Available online at www.sciencedirect.com
SCIENCE DIRECT
Neural Networks 19 (2006) 477–486
Neural Networks
www.elsevier.com/locate/neunet

Evolutionary product unit based neural networks for regression

Alfonso Martínez-Estudillo [a], Francisco Martínez-Estudillo [a,*], César Hervás-Martínez [b], Nicolás García-Pedrajas [b]

[a] Department of Applied Mathematics, ETEA, 14004 Córdoba, Spain
[b] Department of Computing and Numerical Analysis, University of Córdoba, 14071 Córdoba, Spain

Received 9 January 2004; accepted 25 November 2005

Abstract

This paper presents a new method for regression based on the evolution of a type of feed-forward neural networks whose basis function units are products of the inputs raised to real number power. These nodes are usually called *product units*. The main advantage of product units is their capacity for implementing higher order functions.

Nevertheless, the training of product unit based networks poses several problems, since local learning algorithms are not suitable for these networks due to the existence of many local minima on the error surface. Moreover, it is unclear how to establish the structure of the network since, hitherto, all learning methods described in the literature deal only with parameter adjustment. In this paper, we propose a model of evolution of product unit based networks to overcome these difficulties. The proposed model evolves both the weights and the structure of these networks by means of an evolutionary programming algorithm.

The performance of the model is evaluated in five widely used benchmark functions and a hard real-world problem of microbial growth modeling. Our evolutionary model is compared to a multistart technique combined with a Levenberg–Marquardt algorithm and shows better overall performance in the benchmark functions as well as the real-world problem.

Keywords: Product units; Regression; Evolutionary computation

1. Introduction

In the area of neural network application, one of the most interesting fields is function regression (Hwang, Lay, Maechler, Martin, & Schimert, 1994). The fact that a neural network with sigmoidal transfer functions can approximate any given continuous function with the desired accuracy (Hornik, 1989) is a powerful basis for the application of neural networks to regression.

Neural network regression is an instance of model-free or non-parametric regression. We can state a model-free regression problem as follows (Hwang et al., 1994). Given n pairs of vectors

$$(\mathbf{x}_l,y) = (x_{l1},\ldots,x_{lk};y), \qquad l = 1,2,\ldots,n, \tag{1}$$

that have been generated from unknown models

$$y = g(\mathbf{x}_l) + \varepsilon_l, \qquad l = 1,2,\ldots,n, \tag{2}$$

where y is the response variable and $\mathbf{x}_l$ are the independent variables. g is an unknown smooth non-parametric function from p-dimensional Euclidean space to $\mathbb{R}$

$$g : \mathbb{R}^k \to \mathbb{R}. \tag{3}$$

ε_l are random variables with zero mean, $E[\varepsilon_l]=0$, and independent of $\mathbf{x}_l$. The aim of the regression is to construct an estimator $\hat{g}$, which is a function of the data $(\mathbf{x}_l,y)$, to approximate the unknown function g, and use this estimation to predict a new y given a new $\mathbf{x}$

$$\hat{y} = \hat{g}(\mathbf{x}). \tag{4}$$

Neural networks can be considered similar to *basis function* models (Denison, Holmes, Mallick, & Smith, 2002). These models assume that g is made up of a linear combination of basis functions and corresponding coefficients. Hence, g can be written

$$g(\mathbf{x}) = \sum_{j=1}^{m} \beta_j B_j(\mathbf{x}), \qquad \mathbf{x} \in \mathbb{R}^k, \tag{5}$$

* Corresponding author.
E-mail addresses: acme@etea.com (A. Martínez-Estudillo), fjmestud@etea.com (F. Martínez-Estudillo), chervas@uco.es (C. Hervás-Martínez), ngpedrajas@ieee.org (N. García-Pedrajas).

0893-6080/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.neunet.2005.11.001

where $\beta=(\beta_1,\ldots,\beta_m)'$ is the set of coefficients corresponding to basis functions $B=(B_1,\ldots,B_m)$. Typically, the basis functions of (5) are non-linear transformations. Methodologically, there is a major separation in neural network approach, as the combination of the basis functions is not always linear, as in (5), and additional sets of basis functions, represented by hidden layers, can be constructed.

Let us consider a feedforward neural network with k inputs and a hidden layer with m nodes. The hidden layer carries out a non-linear projection of input vector $\mathbf{x}$ to a vector $\mathbf{h}$ where

$$h_j = f\left(\sum_{i=1}^{k} w_{ji}x_i\right), \qquad j = 1,\ldots,m. \tag{6}$$

As we have stated, each node performs a non-linear projection of the input vector. So, $\mathbf{h}=f(\mathbf{x})$, and the output layer obtains its output from vector $\mathbf{h}$. Moreover, the projection performed by the hidden layer of a multilayer perceptron distorts the data structure and inter-pattern distances (Lerner, Guterman, Aladjem, & Dinstein, 1999) in order to achieve a better approximation.

If the function f in (6) is of sigmoid type, e.g. logistic or hyperbolic tangent functions, we obtain the standard multilayer perceptron.

Nevertheless, due to the lack of transparency of the sigmoidal functions in several regression problems, different alternative basis functions have been proposed. Among others, we can mention Gaussian functions (Kosko, 1991), radial basis functions (Lee & Hou, 2002), projection pursuit learning (Zhao & Atkeson, 1996) and general regression networks (Tomandl & Schober, 2001). In the same way, new models of networks have recently been proposed for regression and classification (Arulampalam & Bouzerdoum, 2003; Specht, 1991).

Among the most interesting models are the multiplicative neural networks that contain nodes that multiply their inputs instead of adding them and thus allow inputs to interact non-linearly. This class of multiplicative neural networks comprises such types as sigma-pi networks and product unit networks. A multiplicative node is given by

$$y_j = \prod_{i=1}^{k} x_i^{w_{ji}}, \tag{7}$$

where k is the number of inputs. If the exponents are $\{0,1\}$ we obtain a higher-order unit, also well-known under the name of sigma-pi unit. In contrast to the σ–π unit, in the product unit (PU) the exponents are not fixed and may even take real values. Product units were introduced by Durbin and Rumelhart (1989). They suggested two types of network incorporating PUs. In the first network type, each additive unit is connected to a group of dedicated PUs. The second type consists of alternate layers of product and summation units, terminating the network with an additive unit. This paper used three-layer networks, where only the hidden layer consists of PUs, while the output layer has additive units. Both layers use linear activation functions.

