Transactions on Information and Communications Technologies vol 15, © 1997 WIT Press, www.witpress.com, ISSN 1743-3517

Optimizing the structure of neural networks using evolution techniques

S. D. Likothanassis[a,b], E. Georgopoulos[a] & D. Fotakis[a,b]
[a]*Department of Computer Engineering and Informatics, University of Patras, Patras 26 500, Greece*
[b]*Computer Technology Instiute, Patras 26100, Greece*
E-mail: likothan@cti.gr

Abstract

Evolutionary methods have been widely used to determine the structure of artificial neural networks. In this work, we propose a more efficient implementation which face the above problem, by using a neural network model as general as possible. So, we used a fully connected network, consisting of three parts, the input layer, the output layer and a number of hidden layers. This implementation has been proved, via simulations, that it optimazes the size of the network and gives better results compared with other existing methods, while the use of crossover results to more efficient networks. Furthermore, the random initialization has been proved more efficient, since it reduces significantly the number of the generations needed for the convergence of the algorithm. Finally, some aspects for the convergence of the parallel implementations of the algorithm are discussed.

1 Introduction

A continuing question in the research of neural networks is the size of a neural network required to solve a specific problem. If the training starts with a small network, it is possible that no learning is achieved. On the other hand, if a larger than required network is used, then the learning process is very slow. Furthermore, the underlying model, that provides the data for the

This work was supported in part by the Hellenic General Secretariat of R&T, under grand PENED 1462

training set, is usually unknown or variable, resulting to incomplete driving information. In these cases no standard rules exist, on how one can implement a network which will solve a specific problem.

Genetic Algorithms are a class of optimisation procedures which are good at exploring a large and complex space in an intelligent way to find values close to the global optimum. Hence, they are well suited to the problem of training feedforward neural networks. At the end of last decade, there have been attempts to combine the technology of neural networks with that of genetic algorithms. Davis [1] showed how any neural network can be rewritten as a type of genetic algorithm. Whitley [2] attempted unsuccessfully to train feedforward neural networks using genetic algorithms. Montana and Davis [3] presented a new algorithm for training feedforward networks. It not only succeeds in its task but it outperforms backpropagation, the standard training algorithm, in a difficult example. This success comes from tailoring the genetic algorithm to the domain of training neural networks. A standard genetic algorithm works well on small neural net optimization problems, but typically, it fails on problems with larger encoding. Whitley and Hanson [4] have successfully optimized a class of neural networks using a different genetic algorithm that employs one-at-a-time reproduction and allocates reproductive opportunities according to rank to achieve the desired selective pressure. The above referred algorithms face only the problem of the network training, i.e. the optimization of the weights and they do not deal with the problem of the network size.

A general approach to the problem of optimization of the network size is to define a network that is large or larger than necessary to do the job, and then to use a genetic algorithm to define which combination of connections are sufficient to quickly and accurately learn to perform some target task using back propagation. Miller et al. [5] did this for some small nets. The same problem for larger networks is faced by Whitley and Bogard [6].

A more general method is introduced in [7]. The concept leads in a natural way to a model for the explanation of inherited behaviour. Explicitly, it is studied a simplified model for a brain with sensory and motor neurons. It is used a general asymmetric network whose structure is solely determined by an evolutionary process. Specifically, the structure of the neural net is determined by the algorithm and no global learning rule has to be specified for a given problem, except the parameters of the genetic algorithm. The proposed mutation process in this algorithm is as follows. With a certain probability, it is removed a given number n_r of neurons completely of the brain and it is added a given number n_a of neurons with numbers $S_1,\ldots,S_{na}$ of synapses with randomly chosen couplings to the network. In this procedure, the brains steadily grow until they reach an average brain size that is mainly determined

by mutation parameters, like the number of synapses, but also by the speed of performance, since larger nets become too slow and are removed in the selection step. In general the resulting brains are diluted.

A more recent approach, is presented in [8]. In this approach, learning imparts a finer structure on a neural network coarsely lined-out by a genetic algorithm. Genetic algorithms are used to move the search to an appropriate region in the solution space. Learning then executes a more local search to achieve an optimal performance. A promising approach to solving complex problems with learning neural networks is to have a genetic algorithm specify a modular initial architecture. Finally, in the reference [9], a radial basis function (RBF) network configuration, using genetic algorithms is presented. In the referred work, genetic algorithms are proposed to automatically configure RBF networks. The network configuration is formed as a subset selection problem. The task is then to find an optimal subset of n_c terms from the N_t training data samples.

