An Evolutionary Approach for Tuning Artificial Neural Network Parameters

Leandro M. Almeida and 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. The widespread use of artificial neural networks and the difficult work regarding the correct specification (tuning) of parameters for a given problem are the main aspects that motivated the approach purposed in this paper. This approach employs an evolutionary search to perform the simultaneous tuning of initial weights, transfer functions, architectures and learning rules (learning algorithm parameters). Experiments were performed and the results demonstrate that the method is able to find efficient networks with satisfactory generalization in a shorter search time.

Keywords: Evolutionary algorithms, genetic algorithms, artificial neural network parameterization, initial weights, architectures, learning rules.

1 Introduction

There are a large number of articles in the literature devoted to the analysis and/or usage of Artificial Neural Networks (ANNs). Applications in commercial and industrial fields have contributed more strongly toward proving the importance, efficiency and efficacy of ANNs [9]. The success of ANN usage in fields such as pattern recognition, signal processing, etc. is incontestable [1], but a key problem concerning neural networks usage in practice remains as a challenge [2]. This problem is related to correctly building neural networks specially tailored to a specific problem, which is an extremely important task for attaining success in an application that uses ANNs [2], [3]. The search for ANN parameters is generally performed by a developer through a trial-and-error procedure. Thus, optimality or even near-optimality is not ensured, as the space explored is only a small portion of the entire search space and the type of search is rather random [6].

Manual tuning (trial-and-error search) of ANN parameters for a certain problem is considered a tedious, under-productive, error-prone task [1], [2], [3]. When the complexity of a problem domain increases and when near-optimal networks are desired, manual searching becomes more difficult and unmanageable [1]. An optimal neural network is an ANN tailored to a specific problem, thereby having a smaller architecture with faster convergence and a better generalization performance [1], [2], [3]. A near-optimal ANN is characterized by the choice of its specific, corrected parameters for a specific problem, thereby producing a satisfactory performance [2], [3]. The

E. Corchado, A. Abraham, and W. Pedrycz (Eds.): HAIS 2008, LNAI 5271, pp. 156–163, 2008.

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 functioning in order to define these parameters; and the presence of an expert when such knowledge is lacking [2], [3].

Considering the problems and difficulties related to the use of the manual method for tuning ANN parameters, the adoption of an automatic method for performing this tuning task emerges with the aim of avoiding such problems. 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) [5], is often used to search near-optimal ANN models with topology optimization, as presented in [7], [8]. Others include transfer functions, initial weights and learning rules, as presented in [1], [6]. 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 [10], [11] or freeze weights when the same inputs are submitted to the network [10].

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 functioning, as defined by Xin Yao [13]. 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 lowest 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 highest layer. Thus, there are three GAs for searching these ANN parameters and, consequently, there is interdependence between these GAs [3]. 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. This approach also seeks to 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; and Section 4 summarizes our conclusions and presents future work.

2 Evolutionary Approach for Tuning ANN Parameters

There are many different variants of Evolutionary Algorithms (EA) [5]. GA is a kind of EA that is widely employed to build automatic methods for tuning ANN parameters. The GANNTune is made up of a GA working directly with ANN parameters in their natural format. In other words, there is no encoding scheme. This approach was adopted with the intention of avoiding the frequent encoding and decoding tasks at each iteration of the search process, which occurs when the canonical GA is used [5]. Therefore, an individual is composed of all ANN parameters needed for its functioning: learning algorithm parameters, number of hidden layers and hidden nodes per layer, transfer functions and initial weights.

The individuals used by the GA for tuning ANN parameters are composed of real, integer and discrete information. Traditional genetic operators have been reformulated to deal with these 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 selection of N individuals is performed using the tournament strategy, with a random pressure rate of $P_p = 0.3$ and an elitism rate of $E = 0.1$. This information was found empirically. The tournament selection operator is used to select individuals to compose a new population (survivors) as well as individuals for recombination.

In recombination, given two selected individuals, a random variable is drawn from [0,1) and compared to P_c. If the value is smaller than P_c, one offspring is created via the recombination of the two parents; otherwise, the offspring is created asexually by copying one of the parents [5]. The crossover rate used in this work was $P_c = 0.6$. After recombination, a new random variable is drawn from [0,1) and compared to P_m. If the value is smaller than P_m, the mutation operator is applied to the individual; otherwise, nothing undergoes any changes. The mutation rate used is $P_m = 0.4$.

The execution of the genetic operator for recombination (and mutation) between individuals can be divided into three phases: First, the recombination of the learning algorithm information occurs; then the recombination of architecture information occurs; and, lastly, the recombination of initial weights occurs. The recombination of the 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 occurs with the search for the Back-Propagation (BP) algorithm parameters.

The recombination of architectures considers configurations with between one and three hidden layers and between one and 12 hidden nodes per layer. The selected individuals ("parents") are submitted to the crossover 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 considers all parent information.

