A Neuro-Genetic Approach to Neural Network Design

Antonia Azzini, Massimo Lazzaroni, and Andrea G.B. Tettamanzi

Universitá degli Studi di Milano
Dipartimento di Tecnologie dell'Informazione
via Bramante, 65
26013, Crema, Italy
azzini,lazzaroni,tettamanzi@dti.unimi.it

Abstract. This paper presents an approach to the joint optimization of neural network structure and weights which can take advantage of BP as a specialized decoder. The approach is validated on the toy problem of N-Input *Parity Function* and successfully applied to a real-world engine fault diagnosis problem.

1 Introduction

Neuro-genetic systems have become a very important topic of study in recent years. Like indicated by Yao et al. in [14], they are biologycally-inspired computational models that use evolutionary algorithms (EAs) in conjunction with neural networks (NNs) to solve problems. The evolutionary approach is a more integrated way of designing ANNs since it allows all aspects of NN design to be taken into account at once and does not require expert knowledge of the problem.

Much research has been undertaken on the combination of EAs and NNs. Through the use of EAs, the problem of designing a NN is regarded as an optimization problem.

EAs can be applied to design ANNs in several ways. Some schemes concentrate just on weight optimization: these can be regarded as alternative training algorithms; others have implemented a search over the topology space, or a search for the optimal learning parameters.

The evolution of weights assumes that the architecture of the network must be static. The primary motivation for using evolutionary techniques to establish the weighting values rather than traditional gradient descent techniques such as BP [8], lies in the trapping in local minima and in the non-differentiability of the function. For this reason, rather than adapting weights based on local improvement only, EAs evolve weights based on the whole network fitness. Several works in this direction have been done by Montana and Davis [7] and by Whitley and colleagues [11]; in [12], they also implemented a purely evolutionary approach using binary codings of weights. In other cases an EA and a gradient descent algorithm have been combined [4].

The design of an optimal NN architecture can be formulated as a search problem in the architecture space, where each point represents an architecture. As pointed out by Yao [14, 15, 13], given some performance (optimality) criteria, e.g., minimun error, fastest learning, lower complexity, etc., about architectures, the performance level of all these forms a surface in the design space. Determining the optimal architecture design is equivalent to finding the highest point on this surface. There are several arguments which make the case for using EAs for searching for the best network topology [6, 10].

However, an interesting area of evolutionary NNs is the simultaneous evolution of different aspects of a NN. One of the most important is the combination of architecture and weight evolution. The advantage of combining these two basic elements of a NN is that a completely functioning network can be evolved without any intervention by an expert.

2 The Neuro-Genetic Approach

We outline an approach to the design of NNs based on EAs, whose aim is both to find an optimal network architecture and to train the network on a given data set. The approach is designed to be able to take advantage of BP if that is possible and beneficial; however, it can also do without it. The basic idea is to exploit the ability of the EA to find a solution close enough to the global optimum, together with the ability of the BP algorithm to finely tune a solution and reach the nearest local minimum.

This research was primed by an industrial application in which it was required to design neural engine controllers to be implemented in hardware, with particular attention to reduced power consumption and silicon area. The validity of resulting approach, however, is by no means limited to hardware implementations of NNs.

We restrict our attention to a specific subset of the NN architectures, namely the MLP. MLPs are feedforward NNs with a layer of input neurons, a layer of one or more output neurons and zero or more "hidden" (i.e., internal) layers of neurons in between; neurons in a layer can take inputs from the previous layer only.

A peculiar aspect of our approach is that we do not use BP as some genetic operator, as it is the case in some related work [1]. Instead, the EA optimizes both the topology and the weights of the networks; BP is optionally used to decode a *genotype* into a *phenotype* NN. Accordingly, it is the genotype which undergoes the genetic operators and which reproduces itself, whereas the phenotype is used *only* for calculating the genotype's fitness.

2.1 The Evolutionary Algorithm

A key objective of our work is the design and the evolution of a suitable ANN architecture. Some problem-specific parameters of the algorithm are the cost α of a neuron and β of a synapsis used to establish a parsimony criterion for

the network architecture; a *bp* parameter, which enables the use of BP if set to 1, and other parameters like probability values used to define topology, weight distribution and *ad hoc* genetic operators.