In this work, we propose a more efficient implementation which face the above problems, by using a neural network model as general as possible. So, we used a fully connected network, consisting of three parts, the input layer, the output layer and a number of hidden layers. The method is implemented using linked lists. Furthermore, in the initialisation face , we use a population of randomly generated individuals (neural networks), with random number of hidden layers and randomly generated synaptic weights.

This implementation has been proved, via simulations, that optimises the size of the network and gives better results compared with other existing methods. Furthermore, the random initialisation has been proved more efficient, since it reduces significally the number of the generations needed for the convergence of the algorithm. Finally, the use of crossover gives more efficient networks, with respect to the number of hidden neurons.

The paper is organised as follows. After defining in section 2 the Evolution Programs, section 3 introduce the general model, while the genetic program used to optimise the neural model is presented in section 4. Section 5 depicts the simulation results and in section 6 some aspects for the convergence of the parallel implementation of the algorithm are discussed. Finally the conclusions are presented in section 7, followed by the references.

2. Evolution Programs

The last years there is a increasing interest in systems that make use of the principles of evolution and hereditary. Systems like that maintain a population of potential solutions and they have some selection process based on the fitness of individuals and some "genetic" operators. Such very well known, problem solving systems are, for example, the Evolution Strategies, Fogel's Evolution Programming, Holland's Genetic Algorithms and Koza's Genetic

Programming.

We use the term **Evolution Programs (EP)** to describe all the evolution based systems [10]. An evolution program is a probabilistic algorithm that maintains a population of individuals , $P(t)=\{i_1^t, \ldots, i_n^t\}$ for iteration t. Each individual represents a potential solution to the problem at hand, and is implemented as some data structure S. Each solution (individual) i_j^t is evaluated to give a measure of its fitness. Then a new population (iteration t+1) is created by selecting the more fit individuals (select step). Some members of the population undergo transformations (alter step) by means of "genetic" operators to form the new solutions (individuals). There are unary transformations m_i (mutation type), which create new individuals by a small change in a single individual and higher order transformations c_j (crossover type), which create new individuals by combining parts from two or more individuals. After some number of iterations the program converges. At this point is hopped that the best individual represents a near-optimum (reasonable) solution.

There are a lot of evolution programs that can be written for a given problem. Such programs may differ in many ways; they can use different data structures for implementing the individuals, "genetic" operators for transforming the individuals, methods for creating an initial population, methods for handling constraints of the problem and parameters (population size, probabilities of the "genetic" operators, e.t.c.). However they share a common principle; a population of individuals undergoes some transformations, and during this evolution process the individuals strive for survival.

The idea of evolution programming is not a new one but the idea of evolution programs is different for the previously proposed ones and it was introduced for the first time by Zbigniew Michalewicz[10]. It is based entirely on the idea of genetic algorithms. The difference is that in evolution programs we consider a richer set of data structures (for chromosome) representation together with an expanded set of genetic operators, whereas the classical genetic algorithms use fixed-length binary strings (as a chromosome) for its individuals and two operators : binary crossover and binary mutation.

Until now, with the classical genetic algorithms in order to solve a given problem we have to transform it into a form more appropriate for the genetic algorithm; this includes mapping between potential solutions and binary representation, decoders, e.t.c. The approach of evolution programs is almost the opposite of the former one; we have to transform the genetic algorithm to suit the problem.

3. The model for a Genetically Optimized Neural Network

In this section we will present the model of the Artificial Neural Network that

Transactions on Information and Communications Technologies vol 15, © 1997 WIT Press, www.witpress.com, ISSN 1743-3517

will be processed by the Evolution Program. This model was chosen to be as general as possible. So, we used a Multilayer Perceptron network. A simple network that fulfils the structural requirements of that class of a set of sensory neurons that constitute the input layer, one or more hidden layers of computation neurons (hidden neurons), and an output layer of computation neurons (output neurons).

In the model of neural network that we have implemented the following assumptions were made :

- Each neuron (computation neuron) in the network is represented by the McCulloch-Pitts model.
- Bits 0 and 1 are represented by the levels 0 and +1, respectively.

The neural network model is full connected with feedforward connections. Furthermore, we introduce direct couplings of feedforward type between input and output neurons. Figure 1 depicts a portion of our neural network model.