Advantages of product unit based neural networks (PUNNs) are increased information capacity and the ability to form higher-order combinations of the inputs. Durbin and Rumelhart (1989) determined empirically that the information capacity of product units (measured by their capacity for learning random Boolean patterns) is approximately $3N$, compared to $2N$ of a network with additive units for a single threshold logic function, where N denotes the number of inputs to the network. As well as this, it is possible to obtain upper bounds of the VC dimension of product unit neural networks similar to those obtained for sigmoidal neural networks (Schmitt, 2001).

From the point of view of the analytical structure of the product units, it is interesting to note that the partial derivatives of the function implemented by a product unit are again a function of the same type. This fact makes it easier to study the change rate of the function with respect to the variables. Furthermore, we show in this paper that product units can approximate any function with a given accuracy as well as sigmoidal neural networks (see proof in Section 3.2).

Despite these obvious advantages, product unit based networks have a major drawback. Their training is more difficult than the training of standard sigmoidal based networks (Durbin & Rumelhart, 1989). The main reason for this difficulty is that networks based on product units have more local minima and more probability of becoming trapped in them (Ismail & Engelbrecht, 2000). It is a well known issue (Janson & Frenzel, 1993) that back-propagation is not efficient in training product units. Several efforts have been made to develop learning methods for product units (see Section 2).

In this paper, we propose a model of evolutionary programming for automatically obtaining the structure and weights of a neural network based on product units. We apply a population-based evolutionary algorithm for architectural design and the estimation of weights in the neural network. The algorithm begins the search with an initial population and on each iteration the population is updated. It uses the operations of replication and two types of mutation: structural and parametric. The structural mutation implies a modification of the structure of the function performed by the network and allows an exploration of different regions of the search space. The parametric mutation modifies the coefficients of the model using a simulated annealing algorithm.

In order to test the performance of the model, it is applied to five benchmark functions and a real-world problem of estimation of parameters in microbial growth.

This paper is organized as follows: Section 2 reviews the related work on product unit based neural networks; Section 3 explains the proposed model in depth; Section 4 describes the experiments carried out; and Section 5 states the conclusions of our paper.

2. Related work

As we have said, back-propagation is not efficient in training product units. The large number of local minima makes gradient based algorithms inefficient. Different attempts have

been made in order to obtain useful algorithms for training product unit based networks.

Janson and Frenzel (1993) developed a genetic algorithm for evolving the weights of a network based on product units with a predefined architecture. The major problem of this kind of algorithm is how to obtain the optimal architecture beforehand.

Leerink, Giles, Horne, and Jabri (1995) tested different local and global optimization methods for product unit networks. Their results show that local methods, such as backpropagation, are prone to be trapped in local minima, and that global optimization methods, such as simulated annealing and random search, are impractical for larger networks. They suggested some heuristics to improve backpropagation, and the combination of local and global search methods.

Ismail and Engelbrecht (1999, 2000) applied four different optimization methods to train product unit neural networks: random search, particle swarm optimization, genetic algorithms, and leapfrog optimization. They concluded that random search is not efficient in training these types of networks, and that the other three methods show an acceptable performance in three problems of function approximation with low dimensionality. In a posterior paper (Ismail & Engelbrecht, 2002) they used a pruning algorithm to develop the structure as well as the training of the weights of a product-unit based neural network. Nevertheless, the methods proposed in these papers are only for parametric learning.

The work carried out on PUNNs has not tackled the problem of the design of both the structure and weights of this kind of neural network, either using classic or evolutionary based methods.

3. Neural networks based on product units

In this section, we explain the proposed evolutionary algorithm in depth. We begin with the definition of the family of functions used in the modelling process and its representation by means of a neural network structure.

3.1. Function typology and associated network

Let us consider the functional model

$$f(x_1,x_2,\ldots,x_k) = \beta_0 + \sum_{j=1}^{m} \beta_j \left(\prod_{i=1}^{k} x_i^{w_{ji}} \right)$$

obtained from a product unit neural network. If we expand the second-term, the model can be depicted (Engelbretch, 2003) equivalent to

$$f(x_1,x_2,\ldots,x_k) = \beta_0 + \sum_{j=1}^{m} \beta_j \exp\left(\sum_{i=1}^{k} w_{ji} \ln|x_i| \right) \times \left[\cos\left(\pi \sum_{i=1}^{k} w_{ji} I_i \right) + \mathrm{i} \sin\left(\pi \sum_{i=1}^{k} w_{ji} I_i \right) \right], \quad x_i \neq 0 \tag{8}$$

where

$$I_i = \begin{cases} 0, & \text{if } x_i > 0, \\ 1, & \text{if } x_i < 0. \end{cases} \tag{9}$$

A problem arises with networks containing product units that receive negative inputs and have weights that are not integers. A negative number raised to some non-integer power yields a complex number. Since, neural networks with complex outputs are rarely used in applications, Durbin and Rumelhart (1989) suggest discarding the imaginary part and using only the real component for further processing. This manipulation would have disastrous consequences for the VC dimension when we consider real-valued inputs. No finite VC dimension bounds can in general be derived for networks containing such units (Schmitt, 2001). To avoid this problem, the input domain is restricted, and we consider the set $A = \{(x_1,x_2,\ldots,x_k) \in \mathbb{R}^k : x_i > 0, i = 1,2,\ldots,k\}$.

We define the family of real functions F as

$$F = \bigcup_{m\in\mathbb{N}} F_m = \bigcup_{m\in\mathbb{N}} \left\{ f : A \subset \mathbb{R}^k \to \mathbb{R} : f(x_1,x_2,\ldots,x_k) = \beta_0 + \sum_{j=1}^{m} \beta_j \left(\prod_{i=1}^{k} x_i^{w_{ji}} \right) \right\}, \tag{10}$$

where $\beta_0, \beta_j \in \mathbb{R}, w_{ji} \in \mathbb{R}$, $i = 1,2,\ldots,k$, $j = 1,2,\ldots,m$. Each function of the family can be seen as a polynomial expression of several variables, where the exponents are real numbers.

