Tuning artificial neural networks parameters using an evolutionary algorithm

Leandro M. Almeida, Teresa B. Ludermir
Federal University of Pernambuco – Center of Informatics
P.O. Box 7851, Cidade Universitária, Recife - PE, Brazil, 50732-970
{lma3,tbl}@cin.ufpe.br

Abstract

This paper describes a method to automatically tuning artificial neural networks parameters for a specific problem using an evolutionary algorithm. The method employs an evolutionary search to perform simultaneous tuning of initial weights, transfer functions, architectures and learning rules (learning algorithms parameters). Experiments were performed and the results demonstrate that the method in a shorter time of search, is able to find efficient networks with satisfactory generalization capabilities.

1. Introduction

The power of Artificial Neural Networks (ANNs) has been widely proved through its use with successful in many fields such as pattern recognition, speech recognition, signal processing and function approximation [1, 10]. Even with its success widely proven in the literature and by applications in commercial and industrial fields, the search for an ANN tailored to a specific problem in order to attain success in an application that uses ANNs remains a challenge. Manual tuning (trial-and-error) of ANN parameters for a certain problem is considered a tedious, less productive and error-prone task [1, 2]. A near-optimal ANN is characterized by the choice of its specific and corrected parameters for a specific problem, thereby producing a satisfactory performance [2]. The construction of near-optimal ANN configurations involves difficulties such as the exponential number of parameters that need to be adjusted; the need for *a priori* knowledge of the problem domain and ANN operation in order to define these parameters; and the presence of an expert when such knowledge is lacking [2].

A large number of papers are found in the literature devoted to constructing automatic methods for tuning ANN parameters. These can be categorized as evolutionary and non-evolutionary methods. One kind of evolutionary technique, the Genetic Algorithm (GA) [4], is often used for search near-optimal ANN models with topology optimization, as presented in [6, 7]. Other works include transfer functions, initial weights and learning rules, as presented in [1, 5]. Automatic methods that use non-evolutionary techniques are focused mainly on the manipulation of ANN architectures and weights. Some of these non-evolutionary methods prune connections considered less significant [8, 9] or freeze weights when the same inputs are submitted to the network [8].

In this paper, we present a method denominated GANNTune (GA + ANN + Tune), which is an evolution of its antecessor NNGA-DCOD (aNN + GA - Direct enCODe), presented in [2]. The NNGA-DCOD method is mainly characterized by the adoption of Evolutionary ANNs (EANNs), a framework that makes possible the search for all ANN components needed for its operation, as defined by Xin Yao [10]. EANNs include a sequential and nested layer search process, in which each layer has specific ANN information to be found by a specific GA. In NNGA-DCOD the search for initial weights is performed in the lower layer; the search for hidden layers, nodes per layer (architecture) and transfer functions occurs in the intermediate layer; and the search for learning algorithm parameters is performed in the higher layer. Thus, there are three GAs for searching these ANN parameters and, consequently, there is dependence between this GAs. While the NNGA-DCOD has three nested GAs the GANNTune has only one GA. The aim of this approach is to reduce the time needed to perform the search for ANN parameters for a specific problem and preserve the NNGA-DCOD capability of finding solutions (ANN parameters) with similar or better quality. This paper is organized as follows: Section 2 presents the GANNTune method; Section 3 describes experimental results, including time consumption analysis; Section 4 summarizes our conclusions and presents future work.

2. Developed evolutionary search methodology

GA is a kind of Evolutionary Algorithm (EA) that is widely employed to build automatic methods for tuning ANN parameters. The GANNTune is made up of a GA

978-0-7695-3326-1/08 $25.00 © 2008 IEEE
DOI 10.1109/HIS.2008.117

working directly with ANN parameters in their natural format. In other words, there is no encoding scheme. An individual is composed of all ANN parameters needed for its operation: learning algorithm parameters, number of hidden layers and hidden nodes per layer, transfer functions and initial weights. The traditional genetic operators have been reformulated to deal with such kinds of values. These operators are the main contribution of this work, as there is an absence of such types of genetic operators in the literature. The tournament selection operator is used to select individuals to compose a new population (survivors) as well as individuals for recombination.

The execution of the genetic operator for recombination between individuals can be divided into three phases: In the first, the recombination of learning algorithm information occurs; in the second phase, the recombination of architecture information occurs; and lastly the recombination of initial weights occurs. This separation is also applied to the mutation operator. The recombination of learning algorithm information occurs with the generation of new values within a range based on the parent values. Based on probability, the mutation algorithm for the child performs an addition or subtraction operation in the child rates. Based on its current value, this operation causes a forty percent up or down (increasing or decreasing) change in the parameter. The evolutionary search for learning rules or learning algorithms occurs with the search for parameters from the following algorithms: Back-Propagation (BP), Levenberg- Marquardt (LM), Quasi-Newton Algorithm (QNA) and Scaled Conjugate Gradient (SCG).

