A Hierarchical Genetic Algorithm For The Design Of Beta Basis Function Neural Network

Chaouki Aouiti¹, Adel M. Alimi², Fakhreddine Karray³, Aref Maalej⁴

¹University 7 November Carthage, Faculty of Sciences of Bizerta, Tunisia
chaouki.aouiti@fsb.rnu.tn

²REGIM: Research Group on Intelligent Machines, University of Sfax, ENIS, Department of Electrical Engineering, BP W - 3038, Sfax, Tunisia
adel.alimi@ieee.org

³PAMI: Pattern Analysis and Machine Intelligence Laboratory, Systems design Engineering Department University of Waterloo, Waterloo, ON N2L 3G1, Canada
Karray@uwaterloo.ca

⁴LASEM: Laboratory of Electromechanical Systems, University of Sfax, ENIS, Department of Mechanical Engineering, BP W - 3038, Sfax, Tunisia
aref.maalej@enis.rnu.tn

Abstract - We propose an evolutionary neural network training algorithm for Beta basis function neural networks (BBFNN). Classic training algorithms for neural networks start with a predetermined network structure. Generally the network resulting from learning applied to a predetermined architecture is either insufficient or over-complicated. This paper describes a hierarchical genetic learning model of the BBFNN. In order to examine the performance of the proposed algorithm, they were used for the approximation problems. The results obtained are very satisfactory with respect to the relative error.

I. INTRODUCTION

Artificial neural networks are typically used to generalize an input-output mapping over a set of examples. The first artificial neural network was the perceptron by Rosenblatt in 1957 [10].

Several algorithms, such as back-propagation have been developed for training artificial neural networks. The general procedure is to view training as a search over an n-dimensional parameter space of weights in light of an error function and rely on calculating the gradient of this surface to guide the search. The discovery and popularization of the back-propagation learning rule has strongly stimulated the research on neural networks. However, the standard BP-learning algorithm suffers from the typical handicaps of all steepest descent approaches: Very slow convergence rate and the need for predetermined learning parameters limit the practical use of this algorithm.

Many improved learning algorithms have been reported in literature. Some use heuristic rules to find optimal learning parameters. Other refine the gradient descent method to accelerate convergence. Further approaches employ different nonlinear optimization methods like conjugate gradient, Newton’s method or quasi-Newton techniques [9].

Evolutionary algorithms have been used for the design of artificial neural networks. These algorithms are based on the genetic processes. A population of chromosomes is manipulated and each individual represents a possible solution to the problem. Each chromosome can be assigned a fitness which indicates the quality of the solution that it encodes. The first insights into evolving such networks date back to some of the earliest efforts in evolutionary computation. Friedman in 1956 proposed but didn’t implement an evolutionary search.

Classic training algorithms for artificial neural networks start with a predetermined network structure, and so the response of NN depends strongly on its structure. Generally the network resulting from learning applied to predetermined architecture is either insufficient or over complicated. But in the last few years some researches have developed learning algorithm which incorporate structure selection mechanisms like constructive algorithms and pruning algorithms [12], [13]. A constructive algorithm starts with a minimal network, an artificial neural network with a minimal number of hidden layer, hidden neurons and connections and adds new layers, neurons or connections if it’s necessary, during the training phase. While, a pruning algorithm does the opposite, starts with a maximal network and deletes the unnecessary layers, nodes and connections during training.

Some researches have developed learning GAs that incorporate structure selection mechanisms for neural networks, (Yao and Liu, 1997; Samuel and Gisele, 1998; Billings and Zheng, 1995). Two approaches exists to evolve ANN architectures (Yao and Liu, 1997). In the first approach there is only the evolution of the architectures. If a near optimal architecture is obtained, connection weights are trained. In the second approach, both the architectures and weights are found. For the first category, it is important to choose the information about an architecture that will be coded in the chromosome. An example is to encode all the information (weights, number of layer, the different neurons in each layer,…). This representation is the direct encoding. Another example is to encode only the
most important parameters such that the number of hidden layers.

The present work is the sequence of a series of efforts concerning the application of genetic algorithms for the optimization of Beta Basis Function Neural Networks and generally for feedforward neural networks. In [5] a Beta Basis Function Neural Networks was evolved by a discrete genetic algorithm. In [6] a of Beta Basis Function Neural Networks was evolved by a real genetic algorithm.