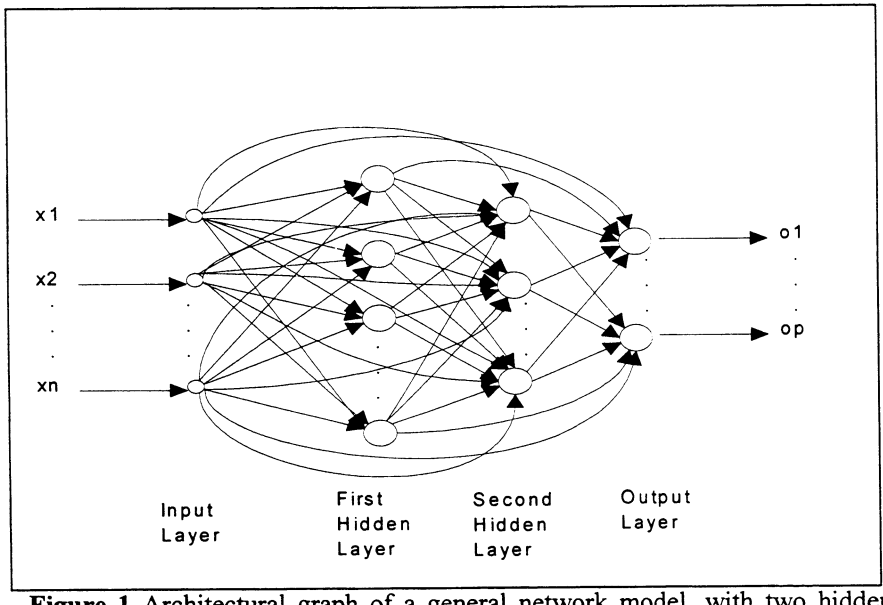

Figure 1 Architectural graph of a general network model, with two hidden layers.

The objective of our approach was to implement a neural network, capable of performing its optimal structure dynamically. Thus, we have used, dynamically data structures, such as linked lists. The whole neural network is constructed as a linked list of layers. While every layer is organised as a linked list of neurons and every neuron as a linked list of synapses (connections), where every synapse is specified by the neuron to which it is connected and by its strength(weight). Hence we are able to add and delete layers from the

neural network, add and delete neurons from the net or a layer, add and delete connections, and so on.

4. The Evolution Program

In this section we will describe the implementation of the Evolution Program which was used for the simulations. First in order to describe the algorithm's implementation, it is useful a sort presentation of the algorithm's steps:

1. Initialisation : *Create an initial population of individuals* (neural networks). Each individual is created randomly. In every neural network we create that way, only the number of inputs and outputs are constant. The number of hidden layers as well as the number of neurons in every hidden layer are chosen randomly. In the same way we generate randomly the synaptic weights in the interval [-1,1]. So, at the end of the initialisation procedure we have a population of randomly generated individuals (neural networks).

2. Calculate every individual's fitness : *Present the teaching patterns, calculate the output of each output neuron of the net and compare them with the target outputs*. Thus the number of right bits that have been learned, for each training pattern, is evaluated. So if the function that we want to teach the net has n input and r output bits, we then have to test 2^n possibilities, resulting $2^n r$ bits that can be right or wrong. Then evaluate the performance of each individual (neural net) using a very simple fitness function which equals to the number of correct bits of the Boolean function. Finally, calculate the fitness functions of every individual

3. Select and create a new population : *Using as a measure the fitness functions of the individuals, that were calculated in the previous step, select these individuals that are going to be copied to the new population.* This can be done with several ways. One way is to find the best individuals in the population (those with the biggest fitness functions), remove a certain number r_d of the worst individuals from the population, and copy, to their place, the best one r_1 times, the second best r_2 times, and so on, so that always $r_d=r_1 + r_2 + \ldots$(we suppose that the size of the population is constant). Another way is by using the classic or an elitist roulette wheel selection. We have implement all these three selection processes.

4. Crossover individuals : *With a certain probability select an even number of individuals from the population. In the sequence, mate these individuals randomly* : for each one in a pair of individuals, we generate a random natural number in the range [1,..., number of hidden layers -1], if the number of hidden layers of the individual is greater than 1, otherwise the "random" number is 1. This number indicates the position of the crossing point; this position is the number of a hidden layer. Since the neural networks

(individuals) in the population may not have the same number of hidden layers, this crossing point is generated independently in each individual of the selected pair. Suppose, that we have two individuals : I H_1 H_2 H_3 O and I H_1' H_2' H_3' H_4' O', where I represents the input layer O the output layer and H_i the ith hidden layer. If the crossing point for the first individual is 2 and for the second one is 3, then the "child" of these two individuals would look like this: I H_1 H_2 H_4' O'. The connections between the layers from the different networks take weights that are generated randomly in the range [-1,1]. That way for each pair of individuals we generate two offspring which replace their "parents" in the population.

