PERGAMON

Neural Networks 14 (2001) 93–105

Neural Networks

www.elsevier.com/locate/neunet

A real-coded genetic algorithm for training recurrent neural networks

A. Blanco, M. Delgado, M.C. Pegalajar*

Department of Computer Science and Artificial Intelligence, E.T.S.I. Informática, University of Granada, Avenida de Andalucía, 18071 Granada, Spain

Received 21 May 1999; accepted 5 September 2000

Abstract

The use of Recurrent Neural Networks is not as extensive as Feedforward Neural Networks. Training algorithms for Recurrent Neural Networks, based on the error gradient, are very unstable in their search for a minimum and require much computational time when the number of neurons is high. The problems surrounding the application of these methods have driven us to develop new training tools.

In this paper, we present a Real-Coded Genetic Algorithm that uses the appropriate operators for this encoding type to train Recurrent Neural Networks. We describe the algorithm and we also experimentally compare our Genetic Algorithm with the Real-Time Recurrent Learning algorithm to perform the fuzzy grammatical inference. © 2001 Elsevier Science Ltd. All rights reserved.

Keywords: Recurrent neural network; Fuzzy recurrent neural network; Training algorithms; Real-coded genetic algorithm; Fuzzy grammatical inference; Fuzzy finite-state automaton

1. Introduction

Many systems in the real world that we want to identify are non-linear systems or systems whose behavior depends on their current state. The Artificial Neural Networks that have given the best results in problems related with this type of systems are Recurrent Neural Networks (RNNs). In recent years, a great number of works have studied the capabilities and limitations of RNNs applied to subjects associated with pattern recognition and control. However, the use of RNNs is not as extended as Feedforward Neural Networks (FNNs) due to the complexity of the development of the learning algorithms. For FNNs, the error-gradient information computed using the backpropagation algorithm has been shown to be an effective and efficient tool for learning complex functions (Bourlard & Wellekens, 1989; Le Cun et al., 1989; Sejnowski & Rosenberg, 1987; Waibel, Hanazawa, Hinton, & Shikano, 1989). Unfortunately, the same does not occur with RNNs. There are algorithms that extend the backpropagation method to these networks, but the optimal training of an RNN using conventional gradient-descent methods is complicated due to many attractors in the state space. To solve these drawbacks, we have developed a genetic algorithm (GA) that optimizes the error made by the RNN.

The use of GAs for ANN training has mainly focused on FNNs (Blanco, Delgado, & Pegalajar, 2000a; Delgado, Mantas, & Pegalajar, 1996; Friedrich & Klaus, 1994; Whitely, Starkweather, & Bogart, 1990), although in recent years the advantages that GAs offer as training tools for RNNs have been also studied (Kim, Ku, & Mak, 1997; Kumagai, Wada, Mikami, & Hashimoto, 1997).

Fuzzy Grammatical Inference is a problem that can be solved using this kind of network. RNNs are able to carry out Crisp Grammatical Inference from positive and negative examples. However, very little has been reported on fuzzy examples under uncertainty conditions (Blanco, Delgado, & Pegalajar, 1998, 2000b,c; Omlin, Karvel, Thornber, & Giles, 1998). This problem seems interesting from the fuzzy point of view due to the great number of applications that fuzzy grammars have in areas such as digital circuit design (Mensch & Lipp, 1990), X-ray analysis (Pathak & Sankar, 1986), detecting and quantifying fuzzy artery lesions (Lalande & Jaulent, 1996), intelligent interface design (Hikmet, 1992), clinical monitoring (Friedrich & Klaus, 1994), lexical analysis (Mateesku, Salomaa, & Yu, 1995). In this work we carry out fuzzy grammatical inference using an RNN together with a linear output neuron.

A related subject is the implementation of fuzzy finite-state automata in Artificial Neural Networks, some methods for which have been proposed in the literature (Grantner & Patyra, 1993, 1994; Lee & Lee,

* Corresponding author.
E-mail address: mcarmen@decsai.ugr.es (M.C. Pegalajar).

1975; Omlin et al., 1998; Omlin, Giles, & Thornber, 1999; Unal & Khan, 1994).

This paper provides a brief introduction to fuzzy grammatical inference, the neural model used and an adaptation of the real-time recurrent learning algorithm in Section 2. In Section 3, a real-coded genetic algorithm is introduced. The real-coded genetic algorithm for training recurrent neural networks is presented in Section 4. In Section 5, we show the results obtained from both training algorithms for a particular example. Finally, some conclusions are provided.

2. Fuzzy grammatical inference and neural model

2.1. The fuzzy grammatical inference problem

Below, we provide some basic definitions and theorems for understanding the fuzzy grammatical inference problem. A more detailed presentation can be found in Dubois and Prade (1980), Gaines and Kohout (1976), Santos (1968), Thomason and Marinos (1974) and Wee Wee and Fu (1969).

Definition 1. A regular fuzzy grammar (RFG), $\mathscr{G}$, is a four-tuple $\mathscr{G} = (N, T, S, P)$, where N is a finite set of non-terminal symbols, T is a finite set of terminal symbols, $N \cap T = \emptyset$, $S \in N$ is the starting symbol, and P is a finite set of productions of the form $A \xrightarrow{\theta} a$ or $A \xrightarrow{\theta} aB$, $A, B \in N$, $a \in T$, $\theta \in (0, 1]$, where θ is a membership degree associated with the production considered.

Unlike the case of “crisp” regular grammars, where strings either belong or do not belong to language generated by the grammar, strings of a fuzzy language have graded membership.

Example 1. Let $\mathscr{G} = (N, T, S, P)$, where:

- $N = \{S, A, B\}$
- $T = \{a, b\}$
- S is the start symbol
- P is the set of productions

$$P = \{S \xrightarrow{0.3} aS, S \xrightarrow{0.5} aA, S \xrightarrow{0.7} aB, S \xrightarrow{0.3} bS, S \xrightarrow{0.2} bA, A \xrightarrow{0.5} b, B \xrightarrow{0.4} b\}.$$

Definition 2. The set of all finite strings formed by symbols in T plus the empty string of length 0 is denoted by T^*.

Definition 3. A fuzzy language, $L(\mathscr{G})$, is a fuzzy subset of T^* with associated membership function $\mu : T^* \rightarrow [0,1]$.

Definition 4. The membership degree $\mu(x)$ of a string x of T^* in the fuzzy regular language $L(\mathscr{G})$ is the maximum value of any derivation of x, where the value of a specific derivation of x is equal to the minimum weight of the productions used:

$$\mu_{\mathscr{G}}(x) = \mu_{\mathscr{G}}(S \overset{*}{\Rightarrow} x) = \max_{S \overset{*}{\Rightarrow} x} \min(\mu_{\mathscr{G}}(S \rightarrow \alpha_1),$$

$$\mu_{\mathscr{G}}(\alpha_1 \rightarrow \alpha_2), \ldots, \mu_{\mathscr{G}}(\alpha_m \rightarrow x))$$

where $\overset{*}{\Rightarrow}$ is the production chaining used to obtain the sequence x.

Example 2. Let us consider the grammar $\mathscr{G}$ in Example 1, for which the membership degree of sequence ab is:

$$\mu_{\mathscr{G}}(ab) = \mu_{\mathscr{G}}(S \overset{*}{\Rightarrow} ab) = \max_{S \overset{*}{\Rightarrow} ab} (\min(\mathscr{S} \xrightarrow{0.5} a\mathscr{A}, \mathscr{A} \xrightarrow{0.5} b),$$

$$\min(\mathscr{S} \xrightarrow{0.7} a\mathscr{B}, \mathscr{B} \xrightarrow{0.4} b) = \max_{S \overset{*}{\Rightarrow} ab} (0.5, 0.4) = 0.5.$$

In fuzzy regular grammars, all applicable production rules are executed to some degree. This leaves some uncertainty or ambiguity about the generated string.

Definition 5. A fuzzy finite-state automaton (FFA) is a six-tuple, $\mathscr{M} = (T, Z, Q, \delta, w, q_S)$, where T is a finite-input alphabet, Z is a finite-output alphabet, Q is a finite set of states, $\delta : T \times Q \times (0, 1] \rightarrow Q$ is the fuzzy-state transition map, $w : Q \rightarrow Z$ is the output map, and $q_S \in Q$ is the fuzzy initial state.

It should be noted that a regular fuzzy grammar as well as a fuzzy finite-state automaton are reduced to a conventional (crisp) one when all the production and transition degrees are equal to 1 (Dubois & Prade, 1980).