The recombination of architectures considers configurations with between one and three hidden layers and between one and 12 hidden nodes per layer. Table 1 displays the transfer functions in the search process. The selected individuals ("parents") are submitted to the recombination process. The process starts with the definition of the number of hidden layers that the child will have, which is obtained through the mean sum of hidden layers from the parents, rounded off based on probability problems. The dimensions of the hidden layers and the transfer functions for the child are then defined through random selection that considering all parent information, hidden layers and transfer functions.

The crossover/recombination operator for architectures produces individuals that are very similar to their parents. Thus, the mutation operator (Algorithm 1) is applied after the crossover in order to maintain diversity in the population. Child selection for mutation is performed through a random process. The process starts with the number of children that will undergo mutation. A random number is then generated within the range $[0, 1)$. This value will be used to define whether an architecture will undergo an increase or decrease in the number of hidden layers or will undergo an increase ($pInc = 0.6$) or decrease ($pDec = 0.5$) in the number of hidden nodes per layer. If an architecture has only one hidden layer, the possibility of the number being three is great within all the possibilities. In the case of a selected random value failing to cause a significant change in the number of hidden layers, a set of values between $arqval = [-2, 5)$ is generated and added to the current dimensions of the architecture. This range was defined empirically with the purpose of maintaining diversity among the individuals. If an architecture has two hidden layers, the possibility of the number being three is great, with the intention of reducing the number of hidden nodes per layer.

Algorithm 1: Mutation architecture information

```
   Data: childVet, n, m //child vector, number of child, mutation rate
   Result: mutChild
1  begin
2      childNumber ← ((m/100) * n)
3      while childNumber > 0 do
4          probability ← rand(1)
5          child ← raffle(childVet)
6          if probability ≥ pInc then
7              if child.numHiddLy = 1ou = 2 then
8                  child ← addLayers(3)
9              else
10                 child ← decreaseLayersTo(2)
11         else if probability ≥ pDec then
12             if hild.numHiddLy = 1ou = 2 then
13                 child ← addLayers(1)
14             else
15                 filho ← decreaseLayersTo(2)
16         else
17             child ← child + raffle(arqval)
18         childVet ← childVet − child;
19         mutChild ← child
20         childNumber ← childNumber − 1
21     mutChild ← mutChild + childVet
22 end
```

The recombination of initial weights also considers the information from two parents, but the parents can have different information regarding architecture, they have matrices of weights with different dimensions as well. For this reason, the recombination starts with generation of the initial weights randomly based on the architecture description $arch$. After random generation of the initial weights, an insertion of genetic material from parents occurs, since the previous conditions have been satisfied. One of these conditions refers to the number of weight matrices (nm). The transfer of genetic material occurs up to the same number of matrices in the parents and child, and up to the minimal dimension of a matrix. Algorithm 2 describes the recombination of initial weights in more detail. The procedure $getDim$ has the capability of identifying the dimensions of all matrices (number of lines and columns). The parameters $pflip = 0.5$ controls the source of genetic material to be transferred to child.

The selection of individuals for mutation is performed randomly, as individuals have no fitness yet. According to

Algorithm 2: Recombining initial weights

Data: $\mathbf{prt}_a$, $\mathbf{prt}_b$, $arch$ //parent A, parent B, architecture description
Result: **child** //new offspring

begin
  $weightsrange = [-0.5, 0.5)$
  $[weights] = getInitialWeights(arch, weightsrange)$
  for $id = 1 : weights.nm$ **do**
    if $\mathbf{prt}_a.nm >> index >> \mathbf{prt}_a.nm$ **then**
      $[dim] = getDim(\mathbf{prt}_a\{id\}, \mathbf{prt}_b\{id\}, weights\{id\})$
      $linmin = min([dim.lina, dim.linb, dim.linc])$
      $colmin = min([dim.cola, dim.colb, dim.colc])$
      for $e = 1 : linmin$ **do**
        for $g = 1 : colmin$ **do**
          if $rand(1) < p_{flip}$ **then**
            $weights.matrices\{id\}(e, g) = \mathbf{prt}_a.wgts\{id\}(e, g)$
          else
            $weights.matrices\{id\}(e, g) = \mathbf{prt}_b.wgts\{id\}(e, g)$
  child $\leftarrow weights$
end