In the overall evolutionary process the population will be initialized with values based on trial runs on the toy problem. If a new population is to be generated, the corresponding networks will be initialized with different hidden layer sizes, using two exponential distributions to determine the number of hidden layers and neurons for each individual, and a normal distribution to determine the weights. In both cases, unlike other approaches like [15], the maximum size and number of the hidden layers is not pre-determined.

2.2 Representation

Each individual is encoded in a structure in which we maintain basic information as illustrated in Table 1. The values of all these parameters are affected by

Table 1. Individual Encoding.

Element	Description
l	Lenght of the topology string, corrisponding to the number of layers.
topology	String of integer values that represent the number of neurons in each layer.
$\mathbf{W}^{(0)}$	Weights matrix of the input layer neurons of the network.
$\mathbf{W}^{(i)}$	Weights matrix for the ith layer, $i = 1, \ldots, l$.
b_{ij}	Bias of the jth neuron in the ith layer.

the genetic operators during evolution, in order to perform incremental (adding hidden neurons or hidden layers) and decremental (pruning hidden neurons or hidden layers) learning.

Note that the use of the *bp* parameter defines two different types of genetic encoding: if no BP-based network training is employed, we have a *direct encoding*, in which the network structure is directly translated into the corresponding phenotype; otherwise, we have an *indirect encoding* of networks, where the phenotype is obtained by the training of an initial (embryonic) network using BP.

2.3 Fitness

The fitness of an individual depends both on its accuracy (i.e., its mse) and on its cost. Although it is customary in EAs to assume that better individuals have higher fitness, we adopt the convention that a lower fitness means a better NN. This maps directly to the objective function of our problem, which is a cost minimization problem.

Therefore, the fitness is proportional to the value of the mse and to the cost of the considered network. It is defined as

$$f = \lambda kc + (1 - \lambda)\text{mse}, \tag{1}$$

where $\lambda \in [0, 1]$ is a parameter which specifies the desired trade-off between network cost and accuracy, k is a constant for scaling the cost and the mse of the network to a comparable scale, and c is the overall cost of the considered network, defined as

$$c = \alpha N_{hn} + \beta N_{syn}, \tag{2}$$

where N_{hn} is the number of hidden neurons, and N_{syn} is the number of synapses.

The mse depends on the *Activation Function*, that calculates all the output values for each single layer of the neural network. In this work we use the *Sigmoid Transfer Function*. The rationale behind introducing a cost term in the objective function is that we seek for networks which use a reasonable amount of resources (neurons and synapses), which makes sense in particular when a hardware implementation is envisaged.

To be more precise, two fitness values are actually calculated for each individual: the fitness f, used by the selection operator, and a test fitness $\hat{f}$. Following the commonly accepted practice of machine learning, the problem data are partitioned into three sets:

- *training set*, used to train the network;
- *validation set*, used to decide when to stop the training and avoid overfitting;
- *test set*, used to test the generalization capabilities of a network.

Now, $\hat{f}$ is calculated according to Equation 1 by using the mse over the test set. When BP is used, i.e., if $bp = 1$, $f = \hat{f}$; otherwise ($bp = 0$), f is calculated according to Equation 1 by using the mse over the training and validation sets together.

2.4 Genetic Operators

The genetic core of the algorithm is described by the following pseudo-code:

1. Select from the population (of size n) $\lfloor n/2 \rfloor$ individuals by truncation and create a new population of size n with copies of the selected individuals.
2. For all individuals in the population:
   (a) Perform crossover.
   (b) Mutate the topology and the weights of the offspring.
   (c) Train the resulting network using the training and validation sets if $bp = 1$.
   (d) Calculate f and $\hat{f}$.
   (e) Save the individual with lowest $\hat{f}$ as the best-so-far individual if the $\hat{f}$ of the previously saved best-so-far individual is higher (worse).
3. Save statistics.

The *selection* strategy used by the algorithm truncation: starting from a population of n individuals, the worst $\lfloor n/2 \rfloor$ (with respect to f) are eliminated. The remaining individuals are duplicated in order to replace those eliminated. Finally, the population is randomly permuted.

The *crossover* operator is a kind of single-point crossover, where the cutting points are extracted independently for either parent, since the genotype length is variable. Furthermore, the genotypes can be cut only in "meaningful" places, i.e., only bewteen one layer and the next: this means that a new weight matrix has to be created to connect the two layers around a crossover point in the offspring. These new weight matrices are initialized from a normal distribution.