As in the crisp case, there is also an equivalence between fuzzy finite automaton and fuzzy regular grammars.

Theorem 1 (Dubois and Prade, 1980). *For a given fuzzy grammar $\mathscr{G}$ there exists a fuzzy finite-state automaton $\mathscr{M}$ such that $L(\mathscr{G}) = L(\mathscr{M})$ and vice-versa.*

Example 3. We consider the grammar $\mathscr{G}$ in Example 1, the fuzzy finite-state automaton $\mathscr{M} = (\Sigma, Z, Q, \delta, w, q_0)$ (see Fig. 1) such that $L(\mathscr{G}) = L(\mathscr{M})$ is:

- $\Sigma = \{a, b\}$
- $Z = \{Final, Non\text{-}Final\}$
- $Q = \{q_0, q_1, q_2, q_3\}$
- The fuzzy state transition map:

$$\delta(q_0, a, 0.5) = q_1, \quad \delta(q_0, b, 0.2) = q_1, \quad \delta(q_0, a, 0.3) = q_0,$$

$$\delta(q_0, b, 0.3) = q_0, \quad \delta(q_0, a, 0.7) = q_2, \quad \delta(q_1, b, 0.5) = q_3,$$

$$\delta(q_2, b, 0.4) = q_3$$

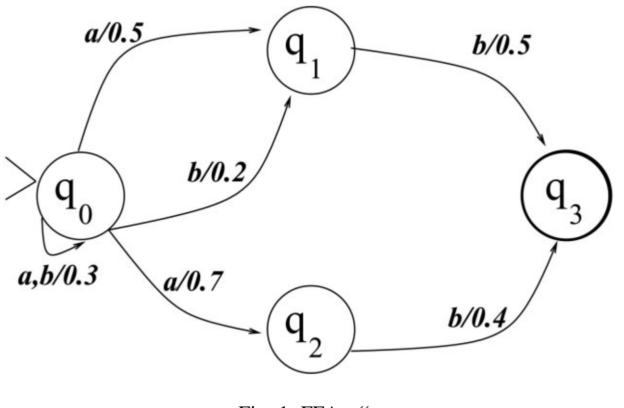

Fig. 1. FFA $\mathscr{M}$.

- The output map:

$$w(q_0) = \text{"Non-final"},\ w(q_1) = \text{"Non-final"},\ w(q_2) = \text{"Non-final"},\ w(q_3) = \text{"Final"}$$

  The final states are represented by circles with a thicker edge (Fig. 1).
- The starting state: q_0.

Definition 6. The membership degree of a string x computed by a fuzzy finite-state automaton is the degree maximum of all the paths accepting the string.

Example 4. The paths to accept the string *ab* are in Fig. 2. Therefore, the membership degree is $\mu(ab) = max(0.5, 0.4) = 0.5$.

Theorem 2 (Thomason and Marinos, 1974). *Given a fuzzy finite-state automaton $\mathscr{M}$, there exists a deterministic finite-state automaton $\mathscr{A}$ with output alphabet $Z \subseteq \{\theta : \theta$ is (a membership degree)$\} \cup \{0\}$ that computes the membership functions $\mu : T^* \to [0,1]$ of the language $L(\mathscr{M})$ in the output of its states (output map, w).*

This algorithm is an extension to the standard algorithm which transforms non-deterministic finite-state automata (Hopcroft & Ullman, 1979); unlike the standard transformation algorithm, we must distinguish accepting states with different fuzzy membership labels.

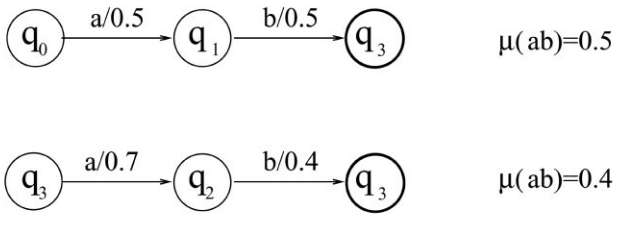

Fig. 2. Accepting paths to compute the membership degree of string *ab*.

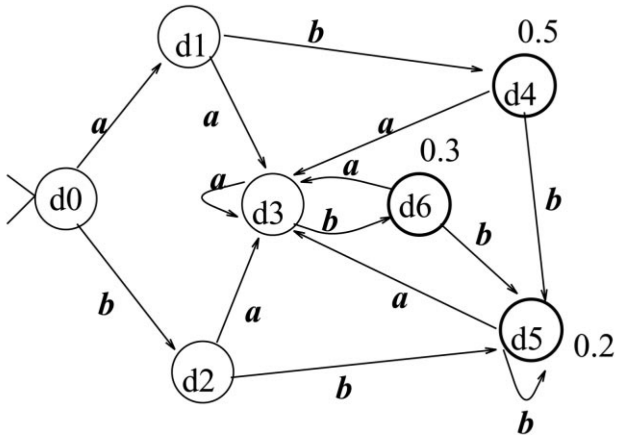

Fig. 3. DFA $\mathscr{A}$.

Example 5. We consider the fuzzy finite-state automaton $\mathscr{M}$ in Example 3. The DFA equivalent $\mathscr{A} = (\Sigma, Z', Q', \delta', w', d_0)$ is (Fig. 3):

- $\Sigma = \{a, b\}$
- $Z' = \{0.0, 0.2, 0.3, 0.5\}$
- $Q' = \{d_0, d_1, d_2, d_3, d_5, d_6\}$
- The state transition map:

$$\delta'(d_0,a) = d_1,\quad \delta'(d_0,b) = d_5,\quad \delta'(d_1,a) = d_3,$$
$$\delta'(d_1,b) = d_2,\quad \delta'(d_2,a) = d_3,\quad \delta'(d_2,b) = d_6,$$
$$\delta'(d_3,a) = d_3,\quad \delta'(d_3,b) = d_4,\quad \delta'(d_4,a) = d_3,$$
$$\delta'(d_4,b) = d_3,\quad \delta'(d_5,a) = d_3,\quad \delta'(d_5,b) = d_6,$$
$$\delta'(d_6,a) = d_3,\quad \delta'(d_6,b) = d_6$$

- w' is the output map (membership function):

$$w'(d_0) = 0.0,\quad w'(d_1) = 0.0,\quad w'(d_2) = 0.5,$$
$$w'(d_3) = 0.0,\quad w'(d_4) = 0.3,\quad w'(d_5) = 0.0,$$
$$w'(d_6) = 0.2.$$

The FFA-to-DFA transformation algorithm can be found in Thomason and Marinos (1974). The following corollary is a consequence of this theorem:

Corollary 1 (Eshelman and Scahffer, 1993). *Given a regular fuzzy grammar $\mathscr{G}$, there exists an equivalent grammar $\mathscr{G}$ in which the productions have the form $A \xrightarrow{\theta} a$ or $A \xrightarrow{1.0} aB$.*

Definition 7. Fuzzy Regular Grammatical Inference (FRGI) is defined as the problem of inferring the fuzzy productions that characterize a grammar or, equivalently, the states and transitions between states associated with a fuzzy state-finite automaton from a training example set.

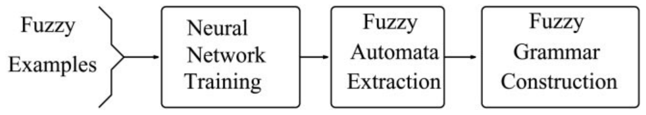

Fig. 4. The steps for performing fuzzy grammatical inference.

The examples are given couples (P_i, μ_i), $i = 1, \dots, n$, n being the number of examples, P_i a symbol sequence and μ_i the membership degree of P_i to the fuzzy language to which it belongs. μ_i will be provided by either an expert or any systematic procedure for each given P_i. Based on Theorem 2, the problem of obtaining the fuzzy finite-state automaton $\mathscr{M}$ is changed to extracting a deterministic finite automaton, $\mathscr{A}$, that calculates the membership function to the fuzzy language in the output map w of its states.

2.2. The neural model for carrying out fuzzy grammatical inference

The steps for carrying out fuzzy grammatical inference using an RNN are shown in Fig. 4. Once the RNN has been trained from an example set, we extract the automaton that recognizes the training set. From this automaton, we obtain the associated fuzzy regular grammar. In this paper, we focus only on the network training, as the automaton extraction can be found in Blanco et al. (2000b) and Delgado, Mantas, and Pegalajar (1996).