Parameters for:		Values
GAs	- Encoding / Selection	Direct / Tournament
	- Elitism / Recomb. / Mutation / Pressure	10% / 60% / 40% / 30%
	- Population size / Generations	50 / 100
ANNs	- Type	MLP, feedforward
	- Transfer functions	Pure-linear (P), Tang-sigmoid (T), Log-Sigmoid (L)
	- Hidden layers / Nodes / Train. epochs	up to: 3 / 16 / 3
	- Range of initial weights	[-0.5, 0.5)
	- Output neuron	linear
Train. algorithms	BP - Learning rate and momentum	[0.05, 0.25)
	LM - Learning rate	[0.001, 0.02)
	SCG - Step lengths	[1.0E-06, 100)
	- Limits on step sizes	[0.1, 0.6)
	QNA - Scale factor to determine performance	[0.001, 0.003)
	- Scale factor to determine step size	[0.001, 0.02)
	- Change in weight for 2nd deriv. approx.	[0, 0.0001)
	- Regulating the indefiniteness of the Hessian	[0, 1.0E-06)

Table 1. GANNTune parameters.

the mutation rate $p_m = 0.4$, forty percent of the child will undergo mutation and this mutation will affect forty percent of its composition as well. Sparse matrices are generated for individuals selected for mutation, with forty percent of the values between $fx = [-0.5, 0.5)$ and the remaining values at zeros. This range of values was also found empirically. These sparse matrices are then added to the child weight matrices. The fitness for is calculated based on the ANN Mean Squared Error (MSE) achieved in the training set.

3. Experiments

The experiments to evaluate the GANNTune were performed with three well-known classification problems found in the UCI repository [3]. Cancer (ca) with 9 attributes (atb), 699 examples (exp) and 2 classes (cla); Heart-Cleveland (he) with 35 atb, 303 exp and 2 cla; and Pima-diabetes (pi) with 8 atb, 768 exp and 2 cla. To perform the experiments, we used 30 two-fold iterations. At each iteration, data were randomly divided into halves. One half was the input for the algorithms (70% for training and 30% for the validation set) and the other half was used to test the final solution (test set). The execution of one iteration corresponds to the creation of an initial population and execution of evolutionary search for 100 generations. After 30 iterations with different data division and initial populations, the best 30 ANNs parameters are chosen based on training error, test error and architecture size. The results reported are the mean value from the 30 ANNs parameters found for each classification problem. The search for ANNs through trial-and-error was performed following the previously described methodology, using the same database split scheme and number of training epochs. We performed 30 runs in each fold for the following network setup $[2, 24)$ hidden neurons for one hidden layer with the (T) transfer function.

Prob./ Algts.		GANNTune				Trial-and-error		
		Time in min.	Arc.	Errors		Arc.	Errors	
				Train.	Test		Train.	Test
Cancer	BP	10.26	3.7	2.38	2.54	16.8	23.16	23.14
	LM	13.12	3.5	2.37	2.53	11.4	1.91	2.67
	SCG	8.91	8.7	2.31	2.48	15	3.48	3.55
	QNA	10.26	3.6	2.21	2.85	7.4	3.39	3.53
Pima	BP	8.37	10.3	17.65	17.25	15.4	23.97	23.96
	LM	17.53	19.6	12.64	15.87	15	14.09	16.13
	SCG	8.85	13.2	15.83	16.9	12.4	18.13	18.51
	QNA	9.67	7.5	15.66	16.19	20.8	17.59	18.40
Heart	BP	8.21	7.3	12.72	13.87	21.2	24.62	24.64
	LM	54.96	24.4	3.58	16.65	16.4	6.48	13.14
	SCG	8.59	11	11.76	13.54	15.8	12.28	12.96
	QNA	12.49	4.5	11.38	13.50	17.8	12.04	12.89

Table 2. Near-optimal ANNs found through NNGA-DCOD and trial-and-error.

Table 2 displays the mean values from near-optimal ANNs found with GANNTune and trial-and-error methods. The mean of the architectures (*arc.*) is based on the amount of hidden units/neurons. Regarding the number of hidden neurons, GANNTune found structurally better networks (with few hidden neurons) than most of those found with the trial-and-error method. Moreover, with application of the t-test, the results of GANNTune were also statistically better than the manual method for all problems using the BP algorithm. In the case of Pima-Diabetes problem, the GANNTune found structurally better networks with the QNA algorithm as well.

Considering the figures in Table 3, which are mean values, GANNTune using SCG is better than the other methods for searching near-optimal ANNs for the Pima-diabetes problem regarding mean error (and similar to its antecessor NNGA-DCOD), but the amount of nodes is high compared to the other methods. For the Pima-diabetes problem, the GANNTune tuning ANN parameters with the BP algo-

rithm achieved a much better mean error performance than all methods, including the NNGA-DCOD with BP. Comparing the GANNTune to NNGA-DCOD, the reduction in the mean performance for all problems when using the BP algorithm is very expressive, but with GANNTune, the same does not occur when the number of nodes/connections is observed.

