n_U is set to $|\Sigma|$, the size of the alphabet and a local (exclusive or *one-hot*) encoding is used so that, for each input symbol, the next state is computed by a different single-layer perceptron.

Criteria for acceptance: After complete processing of a string w, a value of $x_1[|w|]$ close to 1 denotes acceptance and a value close to 0, rejection; two different criteria are used: the *tight* criterion (rejection: $x_1[|w|] \in]0, 2\tau]$, acceptance: $x_1[|w|] \in [1-2\tau, 1[$) and the *loose* criterion (rejection: $x_1[|w|] < 0.5$; acceptance: $x_1[|w|] > 0.5$).

Second-order DTRNN behaving as DFA: Second-order DTRNN are capable of emulating the behavior of any finite-state machine for strings of any length, provided that $n_X = |Q| + 1$ units are used and adequate weight values are chosen[1, 9] (even smaller second-order DTRNN may sometimes also emulate some DFA, [2]). It may be therefore expected that small DTRNN can only stably emulate and furthermore, can only easily *learn* small DFA, although the nature of the inductive bias of gradient-based learning algorithms is not well known.

Training algorithms and error functions: We use a gradient-based method, the *real-time recurrent learning* (RTRL) [14], but this choice is not critical to our study. We train the weights as in [5] and the initial state $x_i[0]$, $i = 1, \ldots, n_X$ as in [4]. When DTRNN are trained to emulate DFA, the usual error function for each string w is $0.5(x_1[|w|] - T_w)^2$, where the target T_w, is either τ (rejection) or $1 - \tau$ (acceptance). A training run is finished when all $|x_1[|w|] - T_w| < \epsilon$ (here, $\epsilon = \tau$, but any $\epsilon \leq \tau$ is adequate).

Learning a difficult language: Consider a learning set in which a large majority of strings are labeled following the rules of a simple DFA whereas a small number of them do not follow these simple rules (their membership labels are reversed). When one presents this learning set to a small second-order DTRNN, it seems that the net initially easily learns the correct behavior for a large majority of strings, but fails to classify clearly and correctly a small number of them; a study of the regions of state space visited by the DTRNN indicates the presence of clusters that may be assumed to correspond to the states of the simple DFA classifying the majority of strings [6]. Later, when trying to learn the labels of this smaller number of strings, the clusters dissolve and the generalization performance (error rate) of the networks when processing longer strings degrades, possibly because the DTRNN is incapable of representing (or learning) a more complex DFA.

An error function that varies during learning: Accordingly, we propose the following scheme: the error function will change when the network classifies clearly (even if not correctly) a fraction of the training set whose size is equal (or close) to the expected number of correct strings. The new error function will allow strings to be *incorrectly* but *clearly* classified by having two minima: one at the correct target T_w and another one at the reversed target $1 - T_w$. This is equivalent to performing unsupervised learning at the later stage of the learning process. It is assumed that strings that

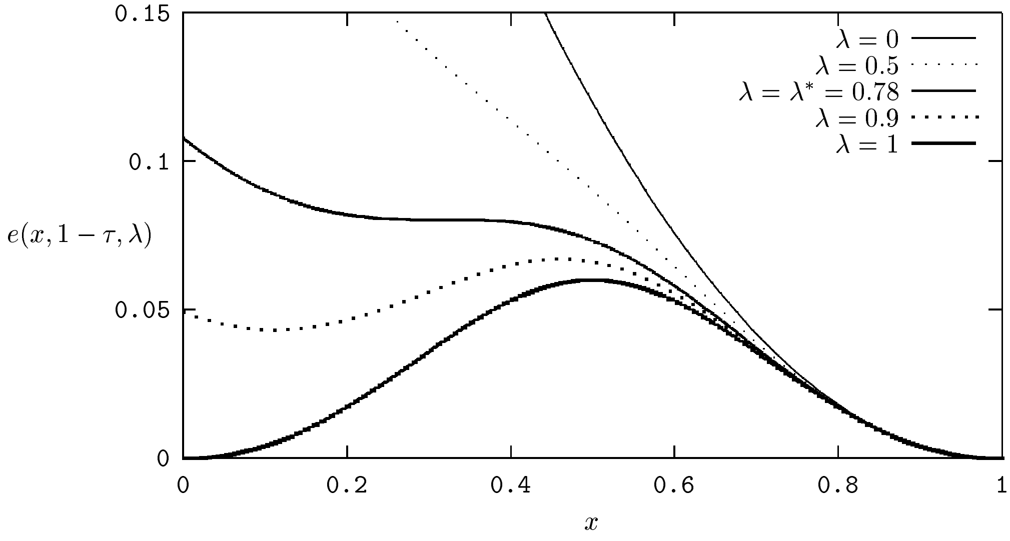

Figure 1: The error function (eq. 4) for $\tau = 0.01$ as a function of λ.