It is interesting to note that this family of functions can be considered as a generalization of response surfaces (Buchanan, Bagil, Goins, & Philips, 1993; Myers & Montgomery, 2002). For instance, if in (10) the exponents are chosen conveniently, $w_{ji} \in \{0,1,2\}$, we obtain a quadratic response surface of the form

$$f(x_1,x_2,\ldots,x_k) = c_0 + \sum_{i=1}^{k} c_i x_i + \sum_{i=1}^{k} c_{ii} x_i^2 + \sum_{i<j}^{k} c_{ij} x_i x_j. \tag{11}$$

Every function of the family can be represented as a neural network. The network must have the following structure: an input layer with a node for every input variable, a hidden layer with several nodes, and an output layer with just one node. The nodes of a layer have no connections among them, and there are no connections between the input and output layers. Fig. 1 shows the structure of such a network.

The network has k inputs that represent the independent variables of the model, m nodes in the hidden layer, and one node in the output layer. The activation of the jth node in the hidden layer is given by

$$\phi_j = \prod_{i=1}^{k} x_i^{w_{ji}}, \tag{12}$$

where w_{ji} is the weight of the connection between input node i and hidden node j. The activation of the node in the output layer is given by

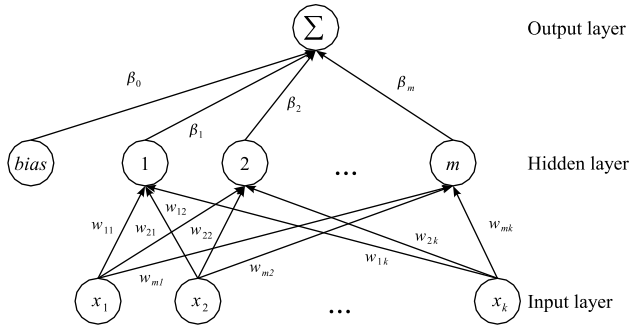

Fig. 1. Model of a product unit based neural network.

$$\beta_0 + \sum_{j=1}^{m} \beta_j \phi_j, \tag{13}$$

where β_j is the weight of the connection between the hidden node j and the output node. The transfer functions of each node in the hidden layer and the output node are the identity function. In this way, each function of the F family is represented by the structure of the network.

3.2. Approximation by product units

Let $\mathbb{R}^n$ be the dimensional Euclidean space and let K be a compact subset of $A \subset \mathbb{R}^n$. The space of continuous functions on K is denoted by $C(K)$ and we use $\|f\|$ to denote the supremum (or uniform) norm of $f \in C(K)$. We represent by F the family of functions $f: K \subset R^n \to \mathbb{R}$ given by (10). Clearly f is continuous on K and therefore F is a subset of $C(K)$.

The main goal of this section is to prove that F is a dense subset of $C(K)$ with respect to the supremum norm, that is, an arbitrary continuous function on K can be approximated by a function of family F. Theorem 1 is a straightforward consequence of the Stone–Weierstrass theorem (Rudin, 1973).

Theorem 1. *The family of functions F is a dense subset in C(K). In other words, given any function $g \in C(K)$ and $\epsilon > 0$, there exists a function $f \in F$ such that*

$$|f(x) - g(x)| < \epsilon, \qquad \forall\, x \in K. \tag{14}$$

Proof 1. The family of continuous functions F is a subalgebra of $C(K)$: if f_1 and f_2 belong to F the sum $f_1 + f_2$, the product $f_1 f_2$, and cf_1 with $c \in \mathbb{R}$ are functions of F. Moreover, the family F verifies the two conditions: constant functions belong to F, and if x, $y \in F$ there exists $f \in F$ such that $f(x) \neq f(y)$. Taking into account that F is a subalgebra of $C(K)$ and that F verifies the previous two conditions, the theorem follows from direct application of the Stone–Weiertrass theorem. □

3.3. Evolutionary algorithm

The population-based evolutionary algorithm for architectural design and estimation of real-parameters has common points with other evolutionary algorithms in the bibliography (Angeline, Saunders, & Pollack, 1994; García-Pedrajas, Hervás-Martínez, & Muñoz-Pérez, 2002; Yao & Liu, 1997). It begins the search with an initial population, and on each iteration the population is updated using a population-update algorithm. The population is subject to the operations of replication and mutation. Crossover is not used due to its potential disadvantages in evolving artificial networks (Angeline et al., 1994). With these features the algorithm falls into the class of evolutionary programming. The general structure of the algorithm is the following:

(1) The initial population B of size N_R is generated.
(2) Repeat until the stopping criterion is fulfilled
   (a) Calculate the fitness of every individual in the population.
   (b) Rank the individuals regarding their fitness.
   (c) The best individual is copied into the new population.
   (d) The 10% percent of best individuals of the population are replicated and substitute the 10% of worst individuals.
   (e) Apply parametric mutation to the 10% of best individuals.
   (f) Apply structural mutation to the rest of the 90% of individuals.

Sections 3.3.1–3.3.5 explain in depth all the aspects of this algorithm.

3.3.1. Generation of the initial population

The algorithm begins with the generation of a larger number of networks than the number used during the evolutionary process. We generate $10N_R$ networks, where N_R is the number of networks of the population during the evolutionary process.

For the generation of a network, the number of nodes in the hidden layer is taken from a uniform distribution in the interval $(0, m/2]$, where m is the maximum number of hidden nodes in any network of the population. In this way, the initial population is formed by models simpler than the most complex model possible. The number of connections between each node of the hidden layer and the input nodes is determined from a uniform distribution in the interval $(0,k]$, where k is the number of independent variables. There is always at least one connection between the hidden layer and the output node. Once the topology of the network is defined, each connection is assigned a weight from a uniform distribution in the interval $[-L, L]$ for the weights between the input and hidden layers, and in the interval $[-M, M]$ for the weights between the hidden layer and the output node. Finally, the initial population that constitutes the base solution B is obtained selecting the best N_R among the $10N_R$ generated.