The problem of whether an FFA can be learned using an RNN only through training examples and the extracting algorithm of the FFA is a question studied in Blanco et al. (1998, 2000c). These works demonstrate that RNNs are able to learn a fuzzy example set and internally represent the equivalent DFA to the FFA, which is extracted together with the membership function to the fuzzy language, using an appropriate method of knowledge extraction (Blanco et al., 1998, 2000b). The neural network that we use here is composed of a second-order recurrent neural network (SORNN) and a linear output neuron (Fig. 5). The activating function of the output neuron is linear

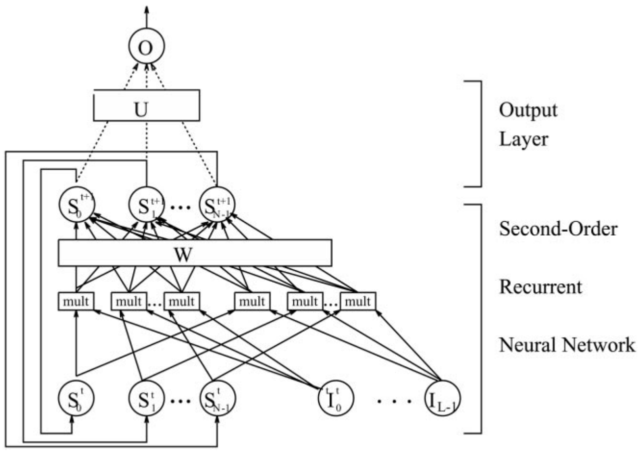

Fig. 5. Neural model for fuzzy grammatical inference.

since we intend to weigh the value of the points belonging to the same cluster in the same way. The output neuron computes the membership function to the fuzzy language we are attempting to identify.

The neural model (Fig. 5) proposed for the fuzzy grammatical inference consists of:

- N hidden recurrent neurons labeled S_j^t, $j = 0, \dots, N-1$
- L input neurons labeled I_k^t, $k = 0, \dots, L-1$
- $N^2 \times L$ weights labeled w_{ijk}
- One linear output neuron, O, connected to the hidden recurrent neurons by N weights labeled u_j, $j = 0, \dots, N-1$

L being the number of symbols belonging to the input alphabet.

This neural network accepts an input sequence ordered in time. Each symbol belonging to a sequence to be processed is sequentially encoded in the input neurons at each step in time t. The membership degree is computed by the linear output neuron once the input sequence has been fully processed by the SORNN.

The dynamics of the neural network is summarized in the following steps:

1. The initial values ($t = 0$) of the recurrent hidden neurons are fixed at $S_0^0 = 1$ and $S_i^0 = 0$ $\forall i \neq 0$.
2. Given an input sequence, Eq. (1) is evaluated for each one of the hidden recurrent neurons, obtaining the values they will take at the instant $t+1$, S_i^{t+1}

$$S_i^{t+1} = g(\theta_i^t), \tag{1}$$

$$\theta_i^t = \sum_{j=0}^{N-1} \sum_{k=0}^{L-1} I_k^t S_j^t w_{ijk}, \tag{2}$$

g being the sigmoidal function:

$$g(x) = \frac{1}{1 + \mathrm{e}^{(-x)}}. \tag{3}$$

Each element in a sequence to be processed is sequentially encoded in the input neurons at each step in time t by means of Kronecker's delta. Let us assume the alphabet is the symbol set $\{a_0, \dots, a_{L-1}\}$. If the tth character in the sequence to be processed is a_i, then it will be encoded in the input neurons exactly in time t by: $I_i^t = 1$, $I_j^t = 0$ $\forall j = 0 \dots L-1$, where $j \neq i$.

3. Once the recurrent neural network has fully processed the input sequence, the value of the output neuron, O, is obtained from the values S_i^m, $i = 0 \dots N-1$ (m being the sequence length), Eq. (4). The output neuron given us the membership degree to the fuzzy language $L(\mathscr{M})$ that the neural network has calculated:

$$O = \sum_{i=0}^{N-1} u_i S_i^m. \tag{4}$$

2.2.1. A learning algorithm based on the error gradient

The development of learning algorithms for RNNs has centered on using gradient descent algorithms, of which there are two basic types:

- *Real Time Recurrent Learning (RTRL)* (Pineda, 1988; Williams & Zipser, 1989). The main drawback of this algorithm is that it has a high computational cost for each iteration.
- *Backpropagation through time (BTT)* (Pearlmutter, 1989; Rumelhart & McClelland, 1986). The main limitation of this algorithm is that one must know in advance the length of the input sequence.

Although these algorithms are based on the backpropagation algorithm (Rumelhart & McClelland, 1986), they are computationally much more hard than the backpropagation algorithm used for feedforward networks.

A variation of the RTRL algorithm to train our neural network is presented below.

The error function in the output neuron is

$$E = \frac{1}{2}(T - O)^2, \tag{5}$$

where T is the desired value for the output neuron and O is the value obtained for the output neuron.

The training algorithm updates the weights at the end of the each sequence presentation when $E > \varepsilon$, where ε is the error tolerance of the neural network. The modification of the weights is given by

$$\Delta u_i = -\alpha \frac{\partial E}{\partial u_i}, \tag{6}$$

$$\Delta w_{lon} = -\alpha \frac{\partial E}{\partial w_{lon}}, \tag{7}$$

where α is the learning rate. The partial derivative in Eq. (6) can be directly obtained as:

$$\frac{\partial E}{\partial u_i} = (T - O) S_i^m, \tag{8}$$

where S_i^m is the final value of neuron i once the network has processed the whole sequence. The derivatives associated with w_{lon} are calculated by

$$\frac{\partial E}{\partial w_{lon}} = (T - O) \sum_{i=0}^{N-1} u_i g'(\theta_i^m) \times \left[\delta_{il} S_o^{m-1} I_n^{m+1} + \sum_{j=0}^{N-1} \sum_{k=0}^{L-1} w_{ijk} I_j^{m-1} \frac{\delta S_j^{m-1}}{\partial w_{lon}} \right]. \tag{9}$$

Since obtaining $\partial E/\partial w_{lon}$ requires a recursive calculation associated with $\partial S_i^{m-1}/\partial w_{lon}$, we fix an initial value of

$$\frac{\partial S_i^0}{\partial w_{lon}} = 0. \tag{10}$$

In the training of the recurrent neural network, the partial derivative of each hidden recurrent neuron related to each weight must be computed at each time t and the network consumes much computational time. In fact, the time complexity of the process is $O(N^4 \times L^2)$, which is a strong incentive for looking for new training algorithms, since when the number of recurrent neurons N increases, the algorithm presents serious problems. On the other hand, it is known that descent gradient training is unstable in the search for a minimum in the error function, usually remaining trapped in local minima. All these drawbacks are increased in the problem of fuzzy grammatical inference. Therefore, it is of great interest to develop new learning algorithms that search not only in depth, as the gradient does but also in width. With this aim we have developed the genetic algorithm we present in Section 4.

3. Genetic algorithms. Genetic algorithms with real codification

Genetic algorithms (GAs) are stochastic optimization algorithms based on the concepts of biological evolutionary theory (Goldberg, 1989; Holland, 1975). They consists in maintaining a population of chromosomes (individuals), which represent potential solutions to the problem to be solved, that is, the optimization of a function, generally very complex. Each individual in the population has an associated fitness, indicating the utility or adaptation of the solution that it represents.

A GA starts off with a population of randomly generated chromosomes and advances toward better chromosomes by applying genetic operators, modeled on the genetic processes occurring in nature. During successive iterations, called generations, the chromosomes are evaluated as possible solutions. Based on these evaluations, a new population is formed using a mechanism of selection and applying genetic operators such as crossover and mutation.

Although there are many possible variants on the basic GA, the operation of a standard genetic algorithm is described in the following steps:

1. Randomly create an initial population of chromosomes.
2. Compute the fitness of every member of the current population.
3. If there is a member of the current population that satisfies the problem requirements then stop. Otherwise, continue to the next step.
4. Create an intermediate population by extracting members from the current population using a selection operator.
5. Generate a new population by applying the genetic operators of crossover and mutation to this intermediate population.
6. Go back to step 2.

3.1. Real-coded genetic algorithms

The most common representation in GAs is binary (Goldberg, 1991). The chromosomes consist of a set of genes, which are generally characters belonging to an alphabet {0,1}. Therefore, a chromosome is a vector x consisting of l genes c_i:

$$x = (c_1, c_2, \ldots, c_l), \qquad c_l \in \{0, 1\},$$

where l is the length of the chromosome.