Two types of *mutation* operators are used: a general random perturbation of weights, applied before the BP learning rule, and three *mutation* operators which affect the network architecture. The weight mutation is applied first, followed by the topology mutations, as follows:

1. Weight mutation: all the weight matrices $\mathbf{W}^{(i)}$, $i = 0, \ldots, l$ and the biases are perturbed by adding noise from a zero-mean normal distribution with standard deviation σ, a parameter of the algorithm. After this perturbation has been applied, neurons whose contribution to the network output is negligible are eliminated. This task is carried out according to the following pseudo-code:

   for $i = 1$ **to** $l - 1$ **do**
   if $N_i > 1$
   for $j = 1$ **to** N_i **do**
   if $||W_j^{(i)}|| < \epsilon$
   delete the jth neuron

   where N_i is the number of neurons in the ith layer, and $W_j^{(i)}$ is the jth column of matrix $\mathbf{W}^{(i)}$.
2. Topology mutations: these operators affect the network structure (i.e., the number of neurons in each layer and the number of hidden layers). In particular, three mutations can occur:
   (a) Insertion of one hidden layer: with probability p^+_{layer}, a hidden layer i is randomly selected and a new hidden layer $i - 1$ with the same number of neurons is inserted before it, with $\mathbf{W}^{(i-1)} = \mathbf{I}(N_i)$ and $b_{i-1,j} = b_{ij}$, with $j = 1, \ldots, N_i = N_{i-1}$, where $\mathbf{I}(N_i)$ is the $N_i \times N_i$ identity matrix.
   (b) Deletion of one hidden layer: with probability p^-_{layer}, a hidden layer i is randomly selected; if the network has at least two layers and layer i has exactly one neuron, layer i is removed and the connections between the $(i-1)$th layer and the $(i+1)$th layer (to become the ith layer) are rewired as follows:

   $$\mathbf{W}'^{(i-1)} \leftarrow \mathbf{W}'^{(i-1)} \cdot \mathbf{W}'^{(i)}.$$

   Since $\mathbf{W}^{(i-1)}$ is a row vector and $\mathbf{W}^{(i)}$ is a column vector, the result of the product of their transposes is a $N_{i+1} \times N_{i-1}$ matrix.

(c) Insertion of a neuron: with probability p^-_{layer}, the jth neuron in the hidden layer i is randomly selected for duplication. A copy of it is inserted into the same layer i as the $(N_i + 1)$th neuron; the weight matrices are then updated as follows:
   i. a new row is appended to $\mathbf{W}^{(i-1)}$, which is a copy of its jth row;
   ii. a new column $W^{(i)}_{N_i+1}$ is appended to $\mathbf{W}^{(i)}$, where

$$W^{(i)}_j \leftarrow \frac{1}{2} W^{(i)}_j,$$
$$W^{(i)}_{N_i+1} \leftarrow W^{(i)}_j.$$

   The rationale for halving the output weights from both the jth neuron and its copy is that, by doing so, the overall network behavior remains unchanged, i.e., this kind of mutation is neutral.

All three topology mutation operators are designed so as to minimize their impact on the behavior of the network; in other words, they are designed to be as little disruptive (and as much neutral) as possible.

Table 2 lists all the parameters of the algorithm, and specifies the default values that they assume in this work.

Table 2. Parameters of the Algorithm.

Symbol	Meaning	Default Value
n	Population size	50
seed	Previously saved population	none
T	Maximum time for simulation	∞
G	Maximum number of generations	∞
bp	Backpropagation selection	1
p^+_{layer}	Probability to insert a hidden layer	0.1
p^-_{layer}	Probability to delete a hidden layer	0.1
p^+_{neuron}	Probability to insert a neuron in a hidden layer	0.1
p_{cross}	Probability to crossover	0.6
ϵ	Threshold for use in weight mutation to decide neuron elimination	1
σ	Standard deviation for the normal distribution	1
h	Mean for the exponential distribution	2
N_{in}	Number of network inputs	*
N_{out}	Number of network outputs	*
α	Cost of single neuron	2
β	Cost of single synapsis	4
λ	Desired tradeoff between network cost and accuracy	0.5
k	Constant for scaling cost and mse in the same range	9.0613×10^{-6}

*) depends on the problem.