3.3.2. Fitness assignment

Let $D = \{(\mathbf{x}_l, y_l): l = 1,2,\ldots,n\}$ be the training data set, where n is the number of samples. Here, we assume that each training sample is independent and identically distributed. We consider the mean squared error (MSE) of an individual g of the population

$$\mathrm{MSE}(g) = \frac{1}{n}\sum_{l=1}^{n}(y_l - \hat{y}_l)^2 \tag{15}$$

where the $\hat{y}_l$ are the predicted values. We define the fitness function $A(g)$ by means of a strictly decreasing transformation of the mean squared error

$$A(g) = \frac{1}{1+\mathrm{MSE}(g)}. \tag{16}$$

3.3.3. Parametric mutation

Parametric mutation consists of a simulated annealing algorithm (Geyer & Thompson, 1996; Kirkpatrick & Vecchi, 1983; Otten & Ginneken, 1989), a strategy which proceeds by proposing jumps from the current model according to some user-designed mechanism. The severity of a mutation to an individual g is dictated by the temperature, $T(g)$, given by

$$T(g) = 1 - A(g), \qquad 0 \leq T(g) < 1. \tag{17}$$

Thus, the temperature is determined by the closeness of the function to any solution of the problem.

Parametric mutation is accomplished for each parameter w_{ji}, β_j of (10) with Gaussian noise, where the variance of the normal distribution depends on the temperature. This allows an initial coarse-grained search, and a progressively finer-grained search, as a model approaches a solution to the problem. The exponents of the function, which represent the weights of the connections from an input node to a hidden node, are modified as follows

$$w_{ji}(t+1) = w_{ji}(t) + \xi_1(t), \quad i = 1,\ldots,k, \quad j = 1,\ldots,m \tag{18}$$

where $\xi_1(t) \in N(0,\ \alpha_1(t)T(g))$ represents a one-dimensional normally distributed random variable with mean 0 and variance $\alpha_1(t)T(g)$.

The coefficients β_j, which represent the weights of the connections from a hidden node to the output node, are modified as follows

$$\beta_j(t+1) = \beta_j(t) + \xi_2(t), \qquad j = 1,\ldots,m \tag{19}$$

where $\xi_2(t) \in N(0,\ \alpha_2(t)T(g))$ represents a one-dimensional normally distributed random variable with mean 0 and variance $\alpha_2(t)T(g)$. The modifications of each term are performed step-wise, modifying one term at a time.

Once the mutation is performed, the fitness of the individual g is recalculated and the usual simulated annealing criterion is applied. Thus, if ΔA is the difference in the fitness function before and after the random step, the criterion is: if $\Delta A \geq 0$ the step is accepted, if $\Delta A < 0$ the step is accepted with a probability $\exp(\Delta A/T(g))$. It should be pointed out that the modification of the exponents, w_{ji}, is different from the modification of the coefficients β_j, $\alpha_1(t) \ll \alpha_2(t), \forall t$. Moreover, they are adaptively changed in every generation by some predefined rule. The parameters α_1, α_2, that determine together with the temperature the variance of the distribution, change throughout the evolution, allowing the learning process to adapt. In other papers (Angeline et al., 1994; García-Pedrajas, Hervás-Martínez, & Muñoz-Pérez, 2003) the parameters of the evolution are fixed during the evolutionary process. The adaptation of the parameters tries to avoid being trapped in local minima and to speed up the evolutionary process when the conditions of the searching are suitable. The adaptation of α_1,α_2 is given by

$$\alpha_k(t) = \begin{cases} (1+\lambda)\alpha_k(t) & \text{if } A(g_s) > A(g_{s-1}),\ \forall s \in \{t,t-1,\ldots,t-\rho\} \\ (1-\lambda)\alpha_k(t) & \text{if } A(g_s) = A(g_{s-1}),\ \forall s \in \{t,t-1,\ldots,t-\rho\} \\ \alpha_k(t), & \text{otherwise} \end{cases} \tag{20}$$

where $k \in \{1,2\}$, $A(g_s)$ is the fitness of the best individual, g_s, in generation s, λ and ρ must be set by the user. In our case we have considered $\alpha_1(0) = 1$, $\alpha_2(0) = 5$, $\lambda = 0.1$, and $\rho = 10$.

A generation is defined as successful if the best individual of the population is better than the best individual of the previous generation. If many consecutive successes are observed, this indicates that the best solutions are residing in a better region in the search space. In this case, we increase the mutation strength hoping to find even better solutions closer to the optimum solution. If the fitness of the best individual is constant for several consecutive generations, we decrease the mutation strength. Otherwise, the mutation strength is constant.

3.3.4. Structural mutation

Structural mutation is more complex because it implies a modification of the structure of the function. Structural mutation allows the exploration of different regions of the search space and helps to keep the diversity of the population. There are five different structural mutations: node addition, node deletion, connection addition, connection deletion, and node fusion. These five mutations are applied sequentially to each network. The first four are similar to the mutations of the GNARL model (Angeline et al., 1994). The mutations are performed as follows:

- *Node addition*. One or more nodes are added to the hidden layer. The connection with the output node has a random value. The connections from the input layer are chosen randomly and its values are also random.
- *Node deletion*. One or more nodes are selected randomly and deleted together with their connections.
- *Connection addition*. One or more connections, with random weights, from an input node to a hidden node are added to randomly selected nodes.
- *Connection deletion*. One or more connections between hidden nodes and input nodes are selected and removed.
- *Node fusion*. Two randomly selected nodes, a and b, are replaced by a node c, which is a combination of both. The connections that are common to both nodes are kept, with a weight given by

$$\beta_c = \beta_a + \beta_b, \qquad w_{ic} = \frac{w_{ia} + w_{ib}}{2}. \tag{21}$$

The connections that are not shared by the nodes are *inherited* by c with a probability of 0.5 and its weight is unchanged.