GA aren’t used only for the design of artificial neural networks but they are used in many other domain like the fuzzy system, an examples is the work of Rahmouni et al. [14]. Our study is inspired from their work. It focuses on the evolutionary design of Beta Basis Function Neural Networks (BBFNNs) with a hierarchical GA. We propose a hierarchical genetic algorithm for the design of BBFNNs, which incorporates two key ideas; an outer GA to find the optimum number of neurons in the hidden layer of the BBFNN and an inner GA to find the parameter of the BBFNNs with the best number of neurons in the hidden layer. The proposed methodology simultaneously derives optimal structure of the BBFNN and optimal parameters of the BBFNNs. Simulation results show the effectiveness of the proposed method.

The rest of the paper is organized as follows: section 2 is devoted to describe an example of feedforward neural networks it’s the Beta Basis Functions Neural Networks. In section 3, we present a hierarchical genetic algorithm for the design of BBFNNs. Numerical examples and discussion are showing in section 4.

II. BETA BASIS FUNCTION NEURAL NETWORKS

The first idea of using Beta functions for the design of Beta Basis Function Neural Network (BBFNN) that are generated versions of RBFNN was introduced by Alimi in 1997 [4]. The Beta function is used as kernel function for many reasons, such as its great flexibility (see Fig.1) and its universal approximation characteristics ([5], [6]). The Beta function is defined by:

\[
\beta(x) = \begin{cases} 
\frac{(p+q)(x-x_0)}{D_p} + 1 & \text{if } x \in \left[x_0, x_1\right] \\
\frac{(p+q)(x-x_0)}{D_q} + 1 & \text{elsewhere} 
\end{cases}
\]

where \( p > 0, q > 0, x_0 \) and \( x_1 \) are real parameters, and:

\[
x_c = \frac{px_1 + qx_0}{p+q}
\]

\[
x = \frac{x-x_0}{x_c-x_0}
\]

\[
x = \frac{x-x_0}{x_c-x_0}
\]

\[
x = \frac{x-x_0}{x_c-x_0}
\]

(1) and (4) \( \Rightarrow \)

\[
\beta(x) = \begin{cases} 
\frac{(p+q)(x-x_0)}{D_p} + 1 & \text{if } x \in \left[x_0, x_1\right] \\
\frac{(p+q)(x-x_0)}{D_q} + 1 & \text{elsewhere} 
\end{cases}
\]

(5)

In the multi-dimensional case (dimension =N ) and if

\[
X = \left(x_1, x_2, ..., x_N\right); X_0 = \left(x_{01}, x_{02}, ..., x_{0N}\right); X_1 = \left(x_{11}, x_{12}, ..., x_{1N}\right)
\]

\[
Q = \left(p_1, p_2, ..., p_N\right) \text{ and } Q = \left(q_1, q_2, ..., q_N\right)
\]

Then

\[
\beta(X, X_0, X_1, P, Q) = \prod_{i=1}^{N} \beta(x_i - x_{i0}, x_{i1}, p_i, q_i)
\]

Alimi has shown ([3], [4]) that if we have a given continuous real function and for any arbitrary precision, there exists a Beta fuzzy basis function expansion that approximates it.

Fig. 1. Examples of the Beta function in the one and two dimension case

III. THE GENETIC MODEL

Many papers have shown that genetic algorithms are able to find near-optimal solutions to complex problems [11]. In this paper we will present a hierarchical genetic models for the design of BBFNN. We use a discrete representation to code the center \( x_c \), the width \( D, p \) and \( q \) of each beta basis function.

Using genetic terminology in our data representation, the first step is choosing a representation for possible solution. Each chromosome that represents a network is a matrix (see Fig.2). The number of lines in this matrix is equal to the number of variables in the function that we will approximate. For the number of columns, each chromosome that represents a network is coded as a variable length because the hidden layer has a variable
number of neurons, so a sequence of four genes codes a beta, the first gene codes for the center, the second gene codes for the width and the other two gene codes for p and q. The whole chromosome codes for the set of the parameters of the hidden layer. Since the repetitive usage of the same neuron in a neural network didn’t improve the approximation capability of the network, each chromosome is formed by distinct sequences.

\[
\begin{array}{cccc}
X^c & D^c & P^c & Q^c \\
Y^c & D^c & P^c & Q^c \\
\end{array}
\]

Fig. 2. A chromosome that represents a BBFNN with \( N \) hidden neurons (the function has two variables)

The second step is to choose the objective function. If one chooses:

\[
f^*(x) = \sum_{j=1}^{w} \beta_j f(x)
\]