are already correctly classified will not abandon the minimum around the correct target, whereas strings incorrectly classified will "fall" in either of the two target values, now that they are given an additional choice. The transition to the new function is gradual, to avoid disturbing the gradient descent algorithm. The initial (completely supervised) error function is

$$e_s(x, T_w) = \frac{1}{2}(x - T_w)^2 \tag{2}$$

and the final error function (completely unsupervised) is

$$e_u(x) = \frac{1}{2}\frac{(x-\tau)^2 (x-(1-\tau))^2}{(x-\tau)^2 + (x-(1-\tau))^2} \tag{3}$$

which has two minima at $x = \tau$ and $x = 1 - \tau$ and a maximum at $x = 0.5$ (see figure 1). The transition is modelled by using an increasing parameter $\lambda \in [0, 1]$ to combine both functions:

$$e(x, T_w, \lambda) = (1-\lambda)e_s(x, T_w) + \lambda e_u(x). \tag{4}$$

The transition between one minimum and two minima occurs for $\lambda = \lambda^*(\tau)$; in our experiments, $\lambda^*(\tau) \approx 0.78$ for values of τ in $[0.01, 0.1]$. To isolate exceptions, we want this transition to occur when the net has learned to classify clearly a fraction of the strings similar to the expected fraction of correct strings. For this purpose, if $n_{\rm train}$ is the number of strings in the training set, $n_{\rm exc}$ the expected number of exceptions, and $n_{\rm clear}$ the number of strings that have been clearly classified ($n_{\rm clear}$), one sets $\lambda = (n_{\rm clear}/n_{\rm train})^z$ with z chosen so that $\lambda = \lambda^*$ when $n_{\rm clear} = n_{\rm train} - n_{\rm exc}$:

$$z = \frac{\log(\lambda^*)}{\log\left(\frac{n_{\rm train} - n_{\rm exc}}{n_{\rm train}}\right)}. \tag{5}$$

The gradient of the error function with respect to weights and initial states is computed without taking into account the small indirect effect of weights on λ through $n_{\rm clear}$, which is a discontinuous function of weights itself. We have found this approximation to be reasonable.

3 Experiments

The results of two sets of experiments are performed. The first set of experiments evaluates the ability of the new method to recover the underlying clean language from a corrupted training set. The second set focuses on the ability of extracting simple languages from corrupted samples, regardless of whether the original language is recovered or not.

Recovering the original language: For each one of Tomita's [12] seven simple languages, a sample of all membership-labeled strings with length 1–9 is generated. Then, it is corrupted (in each simulation) by flipping (reversing) the membership of strings chosen at random according to a uniform distribution with a flip probability r (r=0.01, 0.02, 0.03 and 0.05); these strings are marked as exceptions for further analyses. The strings with length 1–6 are used for training and the rest are the test set; n_X is set to the smallest number of hidden units preliminarily observed to converge on sets having no exceptions (2 for L_1; 3 for L_2, L_3, L_4 and

L_7; 4 for L_5 and 5 for L_6). For each value of r, 30 runs are performed. The number of exceptions found in each training set ($n_{\rm exc}$) follows the binomial distribution, which is rather broadly spread around the expected value $\langle n_{\rm exc}\rangle = rn_{\rm train}$, with $n_{\rm train}$ the number of strings in the training set.

The learning rate and learning momentum are fixed to $\alpha = 1.0$ and $\eta = 0.2$ respectively, and a run is considered to succeed if the DTRNN recovers the uncorrupted language, that is, it (a) it learns to correctly label (tight criterion) the learning set according to the uncorrupted language; (b) does so in less than 2000 epochs, and (c) correctly labels (loose criterion) the test set. Table 1 shows, for each language and exception rate, the fraction of succeeding jobs, the average number of epochs and its standard deviation. The above criteria for success are very stringent, because they accept only those DTRNN that have captured a stable representation of the original DFA. In a real application, the DFA is not known and exceptions have to be *discovered* by the net by relying on an estimation of the exception rate. Indeed, some jobs isolated a different set of exceptions from the training set (usually less than the ones marked as such), discovering a different DFA with a remarkable generalization performance. Other jobs isolate a very large number of exceptions (25–35), sometimes very early, due to the nature of the stopping criterion (*clear* as opposed to *correct* classification).