Table 1
Parameters common to all the experiments

Parameter	Value
Population parameters	
Population size N_R	1000
Maximum number of terms m	8
Exponent interval $[-L, L]$	$[-5,5]$
Coefficient interval $[-M, M]$	$[-5,5]$
Parametric mutation parameters	
$\alpha_1(0)$	1
$\alpha_2(0)$	5
β	0.1
ρ	10
Structural mutation's parameters: interval $[\Delta_{\mathrm{MIN}}, \Delta_{\mathrm{MAX}}]$	
Node addition	[1,2]
Node deletion	[1,3]
Connection addition	$[1,2k]$
Connection deletion	$[1,2k]$

For each mutation (excepting node fusion) there is a minimum value, Δ_{MIN}, and a maximum value, Δ_{MAX}, and the number of elements (nodes or connections) involved in the mutation is calculated as

$$\Delta_{\mathrm{MIN}} + \lfloor uT(g)(\Delta_{\mathrm{MAX}} - \Delta_{\mathrm{MIN}}) \rfloor \tag{22}$$

where u is a random uniform variable in the interval [0,1]. For the parametric mutation, the nodes and connections are chosen sequentially in the given order, with probability $T(g)$ in the same generation on the same network. In the structural mutation, the nodes and connections are selected randomly, where the maximum and minimum number of mutations in every case are given by (22).

Table 1 shows the parameters used in the experiments. The algorithm is quite robust regarding small variations in these parameters.

3.3.5. Stop criterion

The stop criterion is reached whenever one of the following three conditions is fulfilled: (i) the algorithm achieves a given fitness; (ii) the values of $\alpha_1(t)$ and $\alpha_2(t)$ are less than 10^{-4}; (iii) there is no improvement for 20 generations either in the average performance of the best 20% of the population or in the fitness of the best individual.

4. Experiments

In the first set of experiments we test our model for five functions. These functions are Friedman's functions #1, #2, and #3, widely used in the literature, and Cherkassky's functions #10 and #11, which are very difficult to approximate with standard sigmoidal neural networks (Cherkassky, Gehring & Mulier, 1996).

Friedman's functions were used by Friedman (1991) in his work on multiplicative adaptive regression splines (MARS). Cherkassky's benchmark functions (Cherkassky, 1996) are high-dimensional functions that include intrinsically low-dimensional functions that can be easily estimated from data. Table 2 shows the definition of these functions.

The evolutionary algorithm was run 30 times with the parameters in Table 1. The number of generations was 2000, 800, 1200, 1000, and 1200, for f_1, f_2, f_3, ch_{10}, and ch_{11}, respectively. The training set has 200 instances randomly generated within the domain defined in Table 2 where $\varepsilon_i \in N(0, \sigma)$. The standard deviation is $\sigma = 1$ for Friedman #1; for the rest of the functions it is set so that the noise is adjusted to give a 3:1 ratio of signal power to noise power, SNR = 3. The generalization set has 1000 instances randomly generated without noise. This experimental design is made to be as similar as possible to other papers to give a fair comparison with their results. For f_2, f_3, ch_{10}, and ch_{11} functions the inputs

Table 2
Definition of the benchmark functions and their domains

Definition	Domain
$f_1(\mathbf{x}) = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + \epsilon_1$	$x_i \in (0,1)$, $i = 1,\ldots,5$
$f_2(\mathbf{x}) = \left(x_1^2 + \left(x_2 x_3 - \frac{1}{x_2 x_4}\right)^2\right)^{1/2} + \epsilon_2$	$x_1 \in [0,100]$, $x_2 \in [40\pi, 560\pi]$, $x_3 \in [0, 1]$, $x_4 \in [1, 11]$
$f_3(\mathbf{x}) = \tan^{-1}\left(\frac{x_2 x_3 - (1/x_2 x_4)}{x_1}\right) + \epsilon_3$	$x_1 \in [0,100]$, $x_2 \in [40\pi, 560\pi]$, $x_3 \in [0, 1]$, $x_4 \in [1, 11]$
$\mathrm{ch}_{10}(\mathbf{x}) = \exp(2x_1 \sin(\pi x_4)) + \sin(x_2 x_3) + \epsilon_4$	$x_i \in [-0.25, 0.25]$, $i = 1,\ldots,4$
$\mathrm{ch}_{11}(\mathbf{x}) = 4(x_1 - 0.5)(x_4 - 0.5)\sin(2\pi\sqrt{x_2^2 + x_3^2}) + \epsilon_5$	$x_i \in [-1, 1]$, $i = 1,\ldots,5$

Friedman's functions #1, #2, and #3: f_1, f_2, and f_3, and Cherkassky's functions #10 and #11: ch_{10} and ch_{11}.

Table 3
Results of MSE_G of the evolutionary model for Friedman #1, #2, and #3 functions, and Cherkassky #10 and #11 functions

	Training				Generalization			
	Mean	SD	Best	Worst	Mean	SD	Best	Worst
f_1	1.1848	0.1340	0.9941	1.4632	0.9360	0.7885	0.1697	3.2444
f_2	41,569.61	1196.68	39,451.80	44,771.48	5538.50	2858.95	1851.39	12,685.90
f_3	0.2697	0.0031	0.2657	0.2808	0.0500	0.0321	0.0098	0.1023
ch_{10}	0.1122	0.0007	0.1111	0.1140	0.0151	0.0016	0.0126	0.0179
ch_{11}	1.9380	0.1443	1.6482	2.3197	2.9957	0.0946	2.8122	3.1846

Table 4
MSE_G results in Roth (2004) for Friedman's functions

	SVM	RVM	LASSO
f_1	2.92	2.80	2.84
f_2	4140	3505	3808
f_3	0.0202	0.0164	0.0192

variables are scaled in the interval [0.1, 0.9], and the output variables in the interval [1,2].

Table 3 shows the learning and generalization results, MSE_G, of the evolutionary model for 30 runs of the algorithm. For each function we show the mean, standard deviation, and best and worst results.

Although comparison with previous results in the literature must be made cautiously, we think it is interesting to show the results using these functions in some previous works to get a clearer idea of the performance level of our model. For f_1, Vapnik (1999) using bagging techniques obtains $MSE_G = 2.20$, with boosting $MSE_G = 1.65$, and with a Support Vector Machine (SVM) $MSE_G = 0.67$. Roth (2004) used a training set of 240 instances and a testing set of 1000 noiseless instances. Their results (mean prediction error averaged over 100 randomly generated 240/1000 training/test splits) using a SVM, a Relevance Vector Machine (RVM), and LASSO regression are shown in Table 4.