5. **Mutate individuals :** With a certain probability we choose individuals from the population, remove a given number of neurons from their hidden layers and add a given number of neurons. The weights of the new connections are distributed randomly in the interval [-1,1]. This kind of mutation is not the best one but even this crude concept of mutation is suited for improving a neural net with the use of Evolution Programs and is important that there is no need to specify some complicated structure information or a global learning rule.

Steps 2,3 and 4 are repeated until we get a solution of the problem or until we reach a exit loop condition (for example a maximum number of cycles).

5. Simulations

In this section we will present examples of the algorithm's performance. A very common problem that a neural network should be able to solve is the XOR problem. Table 1 presents some examples of teaching the simple XOR function to the evolutionary trained net. Table 2 presents the average number of generations, that the program needs to solve the XOR problem, and the average number of neurons in the hidden layers of the best network. In both tables 1 and 2 we execute the experiments without using the *crossover* operator. In order to create table 2 we run the Evolution Program a sufficient number of times.

Number of Generations	Number of Hidden Neurons
15	2
0	2
8	11
102	6
123	3

Table 1 The first five runs with population size equal to ten networks.

Population Size	Average Number of Generations	Average Number of Hidden Neurons
10	52.45	7.38
20	30.38	7.30

Table 2 The average number for the generations and the hidden neurons in the best net for 40 runs, (probability of mutation = 0.5).

In table 3 we present some examples of teaching a simple XOR function to a genetically trained neural network, from the approach reported in the work of Bornholdt & Graudenz[7]. In this work a diluted biological neural network is used. The superiority of the results depicted by the general model, compared to that of the diluted model, verify the input-output connections are improve the learning. Finally an example of a neural network representation that results using the general model and evolution techniques, for the XOR problem is depicted in figure 2

Population Size	Number of Generations	Number of Hidden Neurons
10	1200	22
10	2400	7
10	5000	8
10	5500	13
10	6000	12

Table 3 The number of generations and hidden neurons (reported in [7]).

For the above results, it is feasible to teach the XOR function to our evolutionary optimized neural network without the use of the crossover operator. However the crossover is a very important operator since, it combines information from two or more individuals to their offspring, in the same way as the reproduction does in nature. So we implement a crossover operator for our program and execute some additional experiments for the XOR problem. Table 4 illustrates some indicative results for probability of crossover =0.5 and probability of mutation = 0.4.

Population Size	Average Number of Generations	Average Number of Hidden Neurons
20	29.9	4.7

Table 4 Average Number of generations and hidden neurons using crossover.

Comparing tables 2 and 4 we can see that the use of Crossover operator results in Neural Networks with significantly smaller sizes than the previous case,

without increasing the average number of generations needed by the E.P. to find the solution.

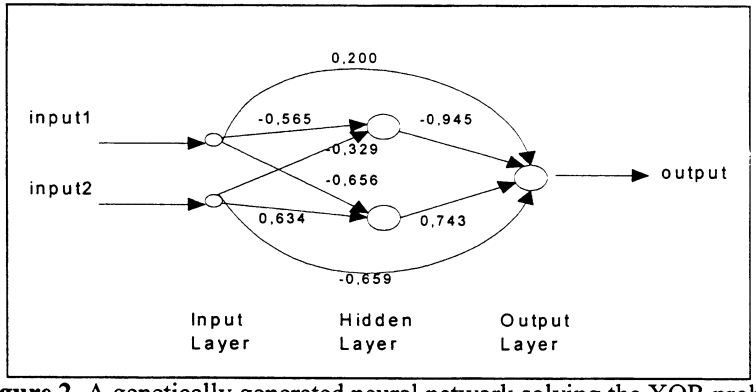

Figure 2 A genetically generated neural network solving the XOR problem.

6. Efficient parallel simulations of the system

In this section, we study some aspects concerning the efficient parallel simulation of a certain subclass of Genetic Systems. We exploit some recent results on Quadratic Dynamical Systems so as to prove that non-degenerate crossover systems can be simulated in the class *NC*. Thus, we provide sufficient conditions for a crossover system to be efficiently simulated in parallel.