Learning an approximate language: The second set of experiments simply probes the ability of the DTRNN to learn simple DFA from the samples. Training is less stringent: a run succeeds if the net learns to label (tight criterion) the training set regardless of the *relabeling* produced, in less than 2000 epochs. To monitor the ability of the DTRNN to learn a simple DFA we use a discrete state-merging DFA inference algorithm, RPNI, [10], on both the corrupted and the DTRNN-relabeled training set. Table 2 shows, for each language and exception rate, the number of succeeding jobs and the average and standard deviation of (a) the number of states RPNI produces for the corrupted and the relabeled training set ($|Q|_{\rm before}$ and $|Q|_{\rm after}$ respectively); the number $n_{\rm exc}$ of exceptions isolated; and the sum of the two preceding values, $n_{\rm exc}+|Q|_{\rm after}$, a measure of approximation (also used in [6]). As can be seen, $|Q|_{\rm after}$ is usually smaller than $|Q|_{\rm before}$, and in many cases it approaches the size of the original (Tomita) DFA.

4 Concluding remarks

The results show that DTRNN can be used to infer simple, approximate DFA describing a majority of the examples when the sample cannot be described by a simple DFA due to the presence of noise in the membership labels or to the fact that the sample indeed does not correspond to any simple DFA. Our method, which uses an error function that changes during training to accomodate exceptions, only needs a hint about the expected exception rate in the learning set; when this rate is not known, the parameter may be used to roughly regulate the quality of the approximation. We plan to study the stability of the DFA representation learned by the DTRNN.

References

[1] R.C. Carrasco, M.L. Forcada, M.A. Valdés, R.P. Ñeco. Stable encoding of finite-state machines in discrete-time recurrent neural nets with sigmoid units. Submitted to *Neural Computation*.

[2] R.C. Carrasco, J. Oncina and M.L. Forcada (1999). Efficient encoding of finite automata in discrete-time recurrent neural networks. In *Proc. ICANN'99* (to appear).

[3] A. Cleeremans, D. Servan-Schreiber, J.L. McClelland. Finite state automata and simple recurrent networks. *Neural Computation*, 1(3):372–381, 1989.

[4] M.L. Forcada, R.C. Carrasco. Learning the initial state of a second-order recurrent neural network during regular-language inference. *Neural Computation*, 7(5):923–930, 1995.

[5] C.L. Giles, C.B. Miller, D. Chen, H.H. Chen, G.Z. Sun, Y.C. Lee.Learning and Extracting Finite State Automata with Second-Order Recurrent Neural Networks. *Neural Computation*, 4(3):393–405, 1992.

[6] M. Gori, M. Maggini, G. Soda. Inductive inference from noisy examples using the hybrid finite state filter. *IEEE Transactions on Neural Networks* 9(3):571–575, 1998.

[7] P. Manolios, R. Fanelli. First order recurrent neural networks and deterministic finite state automata. *Neural Computation* 6(6):1154–1172, 1994.

[8] R.P. Ñeco, S.C. Kremer, M.L. Forcada. Constrained second-order recurrent networks for finite-state automata induction. In *Proc. ICANN'98*, v. 2, p. 529–534.

[9] C.W. Omlin and C.L. Giles. Constructing Deterministic Finite-State Automata in Recurrent Neural Networks. *Journal of the ACM* 43(6):937-972, 1996.

	$r = 0.01$	$r = 0.02$	$r = 0.03$	$r = 0.05$
L_1	23/30 (26 ± 5)	17/30 (28 ± 6)	18/30 (28 ± 6)	10/30 (36 ± 14)
L_2	21/30 (180 ± 39)	17/30 (221 ± 51)	4/30 (195 ± 21)	5/30 (250 ± 120)
L_3	14/30 (300 ± 480)	9/30 (310 ± 410)	10/30 (190 ± 360)	6/30 (250 ± 340)
L_4	11/30 (160 ± 190)	3/30 (120 ± 40)	7/30 (240 ± 300)	2/30 (80 ± 17)
L_5	8/30 (570 ± 180)	6/30 (430 ± 180)	7/30 (880 ± 670)	2/30 (190 ± 110)
L_6	11/30 (140 ± 65)	10/30 (280 ± 330)	5/30 (500 ± 720)	2/30 (148 ± 72)
L_7	5/30 (159 ± 85)	9/30 (222 ± 85)	6/30 (220 ± 100)	2/30 (381 ± 170)

Table 1: Ability of the network to recover Tomita's languages from corrupted samples