Drucker (1997) used 200 training noisy instances and 5000 testing noiseless instances. Results for 10 and 100 runs, using single trees, bagged trees, and boosted trees are shown in Table 5.

Cherkassky (1996) used three different sizes of the training and testing sets. We compare with the medium size, which uses 100 instances. Moreover, they used three different types of noise: no noise, noise with SNR = 2, and noise with SNR = 4. A single data set was generated for each of the two functions, ch_{10} and ch_{11}, consisting of 961 examples randomly sampled in the domain x_i. The performance index used to compare predictive performance (generalization capability) of the methods is the normalized root mean square error, NRMS, which represents the fraction of unexplained standard deviation. Hence, a small value of NRMS indicates good predictive performance, whereas a large value indicates poor performance (the value NRMS = 1 corresponds to the mean response model). Cherkassky using a constrained topological mapping, CTM, and K-nearest neighbors regression, KNN, with generalized memory-based learning, GMBL, obtained values of NRMS = 0.224 for ch_{10} and NRMS = 0.971 for ch_{11}. The values obtained by our model are NRMS = 0.984 for ch_{10} and NRMS = 1.019 for ch_{11}, which shows that these functions are very difficult to be modeled using a neural network.

4.1. Comparison with a multistart Levenberg–Marquardt method

In order to test the efficiency of our algorithm we carry out an experiment to compare it to the performance of the proposed algorithm. We combine a multistart technique with a local search procedure. Thousand random starting points are generated in the search space (the same number as networks of the population of the evolutionary algorithm). The points generated are used as inputs into the local search, and the best solution is recorded. The local search method used is the Levenberg–Marquardt (L–M) algorithm, designed specifically for minimizing a sum-of-squares error. Table 6 shows the results obtained with this method. In several runs the L–M algorithm converged to very poor local minima. These poor local minima have not been considered when averaging the results, as the obtained values would be very large. The table shows the threshold to include a result and the number of times the L–M algorithm failed to reach a useful local minima.

In order to verify whether the differences in mean and variance between the two methodologies are significant we performed three statistical tests using the SPSS (1999) statistical package. A normal distribution can be assumed for all the variables to contrast, except in ch_{10}, because the p-values of a standard Kolmogorov–Smirnov test is greater

Table 5
MSE_G results in Drucker (1997) for Friedman's functions

	10 runs					100 runs	
	Single	Bagging	Boost L	Boost E	Boost S	Bagging	Boosting
f_1	3.58	2.20	1.65	1.67	1.73	2.26	1.74
f_2	29,310	11,463	11,684	10,980	15,615	10,093	10,446
f_3	0.0491	0.0312	0.0218	0.0213	0.0202	0.0303	0.0206

Table 6
MSE_G results of the L–M algorithm using a multistart technique for Friedman #1, #2, and #3 functions, and Cherkassky #10 and #11 functions

	Threshold	Failed/all	Generalization			
			Mean	SD	Best	Worst
f_1	3.5	15/30	2.231	0.854	0.856	3.107
f_2	13,000	16/30	7879.43	3194.27	4044.42	11,467.34
f_3	0.3	8/30	0.032	0.0135	0.015	0.060
ch_{10}	0.25	1/30	0.019	0.002	0.010	0.023
ch_{11}	3.5	13/30	3.061	0.163	2.838	3.496

Table 7
p values of the statistical tests for the comparison of the two methodologies

	Levene	t or U-test
f_1	0.472	0.000
f_2	0.170	0.019
f_3	0.000	0.009
ch_{10}	–	0.000
ch_{11}	0.156	0.084

Table 8
Models of the best network evolved for each function

f_1	$y = 5.248 - 9.122x_3^{0.565} + 9.046x_4^{1.063} + 5.359x_5^{0.828} - 22.755x_1^{3.179}x_2^{3.144} + 9.075x_3^{3.222}x_4^{0.041} + 21.618x_1^{0.756}x_2^{0.733}x_4^{0.021}$
f_2	$y = 0.972 + 0.022x_1^{3.857} + 0.060x_1^{4.416}x_2^{4.454}x_3^{0.678} + 0.539x_1^{-0.132}x_2^{0.931}x_3^{1.089}x_4^{0.018}$
f_3	$y = 1.457 + 0.478x_1^{0.859}x_2^{0.964}x_3^{0.453}x_4^{0.504} + 0.037x_1^{0.802}x_2^{-0.773}x_3^{-0.696}x_4^{4.553} + 0.723x_1^{-1.093}x_2^{4.098}x_3^{7.641}x_4^{2.251}$
ch_{10}	$y = 1.447 - 0.057x_1^{4.258}x_2^{0.177}x_4^{-1.009} + 0.193x_1^{1.963}x_2^{-0.792}x_3^{0.150}x_4^{3.365}$
ch_{11}	$y = 1.398 + 0.269x_4^{0.547} - 0.197x_1^{0.569}x_4^{3.218} - 0.380x_2^{2.086}x_3^{0.430} 0.400x_1^{-0.702}x_2^{4.542}x_3^{4.152} - 0.374x_1^{4.835}x_2^{0.626}x_3^{0.039} + 0.626x_1^{1.893}x_2^{1.413}x_3^{0.197}x_4^{-0.113}$

than 0.05. A Student's t-test or non-parametric Mann–Whitney's U-test was performed for each pair of variables to ascertain whether the differences, in mean or median, among the MSE_G obtained with each methodology were significant. Previously we performed a Levene test for equality of variances. The p-values provided by these tests are shown in Table 7. From these results we can see that the differences between the performance of the two algorithms are significant for functions f_1, f_2, f_3, and ch_{10}. In all of them, with the exception of f_3, the evolutionary algorithm outperforms the L–M algorithm. This result is very good, if we take into account that we removed the worst runs of the L–M experiments, but we considered all the runs of the evolutionary algorithm.

As a final result, we show in Table 8 the equation of the best network for each function.