However, in the optimization problems of parameters with variables in continuous domains, it is more natural to represent the genes directly as real numbers since the representations of the solutions are very close to the natural formulation, i.e. there are no differences between the genotype (coding) and the phenotype (search space). The use of this real-coding initially appears in specific applications, such as in Lucasius and Kateman (1989) for chemometric problems, in Davis (1989) for the use of metaoperators to find the most adequate parameters for a standard GA, in Davis (1991) and Michalewicz (1992) for numerical optimization on continuous domains, etc. See Herrera, Lozano, and Verdegay (1998) for a review related to real-coded genetic algorithms.

In this case, a chromosome is a vector of floating point numbers. The chromosome length is the vector length of the solution to the problem; thus, each gene represents a variable of the problem. The gene values are forced to remain in the interval established by the variables they represent, so the genetic operators must fulfill this requirement.

Below, we show some crossover and mutation operators developed for this encoding.

3.1.1. Crossover operators

Let us assume that $C_1 = (c_1^1 \ldots c_N^1)$ and $C_2 = (c_1^2 \ldots c_N^2)$ are two chromosomes selected for application of the crossover operator.

Flat crossover (Radcliffe, 1991). An offspring $H = (h_1, \ldots, h_i, \ldots, h_N)$, is generated, where h_i is a randomly (uniformly) chosen value of the interval $[min(c_i^1, c_i^2), max(c_i^1, c_i^2)]$.

Simple crossover (Michalewicz, 1992; Wright, 1991). A position $i \in \{1, 2, \ldots, N-1\}$ is randomly chosen and two new chromosomes are built:

$$H_1 = (c_1^1, c_2^1, \ldots, c_i^1, c_{i+1}^2, \ldots, c_N^2),$$

$$H_2 = (c_1^2, c_2^2, \ldots, c_i^2, c_{i+1}^1, \ldots, c_N^1).$$

Arithmetic crossover (Michalewicz, 1992). Two offsprings, $H_k = (h_1^k, \ldots, h_i^k, \ldots, h_N^k)$, $k = 1, 2$, are generated, where $h_i^1 = \lambda c_i^1 + (1-\lambda)c_i^2$ and $h_i^2 = \lambda c_i^2 + (1-\lambda)c_i^1$. λ is a constant (uniform arithmetic crossover) or varies with regard to the number of generations (non-uniform arithmetic crossover).

BLX-α crossover (Blend Crossover) (Eshelman & Schaffer, 1993). An offspring, $H = (h_1, \ldots, h_2, \ldots, h_N)$ is generated, where h_i is a randomly (uniformly) chosen number of the interval $[c_{min} - I\alpha, c_{max} + I\alpha]$, $c_{max} = \max(c_i^1, c_i^2)$, $c_{min} = \min(c_i^1, c_i^2)$, $I = c_{max} - c_{min}$. The BLX-0.0 crossover is equal to the flat crossover.

Linear BGA (Breeder Genetic Algorithm) crossover (Schlierkamp-Voosen, 1994). Under the same consideration as above, $h_i = c_i^1 \pm rang_i \cdot \gamma \cdot \Lambda$, where $\Lambda = (c_i^2 - c_i^1)/\|c_1 - c_2\|$. The "$-$"sign is chosen with a probability of 0.9. Usually, $rang_i$ is $0.5(b_i - a_i)$ and $\gamma = \sum_{k=0}^{15} \alpha_k 2^{-k}$, where $\alpha_i \in \{0, 1\}$ is randomly generated with $p(\alpha_i = 1) = 1/16$.

Wright's heuristic crossover (Wright, 1990). If C_1 is the parent with the best fitness, then $h_i = r(c_i^1 - c_i^2) + c_i^1$ and r is a random number belonging to [0,1].

3.1.2. Mutation operators

Let us assume that $C = (c_1, \ldots, c_N)$ is a chromosome and $c_i \in [a_i, b_i]$ is a gene to be mutated. Below, the gene, c_i', resulting from the application of different mutation operators is shown.

Random mutation (Michalewicz, 1992). c_i' is a random (uniform) number from the domain $[a_i, b_i]$.

Non-uniform mutation (Michalewicz, 1992). If this operator is applied in a generation t, and $g_{\max}$ is the maximum number of generations, then

$$c_i' = \begin{cases} c_i + \Delta(t, b_i - c_i) & \text{if } \gamma = 0 \\ c_i + \Delta(t, c_i - a_i) & \text{if } \gamma = 1 \end{cases}$$

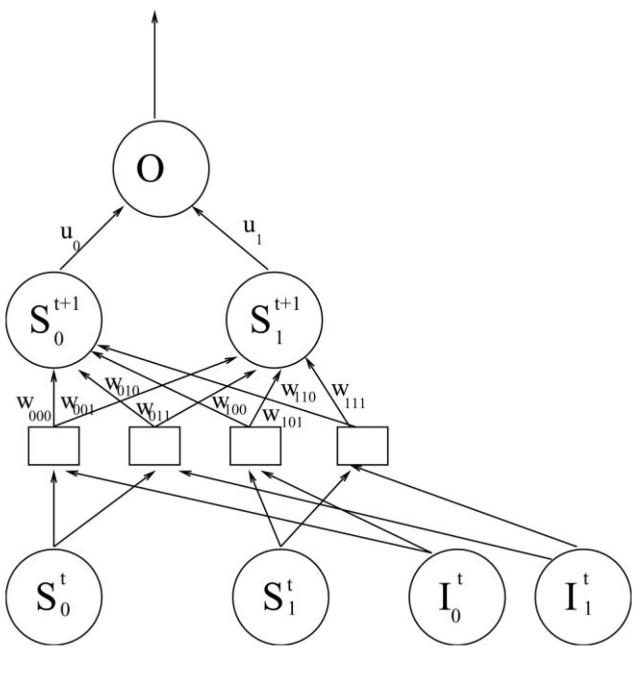

Fig. 6. Recurrent neural network with two recurrent neurons.

$$\boxed{w_{000} \quad w_{001} \quad w_{010} \quad w_{011} \quad w_{100} \quad w_{101} \quad w_{110} \quad w_{111} \quad u_0 \quad u_1}$$

Fig. 7. Chromosome associated with a RNN with two recurrent neurons.

Table 1
Real time recurrent learning results

N	α	Training error	% Training answer	Test error	% Test answer	Second	Cycles
2	0.001	0.0025134	0.0	0.0066073	0.0	28	499
2	0.001	0.0020750	27.3	0.0055342	21.33	28	499
2	0.01	0.0000696	70.7	0.0036460	74.00	28	500
2	0.01	0.0005002	50.0	0.0042157	47.33	28	500
2	0.1	0.0000439	93.3	0.0041507	86.67	28	499
2	0.1	0.0000357	100.0	0.0040093	90.67	28	500
2	0.15	0.0000445	94.0	0.0055345	86.67	28	494
2	**0.15**	**0.0000179**	**100.0**	**0.0032868**	**90.00**	**28**	**493**
2	0.2	0.0000310	99.3	0.0045314	90.00	28	500
2	0.2	0.0000227	100.0	0.0031371	90.0	28	500
3	0.001	0.00215536	35.3	0.0061612	30.0	81	370
3	0.001	0.00076021	22.7	0.0037913	17.4	81	405
3	0.01	0.00083431	0.0	0.0035844	0.0	81	420
3	0.01	0.00067655	49.3	0.0033454	38.5	81	400
3	0.1	0.00006457	88.0	0.0044989	79.2	81	410
3	0.1	0.00002442	100.0	0.0027916	90.0	81	410
3	0.15	0.00002102	100.0	0.0040773	91.33	81	415
3	**0.15**	**0.00002094**	**100.0**	**0.0030103**	**92**	**81**	**430**
3	0.2	0.00002512	99.3	0.0031494	88.67	81	415
3	0.2	0.00002670	100.0	0.0025400	92.2	81	425
4	0.001	0.0006315	8.0	0.0034830	0.0	128	195
4	0.001	0.0013394	0.0	0.0044371	0.0	128	209
4	0.01	0.0012158	3.6	0.004361	0.0	128	209
4	0.01	0.0000713	71.3	0.0030726	63.3	128	209
4	0.1	0.0000234	99.3	0.0021705	82.0	128	209
4	**0.1**	**0.0000202**	**99.3**	**0.0027550**	**89.33**	**128**	**210**
4	0.15	0.0000344	100.0	0.0018895	90.0	128	210
4	0.15	0.0000244	99.3	0.0029299	90.0	128	210
4	0.2	0.0000350	99.3	0.0043744	86.7	128	210
4	0.2	0.0000420	99.3	0.0025164	90.0	128	289

γ is a random number which may have a value of zero or one and

$$\Delta(t,y) = y(1 - r^{(1-(t/g_{max}))^b}),$$

where r is a random number from the interval [0,1] and b is a parameter, chosen by the user, that determines the degree of dependency on the number of iterations. This function gives a value in the range [0,y] such that the probability of returning a number close to zero increases as the algorithm advances.