The recombination operator for architectures produces individuals that are very similar to their parents. Thus, the mutation operator (algorithm in Figure 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 (P_{Inc} = 0.6) or decrease ($P_{Dec} = 0.5$) 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 *archval* = [-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.

```
Data: childVet, n, m //children vector, number of children, mutation rate
Result: mutChild
Begin
    childNumber ← ((m/100) * n)
    while childNumber < 0 do
        probability ← rand(1)
        child ← raffle(childVet)  //choose randomly one child
        if probability <= P_inc then
            if child.numHidLyrs = 1 or 2 then
                child ← addLayers(3)
            else
                child ← reduceLayersTo(2)
        else if probability <= P_dec then
            if child.numHidLyrs = 1 or 2 then
                child ← addLayers(1)
            else
                child ← reduceLayersTo(2)
        else
            child ← child + rand(archval)
        childVet ← childVet – child
        mutChild ← child
        childNumber ← childNumber – 1
    mutChild ← mutChild + childVet
end
```

Fig. 1. Algorithm used for architectures mutation

In the recombination of initial weights, 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 the parents occurs, as 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. The algorithm in Figure 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 the mutation rate $P_m = 0.4$, forty percent of the children will undergo mutation. The mutation operator for initial weights consists of its adjustment (training) by 5 epochs with the BP algorithm using the data for training.

The method starts the search by randomly generating populations of individuals. Fitness is then calculated based on the ANN Mean Squared Error (MSE) achieved in the training set. The genetic operators maintain the diversity of individuals for the tournament search and a small range of random selection, where the new individuals generated for the next offspring must be distinct from individuals in the present offspring. The amount of individuals used in the GANNTune is small, but, as this method is iterative,

```
Data: prta, prtb, arch //parent A, parent B, architecture description
Result: child //new offspring
Begin
    weightsrange ← [-0.05, 0.05)
    weights ← getInitialWeights(arch, weightsrange)
    for id ← 1:weights.nm do
        if prta.nm < index < prtb.nm then
            dim ← getDim(prta{id}, prtb{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) < pflip then
                        weights.matrices{id}(e,g) ← prta.wgts{id}(e,g)
                    else
                        weights.matrices{id}(e,g) ← prtb.wgts{id}(e,g)
end
```

Fig. 2. Algorithm used for initial weights recombination

Table 1. Individual composition description

	Parameters for:	Values
Genetic Algorithm	Encoding	Direct
	Elitism	10%
	Recombination	80%
	Mutation	70%
	Pressure	30%
	Selection / Stopping criteria	Tournament / Max generations
	Population size	50
	Generation	100
ANNs	Type	MLP, Feed-forward
	Transfer functions	Pure-linear (P), Tang-sigmoid (T) and Log-Sigmoid (L)
	Hidden layers	Up to 3
	Hidden nodes per layers	Up to 16
	Training epochs	Up to 3
	Range of initial weights	[-0.05, 0.05)
	Output neuron	Linear
	Learning rate	[0.05 0.25)
	Momentum rate	[0.05 0.25)

many new individuals are created at every generation of each kind of search. Thus, the search space explored is large and satisfactory results are achieved.

3 Experimental Results

The experiments for evaluating the GANNTune were performed with seven well-known classification problems found in the UCI repository [12]. Cancer with 9 attributes (atb), 699 examples (exp) and 2 classes (cla); Heart-Cleveland with 35 atb, 303

exp and 2 cla; Pima-diabetes with 8 atb, 768 exp and 2 cla; Horse with 58 atb, 364 exp and 3 cla; Glass with 9 atb, 214 exp and 6 cla; Card with 51 atb, 690 exp and 2 cla; and Soybean with 82 atb, 683 exp and 19 cla. To perform the experiments, we used 30 two-fold iterations [4]. At each iteration, data were randomly divided into halves. One half was the input for the algorithm (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 ANN parameters are chosen based on training error, test error and architecture size. The results reported are the mean value from the 30 ANN 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 network having between [2, 24) hidden neurons for one hidden layer with the (T) transfer function.

Table 2 displays the mean values from near-optimal ANNs found with GANNTune and trial-and-error methods. The mean of the architectures (Arch.) is based on the amount of hidden units/neurons. Regarding the number of hidden neurons, GANNTune found structurally worse networks (with many hidden neurons) than most of those found with the trial-and-error method. However, with application of the *t*-test, the results of GANNTune were statistically much better than the manual method for all problems considering the training and test errors.