4.2. Application to microbial growth

As we have stated, our major aim is the development of an evolutionary model of product unit networks that could be successfully applied to regression. Such a network can only be thoroughly evaluated, when it is applied to a complex real-world problem. We have chosen a problem of predictive microbiology with real-world data. This is a very complex task (Baranyi & Roberts, 1995).

Acid lactic bacteria (ALB) are considered the main microorganisms responsible for the deterioration of precooked packed meat products. These bacteria produce lactic acid, slime, and CO_2, which causes strange odors and tastes and affects the product acceptance (Huis in't Veld, 1996). In this work, we study the growth of the ALB microorganism Leuconostoc Mesenteroides, which has been frequently isolated as being responsible for different alterations in meat products (Björkroth & Korkeala, 1996; Mäkelä, 1993).

The collected data consist of 210 curves (García, Hervás, Rodríguez, & Zurera, 2005 Zurera, García, Rodríguez, & Hervás, 2005), representing the growth of the microorganism against time. These curves were adjusted to an exponential model (Baranyi & Roberts, 1994) with the program DMFit 1.0 (József Baranyi, Institute of Food Research, Norwich Research Park, Norwich NR4 7UA, UK). The models that describe the response of one or more kinetic parameters from the primary model are called secondary models. Typically there are three parameters that are interesting from the point of view of the biologist: lag time, growth rate and y_{end}.

The growth rate (grate) is the maximum value of the growth curve slope. The lag-time (lag) is the instant in which the intersection between the line of the maximum slope and the lower asymptote of the growth curve is produced; y_{end} is the value of the asymptote to the curve when the value of t is large enough to be considered as infinite.

We must predict three different dependent variables: lag time (ln lag), growth rate (grate), and the asymptotic signal value (y_{end}). We have four input variables: temperature, pH, NaCl concentration, and $NaNO_2$ concentration. In order to normalize all the variables of the models, we scaled the inputs in the interval [0.1, 0.9] and the outputs in the interval [1,2].

For the comparison of the models, we have used the standard error of prediction (SEP). This is a relative error that is expressed by percentage, and has the advantage of being dimensionless. Its expression is the following

$$\mathrm{SEP} = \frac{100}{|\bar{y}|}\sqrt{\frac{\sum_{l=1}^{N}(y_l - \hat{y}_l)^2}{N}} \tag{23}$$

where y_i is the observed value, $\hat{y}_i$ is the predicted value, and N is the number of samples in the training set. We use SEP_T for

Table 9
SEP values for the three predicted variables for the evolutionary algorithm (EA) model and the L–M algorithm

		Training				Generalization			
		Mean	SD	Best	Worst	Mean	SD	Best	Worst
EA	Grate	4.428	0.874	3.388	6.680	4.550	0.897	3.620	6.891
	ln lag	4.965	0.444	4.316	6.007	5.827	0.285	5.363	6.692
	y_{end}	18.204	1.835	15.677	21.788	21.374	1.298	19.221	25.281
L–M	Grate	6.104	1.262	3.660	8.456	5.779	1.394	3.813	8.778
	ln lag	5.534	0.348	4.828	6.095	6.243	0.313	5.808	6.891
	y_{end}	20.286	2.741	16.267	27.034	22.872	2.014	19.388	27.564

Table 10
p-values of Levene and *t* tests

	Levene	*t*
ln lag	0.013	0.000
Grate	0.788	0.000
y_{end}	0.028	0.001

the SEP over the training set and SEP_G for the set over the generalization set.

Table 9 shows the SEP values of the models obtained with the two methods previously used for Friedman and Cherkassky's functions, our evolutionary algorithm and a multistart algorithm together with L–M. We can see how our model obtains better results for all the predicted variables.

In order to verify the true difference in performance between the two models, we conducted the same three statistical tests of the previous experiments. The tests (see Table 10) used in the difference between the performance of the two models are statistically significant for the prediction of the three variables.

5. Conclusions

In this paper, we have introduced a model for evolving both the weights and the structure of product unit based neural networks. We have applied this model to the problem of function regression. It is well known that these networks can implement higher order functions and we have shown that they are able to approximate any continuous function to an arbitrary degree of accuracy.

Its major disadvantage is that classical local optimization algorithms are very inefficient in optimizing the weights of this kind of network. The error surface is usually extremely convoluted with an increased number of local minima, deep ravines, and wide valleys. The shape of the surface is due to the fact that small changes in the exponents of a given model can cause large changes in the error function. Our model is a valid alternative to these local gradient methods. Moreover, the structure is also optimized during the evolutionary process.

The proposed model for the evolution of this type of networks has the following advantages over the previous models for training this kind of neural network:

- It can obtain both the structure and weights of the neural network. It is very difficult to know beforehand the most suitable structure of the network for a given problem. We usually need a long process of trial and error to obtain a good structure. The evolution of the structure partially alleviates this problem.
- It is a global optimization algorithm. This feature is very interesting as these networks are prone to be trapped in local minima.
- The algorithm does not need a derivable error function, so it can be applied to problems where such an error function is not available.

We have applied our model to five benchmark functions and a real-world problem of prediction of microbial growth. The statistical tests show that the evolved product unit networks obtain better overall results than those obtained with the same type of networks with weights optimized by means of a multistart methods together with a Levenberg–Marquardt algorithm.

Acknowledgements

The authors would like to acknowledge HIBRO Research Group of the University of Córdoba, which obtained the experimental data used in this paper. This work has been supported in part by the Projects TIC2002-04036-C05-02 and TIN2005-08386-CO5-02 of the Spanish *Comisión Interministerial de Ciencia y Tecnología* and FEDER funds.

References

Angeline, P. J., Saunders, G. M., & Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, *5*(1), 54–65.

Arulampalam, G., & Bouzerdoum, A. (2003). A generalised feedforward neural network architecture for classification and regression. *Neural Networks*, *16*, 561–568.

Baranyi, J., & Roberts, T. (1995). Response surface model for predicting the effects of temperature, pH, sodium chloride content, nitrate concentration and atmosphere on the growth of *Listeria monocytogenes*. *Journal of Food Microbiology*, *26*, 199–218.

Baranyi, J., & Roberts, T. A. (1994). A dynamic approach to predicting bacterial growth in food. *International Journal of Food Microbiology*, *23*, 277–294.