4. A real-coded genetic algorithm to train recurrent neural networks

As mentioned above, real coding is the most suitable coding for continuous domains. Since our goal is recurrent neural network training, it appears logical to use this coding and genetic operators associated to it. Binary (ordinary) GA has been not used in this field. Among the advantages of using real-valued coding over binary coding is increased precision. Binary coding of real-valued numbers can suffer loss of precision depending on the number of bits used to represent one number. Moreover, in real-valued coding, chromosome strings become much shorter. For real-valued optimization problems, real-valued coding is simply much easier and more efficient to implement, since it is conceptually closer to the problem space. In particular, our aim is to train an RNN to perform fuzzy grammatical inference from a fuzzy example set. In other words, a weight set must be found so that the RNN behaves as the fuzzy finite-state automaton that recognizes the example set.

A chromosome or genotype consists of all the network weights. One gene of a chromosome represents a single weight value. The weights of the neural network are placed on a chromosome, as shown in the example below.

Example 6. We consider a Second-Order Recurrent Neural Network consisting of two hidden recurrent neurons, two input neurons, connected to a linear output neuron (Fig. 6). In this example, the chromosomes have 10 genes (Fig. 7). In general, a chromosome has $N^2 \times L + N$ genes, where N is the number of hidden recurrent neurons and L is the number of input neurons.

Fitness function. The fitness function should reflect the individual's performance in the current problem. We have chosen $1/m_e$ as a fitness function, where m_e is the average error in the training set. The best individual in the population is the one with the minimum error.

Table 2
Wright's heuristic crossover results

N	P_c	P_m	Training error	Training answer	Test error	% Test answer	Seconds	Generations
2	0.6	0.01	0.0000251	75.33	0.0028465	75.33	28	80
2	0.6	0.01	0.0000266	78.67	0.0025786	77.33	28	80
2	0.6	0.01	0.0000392	78.0	0.0038066	73.33	28	80
2	0.6	0.01	0.0000090	100.0	0.0028505	89.33	28	80
2	**0.6**	**0.01**	**0.0000072**	**100.0**	**0.0034316**	**90.67**	**28**	**80**
2	0.8	0.01	0.0000260	78.0	0.0040743	70.0	28	80
2	0.8	0.01	0.000013	90.66	0.0032212	80.0	28	80
2	0.8	0.01	0.000031	80.0	0.0042969	77.33	28	80
2	0.8	0.01	0.0000273	82.00	0.0050467	74.67	28	80
2	0.8	0.01	0.0000125	88.0	0.0042650	79.33	28	80
3	0.6	0.01	0.000005237	100.0	0.005130825	90.67	81	130
3	0.6	0.01	0.000006411	99.33	0.001658639	88.67	81	130
3	0.6	0.01	0.000003922	100.0	0.001160350	89.0	81	130
3	0.6	0.01	0.000004582	99.33	0.00240903	82.3	81	130
3	0.6	0.01	0.000001520	100.0	0.003658780	90.0	81	130
3	**0.8**	**0.01**	**0.000000835**	**100.0**	**0.003327544**	**90.0**	**81**	**130**
3	0.8	0.01	0.000001039	100.0	0.0033645703	90.67	81	130
3	0.8	0.01	0.000002704	100.0	0.005277894	92.7	81	130
3	0.8	0.01	0.000005178	98.67	0.002864849	91.4	81	130
3	0.8	0.01	0.000001513	100.0	0.003457828	97.3	81	130
4	0.6	0.01	0.00000766	99.33	0.0067284	88.0	128	130
4	0.6	0.01	0.00000413	100.0	0.00418239	90.0	128	130
4	0.6	0.01	0.00000659	98.0	0.00304362	86.0	128	130
4	0.6	0.01	0.00000185	100.0	0.00324213	90.67	128	130
4	0.6	0.01	0.00000338	100.0	0.0034262	92.0	128	130
4	**0.8**	**0.01**	**0.00000140**	**100.0**	**0.00217766**	**90.67**	**128**	**130**
4	0.8	0.01	0.00000815	98.67	0.0042978	85.33	128	130
4	0.8	0.01	0.00000224	100.0	0.0037389	89.33	128	130
4	0.8	0.01	0.00000595	99.33	0.0009753	90.67	128	130
4	0.8	0.01	0.00000144	99.33	0.0044461	89.33	128130	

As no gradient information is needed, we can also use the percentage of correct answers in the training set as an alternative (Blanco et al., 2000a; Delgado et al., 1996).

Stopping criterion. The algorithm stops when an individual recognizes all the examples or when a maximum number of generations has been run.

Creating an intermediate population. Selection mechanism. The selection process of stochastic sampling with replacement is used to create the intermediate population P' (Goldberg, 1991; Holland, 1975).

For each chromosome C_i in population P, the probability, $p_s(C_i)$, of including a copy of this chromosome in the intermediate population P' is calculated as:

$$p_s(C_i) = \frac{Fitness(C_i)}{\sum_{j=1}^{|P|} Fitness(C_j)},$$

where $|P|$ is the number of individuals in the population P.

Thus, the chromosomes with above-average fitness tend to receive more copies than those with below-average fitness.

Next the population is mapped onto a roulette wheel. Chromosomes, C_i, are selected in quantities according to their relative fitness values (after ranking). The probability that a chromosome x is selected, is equal to its relative fitness. Thus:

$$p_{select}(x) = \frac{f(x)}{\sum f}.$$

The roulette wheel operator is best visualized by imagining a wheel where each chromosome occupies an area that is related to its fitness.

Selecting a chromosome can be thought of as spinning the roulette wheel. When the wheel stops, a fixed marker determines which chromosome will be selected. This is repeated until the number of chromosomes required for the intermediate population, P', is obtained.

Generating a new population by applying the genetic operators to the intermediate population. Once the intermediate population is created, the next step is to for the population of the next generation by applying the crossover and mutation operators on the chromosomes in P'. Two chromosomes are randomly selected from this intermediate population and serve as parents. Depending upon a probabilistic chance p_c (crossover rate), it is decided whether these two will be crossed over. After applying these genetic operators, the resulting chromosome (offspring) is inserted into the new population. This step is repeated until the new population reaches the population size less two individuals

Table 3
FLAT crossover results

N	P_c	P_m	Training error	% Training answer	Test error	% Test answer	Seconds	Generations
2	0.6	0.01	0.0002645	6	0.0027777	0.0	28	80
2	0.6	0.01	0.0003348	49.33	0.0028329	48.67	28	80
2	0.6	0.01	0.0003763	27.33	0.0033377	13.0	28	80
2	0.6	0.01	0.0005090	50.0	0.0039493	47.33	28	80
2	**0.6**	**0.01**	**0.0000351**	**75.33**	**0.0044056**	**72.67**	**28**	**80**
2	0.8	0.01	0.0001075	62.0	0.0037328	50.0	28	80
2	0.8	0.01	0.0002236	26.66	0.0044952	31.33	28	80
2	0.8	0.01	0.0001097	62.0	0.0027442	55.33	28	80
2	0.8	0.01	0.0001617	49.33	0.0039485	48.00	28	80
2	0.8	0.01	0.0003253	50.00	0.0046291	36.0	28	80
3	0.6	0.01	0.00005226	62.0	0.00192591	62.0	81	125
3	0.6	0.01	0.00007457	48.0	0.00198125	46.67	81	125
3	0.6	0.01	0.00001302	100	0.00311659	88.67	81	125
3	0.6	0.01	0.00004462	62.0	0.00216208	61.33	81	125
3	0.6	0.01	0.00011352	49.3	0.0042735	48.67	81	125
3	0.8	0.01	0.00008057	62.0	0.00579852	61.33	81	130
3	0.8	0.01	0.00001843	84.0	0.00270929	84.0	81	130
3	0.8	0.01	0.00002087	87.3	0.00323798	78.0	81	130
3	0.8	0.01	0.00018110	30.6	0.00725754	25.33	81	130
3	**0.8**	**0.01**	**0.00001015**	**90.6**	**0.00209595**	**79.33**	**81**	**130**
4	0.6	0.01	0.0000564	66.0	0.00156253	66.0	128	140
4	0.6	0.01	0.0000457	84.0	0.00419596	83.3	128	140
4	0.6	0.01	0.0000248	89.33	0.00286092	78.0	128	140
4	0.6	0.01	0.0000146	99.33	0.00274103	90.0	128	140
4	0.8	0.01	0.0000142	99.33	0.00288451	90.67	128	140
4	0.8	0.01	0.0000509	74.66	0.00280194	72.67	128	140
4	0.8	0.01	0.0000273	83.33	0.00561753	73.33	128	140
4	0.8	0.01	0.0000136	95.33	0.00159258	84.67	128	140
4	**0.8**	**0.01**	**0.0000111**	**100.0**	**0.00174343**	**91.33**	**128**	**140**

($|P| - 2$). Moreover, the two best individuals in the current population are included in the new population (elitist strategy; De Jong, 1975), to make sure that the best-performing chromosome always survives intact from one generation to the next. This is necessary since the best chromosome could disappear, due to crossover or mutation.