(7)

where \( n_c \) is the number of neurons in the hidden layer, \( \beta_j \) are the kernel functions of the \( j \)-th neuron, \( w_j \) is the \( j \)-th weight and \( f \) is the output of the network. Given a set of data \((x_i,y_i) \) \((i=1,2,...,N)\) one can obtain the connection weights, the parameters of each neuron and the number of neurons by minimizing the following objective function:

\[
f_1(nc,w,x_0,x,1,p,q) = \sum_{i=1}^{N} (y_i - f^*(x_i)) (y_i - f^*(x_i))
\]

(8)

If one uses this function as an objective function [7], the best structure that minimizes this objective function has \( N \) hidden nodes. To provide a trade-off between network performance and network structure the objective function to be minimized can be amended to:

\[
f_2(nc,w,x_0,x,1,p,q) = \left[ \text{Log} \left( N_{\text{max}} - n_c + 1 \right) \right] + \text{Log} \left( n_c - N_{\text{min}} + 1 \right) \left[ \frac{1}{1 + f_1} \right]
\]

(9)

where \( N_{\text{max}} \) and \( N_{\text{min}} \) are the maximum and the minimum number of neurons in the hidden layer. After the choice of the objective function we randomly chose an initial population of \( M \) chromosomes. Each chromosome has between \( N_{\text{max}} \) and \( N_{\text{min}} \) neurons in the hidden layer. In the following, the genetic operators used in the outer and inner algorithm will be described in detail.

A) The Outer Genetic Algorithm

1) The Crossover operators: The most important operator in a genetic algorithm is the crossover operator. This operator recombines the genetic material from two parents into their children. In our outer genetic algorithm we have used two-crossover operators. One of them changes the number of columns of each chromosome so it changes the number of neurons in the hidden layer. The second operator does not change the number of neurons in the hidden layer.

a) The \( R^c \)- Crossover1 operator: For the first operator, after the selection of the two chromosomes for which we will apply this crossover operator, we choose an arbitrary position \( a \) in the first chromosome and a position \( b \) in the second chromosome according to \( a \). Then, we exchange the second part of the two chromosomes. If one child has more than \( N_{\text{max}} \) or less than \( N_{\text{min}} \) neurons in the hidden layer we choose another position.

b) The \( R^c \)- Crossover2 operator: For the second operator, after the selection of the two chromosomes for which we will apply this crossover operator, we choose an arbitrary position \( a \) in the first chromosome and a position \( b \) in the second chromosome according to \( a \).

Let \( \text{Min}_{\text{point}} = \text{Min}(a,b) \). First we change the values of \( a \) and \( b \) to \( \text{Min}_{\text{point}} \). Then, we exchange the second part of the two chromosomes. In this case, we have necessarily first children having the same length as the second chromosome and the second children having the same length as the first chromosome.

2) The \( R^n \)- Mating operator: When we change the position of a neuron in the hidden layer the output of the network didn’t change. For this reason before applying the crossover operator we apply the mating operator. So this operator changes the position of the neurons in the hidden layer (see Fig 3). With this operator the crossover operator will not be a one-point crossover.

3) The \( R^n \)- Mutation operator: The goal of applying the mutation operator is to inject some new information into the population because, generally, the initial population didn’t have all the information that is essential to the solution. We begin by associating to each gene a real value between 0 and 1. If this value is less than the probability of mutation \( P_m \), than the operator mutation is applied to the gene by changing the value of the gene.

4) The \( R^c \)- Addition and The \( R^n \)- Elimination operators.

Classic training algorithms for Beta Basis Function Neural Networks start with a predetermined network structure and the quality of the response of the BBNF depends strongly on its structure. This problem will be solved
when we use the Genetic Algorithms by the two operators \( R^N \). Addition and \( R^N \) - Elimination. The first operator adds a neuron in the hidden layer (see Fig 4) and the second operator eliminates a neuron (see Fig 5).

![Fig. 4](image1.png)

**Fig. 4** The \( R^N \) - Addition operator

![Fig. 5](image2.png)

**Fig. 5** The \( R^N \) - Elimination operator

**B) The Inner genetic algorithm**

1) **The \( R^N \)-Uniform Crossover operator:**

The \( R^N \)-Uniform crossover operator is radically different to the \( R^N \)-crossover1 operator and \( R^N \) - crossover2 operator. Each gene in the _child_ is created by copying the corresponding gene from one or the other parent, chosen according to a randomly generated crossover mask. Where there is a 1 in the crossover mask, the gene is copied from the first parent and where there is a 0 in the mask the gene is copied from the second parent. The process is repeated with the parents exchanged to produce the second offspring. A new crossover mask is randomly generated for each pair of parents. After the crossover operator we apply the mutation operator like in the outer genetic algorithm.