3 Experiments and Results

The algorithm described in Section 2.1 has been validated by applying it to a real-world diagnosis application. Every industrial application requires a suitable monitoring system for its processes in order to identify any decrease in efficiency and any loss. A generic information from an electric power measurement system, which monitors the power consumption of an electric component, can be usefully exploited for sensorless monitoring of an AC motor drive.

Having in mind the recent trend toward more and more integrated systems, where the drive can be considered as a "black-box", we assume that the only accessible points of the system are the AC input terminals.

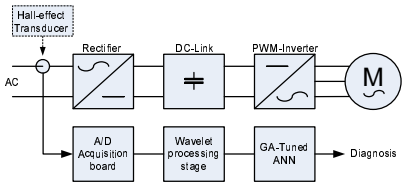

Fig. 1. The experimental setup used for the fault diagnosis application.

The experimental setup is realized as shown in Figure 1, where a three-phase PWM inverter with switching frequency of 4 kHz was used.

The DC-link between the Rectifier and the PWM-Inverter performs a filtering action respect the AC input, theoretically eliminating most of the information about the output circuits of the drive and the motor. Instead, it was proved that the operating condition of the AC motor will appear on the AC side as a transient phenomenon or a sudden variation in the load power. The presence of this electrical transient in the current suggests an approach based on time-frequency or, better, time-scale analysis. In particular, the use of Discrete Wavelet Transform (DWT) [9] could be efficiently used in the process of separating the information.

The proposed approach involves the analysis of the signal—the load current—through wavelet series decomposition. The decomposition results in a set of coefficients, each carrying local time-frequency information. An orthogonal basis function is chosen, thus avoiding redundancy of information and allowing for easy computation.

The computation of the wavelet series coefficients can be efficiently performed with the Mallat algorithm [5]. The coefficients are computed by processing the samples of the signal with a filter bank – as reported in the following figure – where the coefficients of the filters are peculiar to the family of the chosen wavelets. The following figure shows the bandpass filtering, which is implemented as a lowpass $g_i(n)$ - highpass $h_i(n)$ filter pair which has mirrored characteristics.

In particular, in this application the 6-coefficient Daubechies wavelet [3] was used. In Table 3 the filter coefficients of the utilized wavelet are reported.

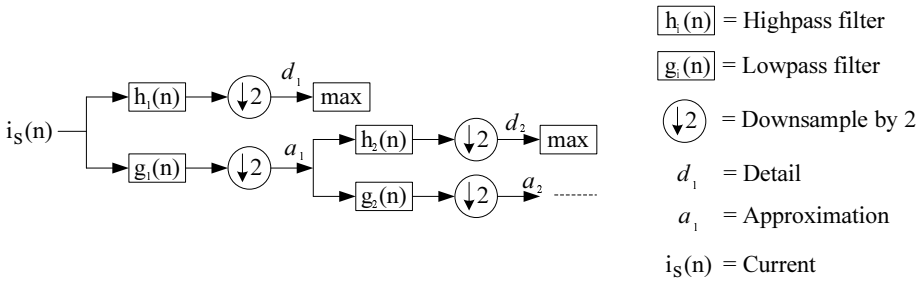

Fig. 2. Bandpass filtering.

Table 3. Filter coefficients of the 6-coefficient Daubechies wavelet.

Filter Coefficients	Low-pass filter decomposition	High-pass filter decomposition
1	0.035226	-0.33267
2	-0.085441	0.80689
3	-0.13501	-0.45988
4	0.45988	-0.13501
5	0.80689	0.085441
6	0.33267	0.035226

The wavelet coefficients allow a compact representation of the signal and the features of the normal operating or faulty conditions are condensed in the wavelet coefficients. Conversely, the features of given operating modes can be recognized in the wavelet coefficients of the signal and the operating mode can be identified.

Employing the wavelet analysis both the identification of the drive operating conditions (faulty or normal operation) and the identification of significant parameters for the specific condition have been obtained.

This problem has been already approached with a neuro-fuzzy network, whose structure was defined *a priori*, trained with BP [2].

By using our approach, both the network structure (or topology) and the weights have to be determined through evolution at the same time. We have to look for networks with 8 inputs and 1 output, which we shall interpret as an estimate of the fault probability.