L_1	SUCCESS RATE	$\|Q\|_{\text{before}}$	$\|Q\|_{\text{after}}$	n_{exc}	$\|Q\|_{\text{after}}+n_{\text{exc}}$
0.01	27/30	6(3)	1.00(0.00)	1.1(0.7)	2.1(0.7)
0.02	18/30	9(3)	1.00(0.00)	2.0(0.9)	3.0(0.9)
0.03	18/30	9.8(2.5)	1.00(0.00)	2.8(1.4)	3.8(1.4)
0.05	19/30	12.7(1.7)	1.00(0.00)	4.6(1.3)	5.6(1.3)
L_2	SUCCESS RATE	$\|Q\|_{\text{before}}$	$\|Q\|_{\text{after}}$	n_{exc}	$\|Q\|_{\text{after}}+n_{\text{exc}}$
0.01	23/30	6(3)	1.9(0.6)	2(2)	3.8(1.8)
0.02	18/30	8(3)	2.1(0.9)	2(2)	4(2)
0.03	11/30	11(1.8)	2.4(1.1)	4(2)	6(2)
0.05	17/30	11(3)	1.7(0.7)	5(2)	6.6(1.9)
L_3	SUCCESS RATE	$\|Q\|_{\text{before}}$	$\|Q\|_{\text{after}}$	n_{exc}	$\|Q\|_{\text{after}}+n_{\text{exc}}$
0.01	14/30	6(2)	3(0)	0.9(0.8)	3.9(0.8)
0.02	14/30	10(3)	3(0)	3(7)	6(7)
0.03	16/30	11(3)	2.9(0.2)	5(10)	8(9)
0.05	8/30	16(4)	3(1)	13(15)	16(15)
L_4	SUCCESS RATE	$\|Q\|_{\text{before}}$	$\|Q\|_{\text{after}}$	n_{exc}	$\|Q\|_{\text{after}}+n_{\text{exc}}$
0.01	16/30	8(3)	3(1)	5(10)	8(10)
0.02	14/30	9(4)	2.8(0.5)	4(9)	7(8)
0.03	15/30	13(3)	2.7(1.3)	12(13)	15(12)
0.05	8/30	12(3)	2.5(0.9)	12(14)	15(13)
L_5	SUCCESS RATE	$\|Q\|_{\text{before}}$	$\|Q\|_{\text{after}}$	n_{exc}	$\|Q\|_{\text{after}}+n_{\text{exc}}$
0.01	7/30	5.3(1.5)	4(0)	0.4(0.5)	4.4(0.5)
0.02	6/30	8(3)	5.0(1.4)	0.8(0.9)	5.8(1.8)
0.03	6/30	12(3)	5.5(1.5)	2.3(0.8)	8(2)
0.05	1/30	17(0)	2(0)	22(0)	24(0)
L_6	SUCCESS RATE	$\|Q\|_{\text{before}}$	$\|Q\|_{\text{after}}$	n_{exc}	$\|Q\|_{\text{after}}+n_{\text{exc}}$
0.01	20/30	7(3)	4(2)	4(10)	8(10)
0.02	14/30	9(3)	6(3)	0.9(0.8)	6(3)
0.03	12/30	12(2)	7(4)	3(5)	11(6)
0.05	13/30	16(3)	6(4)	7(9)	13(8)
L_7	SUCCESS RATE	$\|Q\|_{\text{before}}$	$\|Q\|_{\text{after}}$	n_{exc}	$\|Q\|_{\text{after}}+n_{\text{exc}}$
0.01	14/30	7(3)	3.4(1.2)	7(12)	10(11)
0.02	16/30	11(3)	3.7(1.5)	7(11)	10(10)
0.03	14/30	13(3)	3.3(1.8)	13(14)	16(12)
0.05	7/30	16(3)	2.7(1.5)	19(14)	21(13)

Table 2: Neural relabeling: success rate, size of exact and approximate DFA, and number of exceptions (see text)

[10] J. Oncina, P. Garcia. In *Advances in Structural and Syntactic Pattern Recognition*. World Scientific, 1992.

[11] J.B. Pollack. The Induction of Dynamical Recognizers. *Machine Learning* 7:227–252, 1991.

[12] M. Tomita. Dynamic Construction of Finite-state Automata from Examples Using Hill-climbing. In *Proc. 4th Ann. Cognitive Sci. Conf*, p. 105–108.

[13] R.L. Watrous, G.M. Kuhn. Induction of Finite-State Languages Using Second-Order Recurrent Networks. *Neural Computation* 4(3):406–414, 1992.

[14] R.J. Williams, D. Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation* 1(2):270–280, 1989.