2) **The New_center operator:** The last operator in the inner genetic algorithm is the \( R^N \)-Addition-Elimination operator. Just as \( P_{mu} \) controls the probability of mutation, another parameter, \( P_{ad-el} \) gives the probability that a chromosome will be flipped.

If the chromosome represent a BBFNN with \( N_h \) neurons in the hidden layer.

Let \( X=(x_1,x_2,...,x_N) \) the \( N \) training points and \( Z=(z_1,z_2,...,z_N) \) the desired output.

We find the vector of weights, \( W=(w_1,w_2,...,w_{Nh}) \) between the first and the hidden layer by using the least squares method.

Let \( Y=(y_1,y_2,...,y_N) \) the vector of the output of the BBNNs.

IV. SIMULATION RESULTS AND DISCUSSION
We trained the BB FNN with different set functions to approximate different functions. We tested our hierarchical genetic algorithm on many approximands in the 1-D case, in the 2-D case, in the 3-D case and in the 4-D case. We sampled 100 points of the function in the 1-D case to give a training set. The 2-D case used 10x10=100 samples. The 3-D case used 7x7x7=343 samples. The 4-D case used 5x5x5x5=625 samples.

Below are some examples of test functions we used as approximands:

\[
f_1(x) = 10 \left[ \exp(-5|x|) + \exp(-0.3|x-0.8|) + \exp(x + 0.6) \right]
\]
for \(-1 \leq x \leq 1\)

\[
g_1(x,y) = \begin{cases} 
1 & \text{if } x=0 \text{ and } y=0 \\
\sin x & \text{if } x\neq0 \text{ and } y=0 \\
\sin y & \text{if } x=0 \text{ and } y\neq0 \\
\sin x \sin y & \text{if } x\neq0 \text{ and } y\neq0
\end{cases}
\]
for \(-1 \leq x,y \leq 1\)

\[
h_1(x,y,z) = \left| \sin \left( \frac{\Pi x}{2} \right) \cos \left( \frac{\Pi y}{2} \right) \cos \left( \frac{\Pi z}{2} \right) \right|
\]
for \(-1 \leq x,y,z \leq 1\)

\[
k_1(x,y,z,t) = 10 \left[ \exp(-5|x|) + \exp(-10|x+0.6|) \right] \\
10 \left[ \exp(-5|y|+0.9) \exp(-3|x-0.8|/10) \right] \\
10(z-1)(z-1.9)(t-0.7)(t+1.8)
\]
for \(-1 \leq x,y,z,t \leq 1\)
Fig. 10 a plots the evolution of the maximum, minimum and best number of neurons in the hidden layer for $f_j$. The best BBFNN has 14 neurons in the hidden layer. Fig. 10 b plots the evolution of the maximum, minimum and best number of neurons in the hidden layer for $g_j$. The best BBFNN has 32 neurons in the hidden layer. Fig. 10 c plots the evolution of the maximum, minimum and best number of neurons in the hidden layer for $h_j$. The best BBFNN has 34 neurons in the hidden layer. Fig. 10 d plots the evolution of the maximum, minimum and best number of neurons in the hidden layer for $k_j$. The best BBFNN has 56 neurons in the hidden layer. The results confirm our previous objective owing to show the ability of genetic algorithm to be used for the design of BBFNN. We tested our hierarchical GA for the design of BBFNN on many approximands and it perform well with all functions.

**V. CONCLUSION**

This paper presented a hierarchical GA to train BBFNN. The problem was to find the optimal network structure and the different parameters of the network.

Our genetic models can automatically determine appropriate structures and network parameters of beta basis function neural networks. In order to find the network's optimal structure, the process modifies the number of neurons in the hidden layer. The performance of the algorithm is achieved by evolving the initial population and by using operators that alter the sizes of the networks. This strategy attempts to avoid Lamarckism.

The originality of the genetic model resides on:

- A hierarchical GA for the design of BBFNN
- Our genetic model can automatically determine appropriate structures and network parameters of BBFNN.
- The new genetic operators (the mating operator the new_center operator).
- The design of BBFNN used to approximate a multi-variable function by GA.

**REFERENCE**