A few exploratory runs were performed to check whether there was a clear superiority of the version of the algorithm which uses BP w.r.t. the version without BP. The advantage of using BP is not so clear-cut for this problem; however, the version with BP consistently outperformed the version without BP. Therefore, we set $bp = 1$.

As in the case of 5-input parity, we set all the parameters to the default values shown in Table 2 and carried out some experimental runs to find out optimal settings of the parameters p_{cross}, $p^{\pm}_{\text{layer}}$, and p^{+}_{neuron}.

A summary of the results is presented in Table 4: here only data about the best solutions found for each parameter setting over 10 runs are reported. In all cases, the relative standard deviation was below 10%, substantially higher than in the toy problem, but still sufficiently small to guarantee finding a good solution in a few runs. The table also reports the mse and cost for each network: except for one, all settings appear to agree that a cost of 38 is optimal; the lowest mse, however, is obtained when all the parameters are set to their default value.

Table 4. Engine Fault Diagnosis Experimental Results.

Parameter setting			best solution		
p_{cross}	$p^{\pm}_{\text{layer}}$	p^{+}_{neuron}	fitness	mse	cost
0.8	0.1	0.1	0.0654	0.1302	38
0.8	0.05	0.05	0.0636	0.1265	38
0.6	0.1	0.1	0.0528	0.1052	38
0.3	0.1	0.1	0.0670	0.1336	40
0.3	0.05	0.05	0.0644	0.1285	38

A comparison with the results obtained in [2] for a hand-crafted neuro-fuzzy network did not reveal any significant difference. This is an extremely positive outcome, given the expert time and effort spent in hand-crafting the neuro-fuzzy network, as compared to the practically null effort required to set up our experiments. On the other hand, the amount of required computing resources was substantially greater with our approach.

4 Conclusions

We illustrated an evolutionary approach to the joint design of neural network structure and weights which can take advantage of BP as a specialized decoder. The approach has been validated on a toy problem, namely the 5-input parity problem, and then it has been successfully applied to a real-world electric engine fault diagnosis problem.

The experiments showed that the algorithm is somewhat robust w.r.t. the setting of its parameters, i.e., its performance is little sensitive of the fine tuning of the parameters.

The results obtained on the real-world problem compared well against alternative approaches based on the conventional training of a predefined neuro-fuzzy network with BP.

References

1. P. A. Castillo, J. Carpio, J. J. Merelo, A. Prieto, V. Rivas, and G. Romero. Evolving multilayer perceptrons. *Neural Processing Letters*, 12(2):115–127, 2000.

2. L. Cristaldi, M. Lazzaroni, A. Monti, and F. Ponci. A neurofuzzy application for ac motor drives monitoring system. *IEEE Transactions on Instrumentation and Measurement*, 53(4):1020–1027, August 2004.
3. I. Daubeschies. *Ten Lectures on Wavelet*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1992.
4. R. Keesing and D.G. Stork. Evolution and learning in neural networks: the number and distribution of learning trials affect the rate of evolution. *Advanced in Neural Information Processing Systems*, 3:805–810, 1991.
5. S. Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, San Diego, CA, 1999.
6. G.F. Miller, P.M. Todd, and S.U. Hegde. Designing neural networks using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384. J.D. Schaffer, 1989.
7. D. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Conference on Artificial Intelligence*, pages 762–767. Morgan Kaufmann, 1989.
8. D. E. Rumelhart, J. L. McClelland, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.
9. C. Sidney Burrus, Ramesh A. Gopinath, and Haitao Guo. *Introduction to Wavelets and Wavelet Transorms - A Primer*. Prentice Hall, Upper Saddle River, New Jersey 07458, 1998.
10. K. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. Technical report, University of Texas at Austin, Department of Computer Science, Austin, Texas, 2001.
11. D. Whitley and T. Hanson. Optimizing neural networks using faster, more accurate genetic search. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 391–396. J. D. Schaffer, 1989.
12. D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel computing*, 14:347–361, 1993.
13. X. Yao. Evolving artificial neural networks. In *Proceedings on IEEE*, pages 1423–1447, 1999.
14. X. Yao and Y.Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, 8(3):694–713, May 1997.
15. X. Yao and Y.Liu. Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91(1):83–90, 1998.