Wright's heuristic crossover operator (Wright, 1990) (see Section 3.1.1) is used because of its good experimental behavior. After the application of the crossover operator, each of the genes of the resulting chromosome is subject to possible mutation, which depends on a probabilistic chance p_m, the mutation rate. The mutation operator used is random mutation (see Section 3.1.2).

5. Simulation

The simulation is designed as follows:

1. We start from a fuzzy finite-state automaton, $\mathscr{M}$, from Example 3 (Fig. 1), from which we will generate a set of 500 couples, $(P_i, \mu i)$, recognized by $\mathscr{M}$. Taking into account that now the running of $\mathscr{M}$ is to process any element from T^* and to give it a membership degree (which may be equal to 0 in the case that the string is not really a string of the fuzzy language), obtaining those 500 examples will be performed according to the following steps:
   - We calculate its length, f. We choose a random number f in the interval $[0\ldots Max]$, where Max is the maximum length of an example, $Max = 30$.
   - We calculate a symbol sequence, P_i. We randomly choose f symbols belonging to the alphabet, $\Sigma = \{a, b\}$.
   - We process sequence P_i by the automaton $\mathscr{M}$, obtaining the membership degree $\mu_{\mathscr{M}}(P_i)$ to the fuzzy language recognized by $\mathscr{M}$.
2. The example set obtained is divided into two subsets, a training set and a test set.
3. Once we have built an example set, our aim is to obtain the weights of the neural network that minimize the error of the training set. To obtain these weights we have trained a neural network with the RTRL algorithm and our genetic algorithm.
4. Finally, the results obtained by the two methods are compared.

5.1. Real-time recurrent learning algorithm results

We have trained neural networks with two, three and four hidden recurrent neurons. For each neural network we have

Table 4
BLX Crossover results

N	P_c	P_m	Training error	% Training answer	Test error	% Test answer	Seconds	Generations
2	0.6	0.01	0.0000292	74.66	0.0034141	70.0	28	80
2	0.6	0.01	0.0000879	27.33	0.0040177	20.5	28	80
2	0.6	0.01	0.0000290	82.0	0.0030317	79.6	28	80
2	0.6	0.01	0.0000154	94.0	0.0033146	88.0	28	80
2	0.6	0.01	0.0000198	89.3	0.0042698	83.0	28	80
2	0.8	0.01	0.0000171	86.6	0.0032124	79.0	28	80
2	**0.8**	**0.01**	**0.0000076**	**98.66**	**0.0030808**	**89.5**	**28**	**80**
2	0.8	0.01	0.0000124	100.0	0.0031278	91.33	28	80
2	0.8	0.01	0.0000270	78.0	0.0026403	78.0	28	80
2	0.8	0.01	0.0000077	100.0	0.0037394	91.33	28	80
3	0.6	0.01	0.00000769	100.0	0.00355957	89.33	81	120
3	0.6	0.01	0.00000699	99.3	0.00340276	91.33	81	120
3	0.6	0.01	0.00001455	99.3	0.00323907	89.33	81	120
3	0.6	0.01	0.00000423	99.3	0.00186712	90.0	81	120
3	**0.6**	**0.01**	**0.00000123**	**100.0**	**0.00222766**	**90.67**	**81**	**120**
3	0.8	0.01	0.00002105	99.3	0.00283124	84.67	81	120
3	0.8	0.01	0.00000663	100	0.00189647	90.67	81	120
3	0.8	0.01	0.00001166	98	0.00181295	94.0	81	120
3	0.8	0.01	0.00001459	84	0.00153861	80.0	81	120
3	0.8	0.01	0.00000689	99.3	0.00297651	88.67	81	120
4	0.6	0.01	0.000006872	100.0	0.00207004	90.67	128	140
4	0.6	0.01	0.000016593	90	0.00291990	79.33	128	140
4	0.6	0.01	0.000004080	100	0.00244884	90.0	128	140
4	0.6	0.01	0.000018792	90	0.00211171	82.67	128	140
4	0.6	0.01	0.000011337	99.3	0.00387898	88.67	128	140
4	0.8	0.01	0.000005364	98.6	0.00338552	86.0	128	140
4	0.8	0.01	0.000003909	99.3	0.00208693	91.33	128	140
4	**0.8**	**0.01**	**0.000001072**	**100**	**0.00325102**	**90.67**	**128**	**140**
4	0.8	0.01	0.000005037	100	0.00244563	88.67	128	140
4	0.8	0.01	0.000026694	93.3	0.00442973	81.33	128	140

performed 10 simulations. Each simulation has been run a certain number of seconds (seventh column). The initial weights were randomly chosen in the interval [−1,1]. The results obtained are shown in Table 1. The first column represents the number of neurons used. The second column shows the learning rate. The third and fourth columns give the Error and the percentage of successes in the training set. The fifth and sixth columns show the Error and the percentage of successes in the test set. The number of seconds that the algorithm has been run and the cycles performed are shown in the seventh and eighth columns, respectively.

Rows 1–10 in Table 1 show the results obtained for an RNN with two recurrent neurons. The best result appears in row 8, with a training error of $E = 0.0000179$. The mean training error for the simulations of two recurrent neurons is 5.3539×10^{-4}.

Rows 11–20 in Table 1 give the results for an RNN with three recurrent neurons. The best result is a training error, $E = 0.00002094$ (row 18). The mean training error for three recurrent neurons is 4.6092×10^{-4}.

Rows 21–30 in Table 1 show the results obtained with an RNN with four recurrent neurons. The best result obtained is a training error, $E = 0.0000202$ (row 26). The mean training error for four recurrent neurons is 3.4374×10^{-4}.

5.2. Real-coded genetic algorithm results

Wright's heuristic crossover. As in the previous algorithm, we have trained the RNN with two, three and four hidden recurrent neurons. For each network we performed 10 simulations for a certain number of seconds, with a population of 50 individuals. The interval where the genes can take their values is [−1,1]. The parameters associated to each simulation and the results obtained appear in Table 2. Column 1 gives the number of neurons used in each simulation. Columns 2 and 3 show the crossover and mutation probabilities, respectively. The error and percentage of successes for the training set are presented in columns 4 and 5. The information associated with the test set appears in columns 6 and 7. Column 8 shows the seconds that the genetic algorithm has been run and column 9 the cycles performed during that time. Table 2 is associated with the RNN with two, three and four recurrent neurons.

Rows 1–10 in Table 2 show the results for an RNN with three recurrent neurons. The best result is simulation 5, with a training error of $E = 0.0000072$. The mean training error for two recurrent neurons is 2.168×10^{-5}.

Rows 11–20 in Table 2 show the simulations associated to the RNN with three recurrent neurons. The

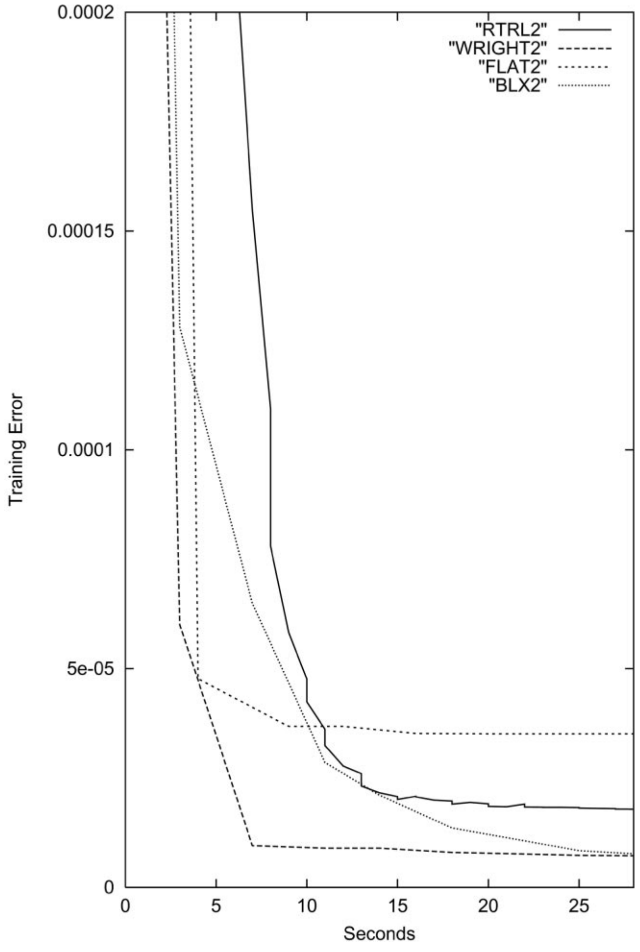

Fig. 8. The best simulation for a recurrent neural network with two recurrent hidden neurons.