Table 2. Description of near-optimal ANNs found with GANNTune and trial-and-error methods and time consumption of the developed method

Classification Problems	GANNTune				Trial-and-error		
	Time cons. in minutes	Arch.	Errors		Arch.	Errors	
			Training	Test		Training	Test
Cancer (ca)	14.36	25.4	3.03	3.63	16.8	7.16	8.14
Heart (he)	14.10	20.5	10.03	14.93	21.2	17.62	24.54
Pima (pi)	14.63	26.6	16.54	16.99	15.4	23.97	23.96
Horse (ho)	15.05	24.5	12.35	17.79	20	19.71	19.72
Glass (gl)	14.50	27.3	9.62	10.27	11.7	13.93	13.94
Card (cd)	14.65	18.6	8.55	12.35	12.3	13.03	16.07
Soybean (sy)	17.65	38.9	4.76	4.79	22.8	20.47	32.35

Considering the figures in Table 3, which are mean values, GANNTune is better than the other methods for searching near-optimal ANNs for the Pima-diabetes, Glass, Soybean and Horse problems regarding mean error (and similar to its antecessor NNGA-DCOD for the Horse problem), but the amount of nodes is high compared to the other methods. Comparing the GANNTune to NNGA-DCOD, the reduction in the mean performance for all problems is very expressive, but with GANNTune, the same does not occur with the number of nodes/connections.

None of the methods found in the literature on searching for near-optimal ANNs studied the time consumption of the search methodology. The second column in Table 2 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 second column of Table 2 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 considers a large search space (where near-optimal ANNs can be found) and, consequently, requires a large amount of time to explore this search space [6].

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

Information	Problems	Methods						
		GANNTune	NNGA-DCOD [2]	GEPNET [8]	COVNET [8]	MOBNET [8]	COOPNN [7]	CNNDA* [10]
Test error	ca	3.63	6.22	-	-	-	1.38	1.15
	he	14.93	14.26	13.63	14.26	13.63	11.96	-
	pi	16.99	21.15	19.27	19.90	19.84	19.69	19.91
	ho	17.79	17.52	-	-	-	26.74	-
	gl	10.27	12.75	35.16	-	35.16	22.89	-
	cd	12.35	-	-	-	-	12.17	-
	sy	4.79	-	-	-	-	7.61	-
Number of hidden nodes	ca	25.4	12.8	-	-	-	5.89	3.5
	he	20.5	12.1	6.37	4.77	11.4	7.28	-
	pi	26.6	6.1	4.57	6.17	7.9	7.9	4.3
	ho	24.5	7.2	-	-	-	20.3	-
	gl	27.3	5.6	6.33	-	14.87	6.73	-
	cd	18.6	-	-	-	-	6.89	-
	sy	38.9	-	-	-	-	19.42	-

Compared to the time consumption of the NNGA-DCOD method, which is 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.

4 Conclusions

The search for ANNs that are custom-tailored to a specific problem is considered a complex task and has mainly employed manual search methods. 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 neural networks with satisfactory performances when compared to a simulation of the manual search method and other recent methods described in the literature. Moreover, the overall method requires just 15 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 the genetic operators and addressing the issues of pruning connections.

Acknowledgments

The authors would like to thank CNPq (Brazilian Research Council) for their financial support.

References

1. Abraham, A.: Meta learning evolutionary artificial neural networks. Neurocomputing 56, 1–38 (2004)
2. Almeida, L.M., Ludermir, T.B.: Automatically searching near-optimal artificial neural networks. In: 15th European Symposium on Artificial Neural Networks, pp. 549–554 (2007)
3. Almeida, L.M., Ludermir, T.B.: An improved method for automatically searching near-optimal artificial neural networks. In: International Joint Conference on Neural Networks (2008)
4. Cantú-Paz, E., Kamath, C.: An empirical comparison of combinations of evolutionary algorithms and neural networks for classification problems. IEEE Transactions on Systems, Man, and Cybernetics, Part B 35(5), 915–927 (2005)
5. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer, Heidelberg (2003)
6. Ferentinos, K.P.: Biological engineering applications of feed-forward neural networks designed and parameterized by genetic algorithms. Neural Networks 18(7), 934–950 (2005)
7. García-Pedrajas, N., Hervás-Martínez, C., Ortiz-Boyer, D.: Cooperative co-evolution of artificial neural network ensembles for pattern classification. IEEE Transaction on Evolutionary Computation 9(3), 271–302 (2005)
8. García-Pedrajas, N., Ortiz-Boyer, D., Hervás-Mart´ınez, C.: Cooperative co-evolution of generalized multilayer perceptrons. Neurocomputing 56, 257–283 (2004)
9. Haykin, S.: Neural Networks: A Comprehensive Foundation. Prentice-Hall, Englewood Cliffs (1999)
10. Islam, M.M., Murase, K.: A new algorithm to design compact two-hidden-layer artificial neural networks. Neural Networks 14(9), 1265–1278 (2001)
11. Ma, L., Khorasani, K.: New training strategies for constructive neural networks with application to regression problems. Neural Networks 17(4), 589–609 (2004)
12. Asuncion, A., Newman, D.: UCI machine learning repository, University of California, Irvine, School of Information and Computer Sciences (2007), `http://mlearn.ics.uci.edu/MLRepository.html`
13. Yao, X.: Evolving artificial neural networks. Proceedings of the IEEE 87(9), 1423–1447 (1999)