Björkroth, K., & Korkeala, H. (1996). Evaluation of lactobacillus sake contamination in vacuum packaged sliced cooked meat products by ribotyping. *Journal of Food Protection*, *59*, 398–401.

Buchanan, R., Bagil, K., Goins, R., & Philips, J. (1993). Response surface analysis for the growth kinetics of *Escherichia coli*. *Food Microbiology*, *10*, 303–315.

Cherkassky, V., Gehring, D., & Mulier, F. (1996). Comparison of adaptive methods for function estimation from samples. *IEEE Transactions on Neural Networks*, *15*(4).

Denison, D. G. T., Holmes, C. C., Mallick, B. K., & Smith, A. F. M. (2002). *Bayesian methods for nonlinear classification and regression*. West Sussex, England: Wiley.

Drucker, H. (1997). Improving regressors using boosting techniques. In D. H. Fisher (Ed.), *Proceedings of the 14th international conference on machine learning* (pp. 107–115). San Mateo, CA: Morgan Kaufmann.

Durbin, R., & Rumelhart, D. (1989). Product units: A computationally powerful and biologically plausible extension to backpropagation networks. *Neural Computation*, *1*, 133–142.

Engelbretch, A. P. (2003). *Computational intelligence: An introduction*. New York: Wiley.

Friedman, J. (1991). Multivariate adaptive regression splines (with discussion). *Annals of Statistics*, *19*, 1–41.

García, R. M., Hervás, C., Rodríguez, M. R., & Zurera, G. (2005). Modelling the growth of *Leuconostoc mesenteroides* by means of an artificial neural network. *International Journal of Food Microbiology, 105, 317–332*.

García-Pedrajas, N., Hervás-Martínez, C., & Muñoz-Pérez, J. (2002). Multi-objective cooperative coevolution of artificial neural networks. *Neural Networks*, *15*(10), 1255–1274.

García-Pedrajas, N., Hervás-Martínez, C., & Muñoz-Pérez, J. (2003). COVNET: A cooperative coevolutionary model for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, *14*(3), 575–596.

Geyer, C. J., & Thompson, E. A. (1996). Annealing Markov chain Montecarlo with applications to ancestral inference. *Journal of the American Statistical Association*, 909–920.

Hornik, K. (1989). Multilayer feedforward neural networks are universal approximators. *Neural Networks*, *2*(5), 359–366.

Huis in't Veld, J. H. J. (1996). Microbial and biochemical spoilage of foods: An overview. *International Journal of Food Microbiology*, *33*, 1–18.

Hwang, J. N., Lay, S. R., Maechler, M., Martin, D., & Schimert, S. (1994). Regression modeling in backpropagation and projection pursuit learning. *IEEE Transactions on Neural Networks*, *5*(3), 342–353.

Ismail, A., & Engelbrecht, A. P. (1999). Training product units in feedforward neural networks using particle swarm optimisation. In V. B. Bajic & D. Sha, (Eds.), *Development and practice of artificial intelligence techniques. Proceeding of the international conference on artificial intelligence* (pp. 36–40). Durban, South Africa.

Ismail, A., & Engelbrecht, A. P. (2000). Global optimization algorithms for training product unit neural networks. In *IEEE international conference on neural networks*. Como, Italy: IEEE Press.

Ismail, A., & Engelbrecht, A. P. (2002). Pruning product unit neural networks. In *IEEE international joint conference on neural networks.* Honolulu, Hawai.

Janson, D. J., & Frenzel, J. F. (1993). Training product unit neural networks with genetic algorithms. *IEEE Expert*, *8*(5), 26–33.

Kirkpatrick, S., Jr. Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680.

Kosko, B. (1991). *Neural networks and fuzzy systems: A dynamical systems approach to machine intelligence*. Englewood Cliffs, NJ: Prentice-Hall.

Lee, S.-H., & Hou, C.-L. (2002). An ART-based construction of RBF networks. *IEEE Transactions on Neural Networks*, *13*(6), 1308–1321.

Leerink, L. R., Giles, C. L., Horne, B. G., & Jabri, M. A. (1995). Learning with product units. In *Advances in neural information processing systems 7* (pp. 537–544). Cambridge, MA: MIT Press.

Lerner, B., Guterman, H., Aladjem, M., & Dinstein, I. (1999). A comparative study of neural networks based feature extraction paradigms. *Pattern Recognition Letters*, *20*(1), 7–14.

Mäkelä, P. (1993). *Lactic acid bacterial contamination at meat processing plants*. Unpublished doctroal dissertation, College of Veterinary Medicine, Helsinki.

Myers, R. H., & Montgomery, D. C. (2002). *Response surface methodology*. New York: Wiley.

Otten, R. H. J. M., & Ginneken, L. P. P. P. (1989). *The annealing algorithm*. Boston, MA: Kluwer.

Roth, V. (2004). The generalized lasso. *IEEE Transactions on Neural Networks*, *15*(1).

Rudin, W. (1973). *Functional analysis*. New York: McGraw-Hill.

Schmitt, M. (2001). On the complexity of computing and learning with multiplicative neural networks. *Neural Computation*, *14*, 241–301.

Specht, D. F. (1991). A general regression neural network. *IEEE Transactions on Neural Networks*, *2*(6), 568–576.

SPSS, I. (1999). *SPSS ©9.0 advanced models*. Chicago, IL: SPSS Inc.

Tomandl, D., & Schober, A. (2001). A modified general regression neural network (MGRNN) with new efficient training algorithms as a robust 'black box'—tool for data analysis. *Neural Networks*, *14*, 1023–1034.

Vapnik, V. (1999). *The nature of statistical learning theory*. Berlin: Springer.

Yao, X., & Liu, Y. (1997). A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, *8*(3), 694–713.

Zhao, Y., & Atkeson, C. G. (1996). Implementing projection pursuit learning. *IEEE Transactions on Neural Networks*, *7*(2), 362–373.

Zurera, G., García, R. M., Rodríguez, M. R., & Hervás, C. (2006). Performance of response surface model for prediction of *Leuconostoc mesenteroides* growth parameters under different experimental conditions. *Food Control 17, 429–438*.