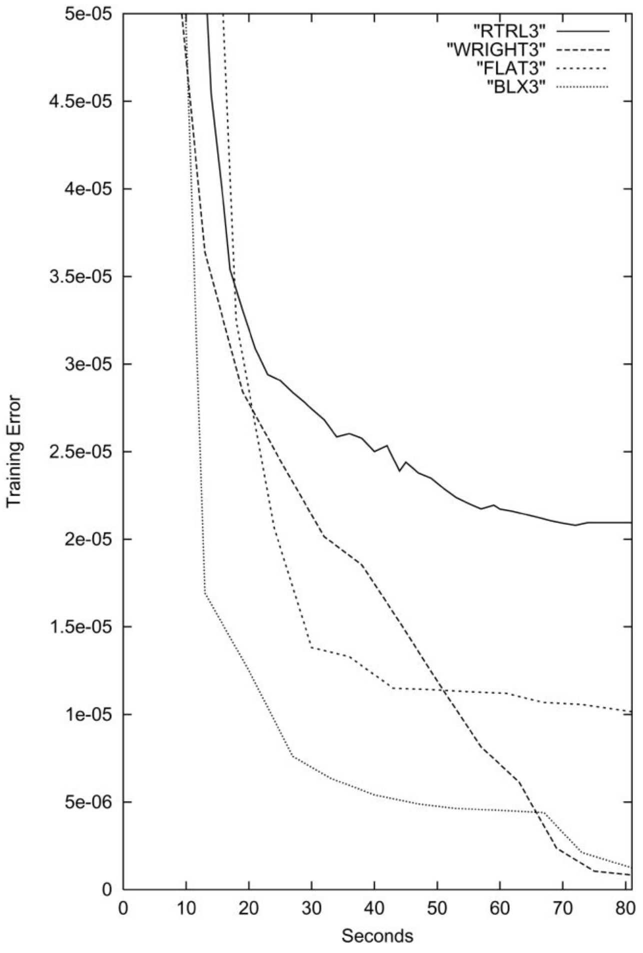

Fig. 9. The best simulation for a recurrent neural network with three recurrent hidden neurons.

best result is a training error of $E = 0.000000835$ (row 16). The mean training error for three recurrent neurons is 3.2922×10^{-6}.

Rows 21–30 in Table 2 show the simulation results for the RNN with four recurrent neurons. Simulation 26 gives the best results, $E = 0.00000140$. The mean training error for four recurrent neurons is 4.279×10^{-6}.

5.2.1. Using other crossover operators

In this section we have used other crossover operators to experimentally demonstrate that Wright's crossover is best. Next, we show the results obtained with the BLX and FLAT crossovers. The parameters used in the GA are the same as in the previous case.

FLAT crossover. For the case of two neurons (rows 1–10 in Table 3), the best result obtained is $E = 0.0000351$ (row 5). The mean training error for two recurrent neurons is 2.4475×10^{-4}.

For three neurons (rows 11–20 in Table 3), the best result is in simulation 20, $E = 0.00001015$. The mean training error for three recurrent neurons is 6.0911×10^{-5}. Finally, for four neurons (rows 21–30 in Table 3), the best result obtained is $E = 0.0000111$ (simulation 30). The mean training error for four recurrent neurons is 2.586×10^{-5}.

BLX Crossover. For the case of two neurons (rows 1–10 in Table 4), the best result obtained with the BLX crossover is in simulation 7 with an error of $E = 0.0000076$. The mean training error for two recurrent neurons is 2.531×10^{-5}.

Rows 11–20 in Table 4 show the results obtained for the simulations with a neural network of three neurons. The best result appears in row 15, $E = 0.00000123$. The mean training error for three recurrent neurons is 9.551×10^{-6}.

Finally, the result obtained with four neurons appears in rows 21–30 (Table 4). The best simulation obtained a training error, $E = 0.000001072$ (row 28). The mean training error for four recurrent neurons is 9.975×10^{-6}.

5.3. Comparison of results

If we observe the results obtained in Tables 1–4, we see that the Real-Coded Genetic Algorithm behaves better than the RTRL algorithm in the search of the Recurrent Neural Network weights.

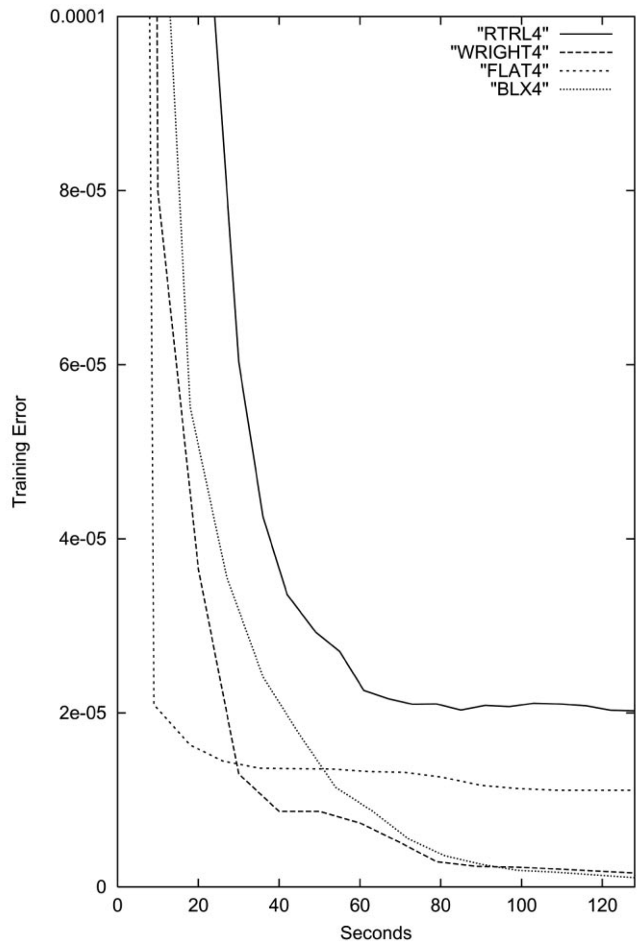

Fig. 10. The best simulation for a recurrent neural network with four recurrent hidden neurons.

Comparing the results summarized in Tables 1–4, we can conclude the following:

- In the RTRL algorithm, if smaller learning rates are used, the learning is more stable, although the training is slower. On the other hand, if large learning rates are used, then the training is more unstable.
- When N increases, the RTRL is hard to apply, because it becomes very slow.
- The genetic algorithm with any crossover (BLX, Flat or Wright's heuristic crossover), provides better mean results than RTRL and, furthermore, is quicker in the search for a good result. Figs. 8, 9 and 10 show the best simulation performed (the simulation with the least error), case four, with two, three and four neurons, respectively. We can observe that the genetic algorithm obtains the best results more quickly than the RTRL.
- The genetic algorithm with Wright's heuristic crossover obtains the best mean results, although the BLX-crossover gives ones very close to it. Hence, both crossovers can be used indistinctly.
- The crossover producing the worst results is the Flat crossover.

6. Conclusions

The RTRL algorithm is an unstable algorithm searching a global minimum and may easily remain trapped in local minimums. It has the drawback that if a small learning rate is used the training is slow, and if, on the contrary, a large learning rate is used, the training is unstable. Furthermore, this algorithm has a high complexity time, $O(N^4 \times L^2 + N)$. When N increases, the RTRL algorithm presents serious problems. All this leads us to outline new training methods to smoothen these problems. The GAs are able to search not only in depth, like the RTRL algorithm, but also in width. The most appropriate coding for neural network training is therefore real-coding. Experimentally, our Real-Coded Genetic Algorithm using Wright's heuristic crossover and random mutation obtains the best results. The time complexity of this GA is $O(N^2 \times L^2 + N)$, which means the training cost is still manageable as the number of neurons increases. In addition, our genetic algorithm using a BLX or Flat crossover obtains the best mean results and does so more quickly than the RTRL.