The methods found in the literature on searching for near-optimal ANNs did not study the time consumption of the search methodology. Table 3 also displays information on the time GANNTune needs to search for near-optimal ANNs for each problem using all training algorithms. Time is related to the search performed in one fold of the each problem in which the evolutionary search runs up to 100 generations. Therefore the figures in Table 3 are mean values from 30 iterations in minutes. The main drawback of methods that use evolutionary techniques (such as GAs) is the long search processing time. This is due to the fact that GAs consider a large search space (where near-optimal ANNs can be found) and, consequently, require a large amount of time to explore this search space [5].

Considering the time consumption of the NNGA-DCOD method, in which was measured in hours of processing, the time consumption of GANNTune applied to the same classification problems is displayed in minutes of processing. This very significant reduction in time consumption was obtained with the adoption of only one GA to perform the search, whereas the NNGA-DCOD has three nested GAs to perform the same search. The greatest amount of processing time was expended with training algorithms that use second-order information. This can be explained by the fact that such algorithms require complex calculus to adjust the weights and, consequently, require more memory and processing time. The search with BP was faster because the training algorithm adjusts the weights in a simple way, requiring less memory and processing time. The search for ANNs in the Glass problem was faster due to the smaller amount of examples, whereas the search was slower for the Horse problem due to the greater number of attributes.

Information	Problems	Methods						
		GANNTune with BP	GANNTune with SCG	NNGA-DCOD with BP	NNGA-DCOD with SCG	CovNet [7]	CoopNN [6]	CNNDA* [8]
Errors	ca	2.54	2.48	6.22	3.20	-	1.38	1.15
	he	17.25	16.90	21.15	16.90	19.90	21.35	19.91
	pi	13.87	13.54	14.26	12.62	14.26	12.79	-
Nodes	ca	3.7	8.7	12.8	4.5	-	5.89	3.5
	he	10.3	13.2	6.1	2	6.17	7.9	4.3
	pi	7.3	11	12.1	5.1	4.77	7.28	-
Conn.	ca	38.6	88.7	140.8	49.5	-	58.30	35.8
	he	101.9	131.7	61	20	29.43	76.32	40.4
	pi	191.8	384.1	447.7	188.7	33.07	90.31	-

Table 3. Comparison between evolutionary and non-evolutionary(*) methods.

4. Conclusions

In this paper, we presented a new version of a hybrid method for automatically searching near-optimal ANNs. This method is composed of a combination of GAs and ANNs that search different ANN information. Experiments were performed and the results demonstrate that this method is able to achieve compact neural networks with satisfactory performances when compared to a simulation of the manual search method. Comparisons with other recent methods in the literature on searching for ANNs were carried out and the GANNTune achieved the best results regarding error performance for some problems. Furthermore, the method overall requires just 14.44 minutes to perform the search for ANN parameters (compared to 33.63 hours for its antecessor). In other words, the reduction in time consumption is clear. Future work will concentrate on refining of the genetic operators and as well as addressing the connection issues. The authors would like to thank CNPq (Brazilian Research Council) for their financial support.

References

[1] A. Abraham. Meta learning evolutionary artificial neural networks. *Neurocomputing*, (56):1–38, 2004.

[2] L. M. Almeida and T. B. Ludermir. An improved method for automatically searching near-optimal artificial neural networks. *International Joint Conference on Neural Networks (IJCNN 2008)*, 2008.

[3] A. Asuncion and D. Newman. UCI machine learning repository, 2007.

[4] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

[5] K. P. Ferentinos. Biological engineering applications of feedforward neural networks designed and parameterized by genetic algorithms. *Neural Networks*, 18(7):934–950, 2005.

[6] N. García-Pedrajas, C. Hervás-Martínez, and D. Ortiz-Boyer. Cooperative coevolution of artificial neural network ensembles for pattern classification. *IEEE Trans. Evolut. Computation*, 9(3):271–302, 2005.

[7] N. García-Pedrajas, D. Ortiz-Boyer, and C. Hervás-Martínez. Cooperative coevolution of generalized multilayer perceptrons. *Neurocomputing*, 56:257–283, 2004.

[8] M. M. Islam and K. Murase. A new algorithm to design compact two-hidden-layer artificial neural networks. *Neural Networks*, 14(9):1265–1278, 2001.

[9] L. Ma and K. Khorasani. New training strategies for constructive neural networks with application to regression problems. *Neural Networks*, 17(4):589–609, 2004.

[10] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.