References

Blanco, A., Delgado, M., & Pegalajar, M. C. (1998). Fuzzy automaton induction using neural networks. *Technical Report.* Department of Computer Science and Artificial Intelligence, University of Granada, Spain.

Blanco, A., Delgado, M., & Pegalajar, M. C. (2000a). A genetic algorithm to obtain the optimal recurrent neural network. *International Journal of Approximate Reasoning*, *23*, 67–83.

Blanco, A., Delgado, M., & Pegalajar, M. C. (2000b). Extracting rules from a (fuzzy/crisp) recurrent neural networks using a self-organizing map. *International Journal of Intelligent Systems*, *2* (7), 595–621.

Blanco, A., Delgado, M., & Pegalajar, M. C. (2000c). Identification of fuzzy dynamic systems using max–min recurrent neural networks. *Fuzzy Sets and Systems*, (in press).

Bourlard, H., & Wellekens, C. (1989). Speech pattern discrimination and multi-layered perceptrons. *Computer Speech and Language*, *3*, 1–19.

Davis, L. (1989). Adapting operator probabilities in genetic algotihtms. In J. David Schaffer, *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 61–69). San Mateo: Morgan Kaufmann.

Davis, L. (1991). *Handbook of genetic algorithms*, New York: Van Nostrand Reinhold.

De Jong, K. A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* Doctoral dissertation, University of Michigan.

Delgado, M., Mantas, C., & Pegalajar, M. C. (1996). A genetic procedure to tune perceptrons. In *Proceeding of Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU'96),* Vol. 2 (pp. 963–969).

Dubois, D., & Prade, H. (1980). Fuzzy sets and systems: theory and applications. *Mathematics in Science and Engineering*, *144*, 220–226.

Eshelman, L. J., & Scahffer, J. D. (1993). Real-coded genetic algorithms and interval-schemata. In L. Darrel Whitley, *Foundations of genetic algorithms 2* (pp. 187–202). San Mateo: Morgan Kaufmann.

Friedrich, S., & Klaus, P. A. (1994). Clinical monitoring with fuzzy automata. *Fuzzy Sets and Systems*, *61*, 37–42.

Gaines, B., & Kohout, L. (1976). The logic of automata. *International Journal of Genetic Systems*, *2*, 191–208.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*, New York: Addison-Wesley.

Goldberg, D. E. (1991). Real-coded genetic algorithms. Virtual alphabets, and blocking. *Complex Systems*, *5*, 139–167.

Grantner, J., & Patyra, M. (1993). VLSI implementations of fuzzy logic finite state machine models. In *Proceedings of the Fifth IFSA Congress* (pp. 781–784).

Grantner, J., & Patyra, M. (1994). Synthesis and analysis of fuzzy logic finite state machine models. In *Proceedings of the Third IEEE Conference on Fuzzy Systems*, Orlando, FL, Vol. I (pp. 205–210).

Herrera, F., Lozano, M., & Verdegay, J. L. (1998). Tackling real-coded genetic algorithms: operators and tools for behavioural analysis. *Artificial Intelligence Review*, *12*, 265–319.

Hikmet, S. (1992). Fuzzy command grammars for intelligent interface design. *IEEE Transactions on Systems, Man, and Cybernetics*, *22* (5), 1124–1131.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*, The University of Michigan Press.

Hopcroft, J., & Ullman, J. (1979). *Introduction to automata theory, languages, and computation*, Reading, MA: Addison-Wesley.

Kim, W., Ku, C., & Mak, M. W. (1997). Exploring the effects of Lamarckian and Baldwinian learning in evolving recurrent neural networks. *IEEE Transactions on Evolutionary Computation*, 617–620.

Kumagai, T., Wada, M., Mikami, S., & Hashimoto, R. (1997). Structured learning in recurrent neural network using genetic algorithm with internal copy operator. *IEEE Transactions on Evolutionary Computation*, 651–656.

Lalande, A., & Jaulent, M. (1996). A fuzzy automaton to detect and quantify fuzzy artery lesions from arteriograms. In *Sixth International Conference IPMU96 (Information Processing and Management of Uncertainty in Knowledge-Based Systems),* Vol. 3 (pp. 1481–1487).

Le Cun, Y., Boser, B., Denker, J., Henderson, D., Howard, Hubbard, W., & Jackel, L. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, *1*, 541–551.

Lee, S., & Lee, E. (1975). Fuzzy neural networks. *Mathematical Biosciences*, *23*, 151–177.

Lucasius, C. B., & Kateman, G. (1989). Applications of genetic algorithms in chemometrics. In J. David Schaffer, *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 170–176). San Mateo: Morgan Kaufmann.

Mateesku, A., Salomaa, A., Solomaa, K., & Yu, S. (1995). Lexical analysis with a simple finite-fuzzy-automaton model. *Journal of Universal Computer Science*, *1* (5), 292–311.

Mensch, S., & Lipp, H. (1990). Fuzzy specification of finite state machines. In *EDAC Proceedings of the European Design Automation Conference,* Glasgow, UK, March (pp. 622–626).

Michalewicz, Z. (1992). *Genetic algorithms + data structures = evolution programs*, New York: Springer.

Omlin, C. W., Karvel, K., Thornber, K. K., & Giles, C. L. (1998). Fuzzy finite-state automata can be deterministically encoded into recurrent neural networks. *IEEE Transactions on Fuzzy Systems*, *6*, 76–89.

Omlin, C. W., Giles, C. L., & Thornber, K. K. (1999). Equivalence in knowledge representation: automata, recurrent neural networks, and dynamical fuzzy systems. *Technical Report,* Department of Computer Science, University of Stellenbosh, South Africa.

Pathak, A., & Sankar, K. P. (1986). Fuzzy grammars in sintactic recognition of skeletal maturity from X-Rays. *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-16* (5), 657–667.

Pearlmutter, B. (1989). Learning state space trajectories in recurrent neural networks. *Neural Computation*, *1*, 263–269.

Pineda, F. (1988). Dynamics and architecture for neural computation. *Journal of Complexity*, *4*, 216–245.

Radcliffe, N. J. (1991). Equivalence class analysis of genetic algorithms. *Complex Systems*, *5* (2), 183–205.

Rumelhart, D., & McClelland, J. (1986). *Parallel distributed processing: explorations in the microstructure of cognition: foundations*, Vol. 1. Cambridge, MA: MIT Press.

Santos, E. (1968). Maximin automata. *Information Contributions*, *13*, 363–377.

Schlierkamp-Voosen, D. (1994). Strategy adaptation by competition. In *Proceedings of the Second European Congress on Intelligent Techniques and Soft Computing* (pp. 1270–1274).

Sejnowski, T., & Rosenberg, C. (1987). Parallel networks that learn to pronounce English text. *Complex Systems*, *1*, 145–168.

Thomason, M., & Marinos, P. (1974). Deterministic acceptors of regular fuzzy languages. *IEEE Transactions on Systems, Man, and Cybernetics*, *3*, 228–230.

Unal, F., & Khan, E. (1994). A fuzzy finite state machine implementation based on a neural fuzzy system. In *Proceedings of the Third International Conference on Fuzzy Systems,* Orlando, FL, Vol. 3 (pp. 1749–1754).

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. (1989). Phoneme recognition using time delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *37*, 328–339.

Wee Wee, W., & Fu, K. (1969). A formulation of fuzzy automata and its applications as a model of learning systems. *IEEE Transactions on Systems, Man, and Cybernetics*, *5*, 215–223.

Whitely, D., Starkweather, T., & Bogartm, C. (1990). Genetic algorithms and neural networks — optimizing connections and connectivity. *Parallel Computing*, *14*, 347–361.

Williams, R., & Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, *1*, 270–277.

Wright, A. (1990). Genetic algorithms for real parameter optimization. foundations of genetic algorithms. In G. J. E. Rawlin, *First Workshop on the Foundations of Genetics Algorithms and Classifier Systems* (pp. 205–218). Los Altos, CA: Morgan Kaufmann.

Wright, A. (1991). Genetic algorithms for real parameter optimization. In G. J. E. Rawlin, *Foundations of genetic algorithms 1* (pp. 205–218). San Mateo: Morgan Kaufmann.