Quadratic Dynamical Systems (QDSs), whose definition extends that of Markov chains, have recently proposed as model for Genetic Algorithms [11]. Like a Markov chain (a Linear System), a QDS maps a probability distribution p on a state space N into another probability distribution p' on N. The map is quadratic in the sense that two random samples are picked independently from p to produce a random sample from p' according to some mating rule β.

It is well known that irreducible and aperiodic Markov chains converge to a unique limit (stationary) distribution and there is a nice spectral theory explaining how the rate of convergence relates to the second largest eigenvalue of the transition matrix. A Markov chain is called *rapidly mixing*, if it converges to the stationary distribution to polylogarithmic (to the number of states) number of steps. This definition can be generalized to the case of Q.D.S. Moreover, a QDS is called *very rapidly mixing* (or *NC rapidly mixing*), if it converges to a stationary distribution to poly(loglog|N|) number of steps. Recently, Diaz, Serna and Spirakis [12] proved that a crossover system for almost uniformly sampling matchings from a given graph is very rapidly mixing. Also, they gave an *NC* simulation of this crossover system, showing that a restricted population model of polynomial size is enough to carry the simulation.

It is known that some Markov chains mix rapidly. This can be [11] generalized to crossover (non-degenerate) Genetic Systems basically, since the "history" of matings, because of which an individual was finally generated (at "steady state"), is a "tree" (a family tree indeed). The depth of this tree is statistically only logarithmic in n (the number of alleles) because each mating is a *random partition* of the inherited (active) alleles to parents (if thought in the reverse).

Let us consider a non-degenerate crossover distribution such that $\frac{1}{\ln r(\Pi)^{-1}} = O(poly(\log n))$ (this is true for some widely applied crossover distributions such as the uniform crossover and the Poisson crossover. Thus, the corresponding non-degenerate crossover system converges to a well-defined stationary distribution in polylogarithmic (to the size of the representation of individuals) number of steps. Since $N=2^n$ any non-degenerate crossover system of this kind is very rapidly mixing because it converges in poly(log n) = poly (loglog|N|) number of steps. Also any crossover system of this kind can be simulated using a restricted population model of polynomial size.

Theorem 1 [11]. *For an arbitrary crossover distribution the quadratic system p_t can be simulated by a similar system f_t based on finite population of size m such that* $\|p_t - f_t\| \le \frac{8n^2t}{m}$.

The previous theorem implies that for any $\delta > 0$, the finite population system and the quadratic system remain within variation distance δ for at least t steps provided only that the population size is at least $m = \left\lceil \frac{8n^2t}{\delta} \right\rceil$.

Thus for any non-degenerate crossover system that converges to the stationary distribution in poly(log$n$) number of steps, a finite population of polynomial size is enough for the simulation of the system. Additionally , if the corresponding crossover distribution can be simulated in *NC*, then the crossover system can easily be simulated in NC by assigning a group of processors to each individual [12]. The following theorem concludes our study:

Theorem 2. Let Π be a non-degenerate crossover distribution such that :

(a) $\frac{1}{\ln r(\Pi)^{-1}} = O(poly(\log n))$, and

(b) can be simulated in *NC* in time t_c and using p_c processors.

Then the corresponding crossover system, can also be simulated with an error ε in *NC* in *time* $t = O(\frac{1}{\ln r(\Pi)^{-1}}(2\ln n + \ln \varepsilon^{-1})t_c)$ and using $O(\frac{n^2t}{\varepsilon}p_c)$ number of processors, where r(Π) = max $r_{ij}(\Pi)$ and $r_{ij}(\Pi)$ = Prob[S does not separate i,j].

7. Conclusions

In this work it has been shown that it is feasible to train a neural network using Evolution Programs without specifying any global learning rule. The structure of the hidden region of the network doesn't need to be specified in detail since this job is left to the Evolution Program.

From the numerical simulations it is proved that using the Evolution Program a network learns much faster a given Boolean function, than the procedure that is presented in [7].

The use of random initialization has been proved that reduces significantly the number of generations needed for the convergence of the algorithm. Furthermore, the use of crossover, results in significantly smaller net sizes, for the same average number of generations. Finally, the crossover system, can be efficiently simulated in parallel, with an error ε in NC, if some sufficient conditions are satisfied.

The conclusion remark is that evolutionary generated neural networks are not tied to any global learning rule and they can be used to a variety of computational problems. Therefore these kind of networks may well be a promising concept for the future of neural networks.

References

1. Davis, L., 'Mapping classifier systems into neural networks', *Proceedings of the 1988 Conference on Neural Information Processing Systems*, Morgan Kaufmann, (1988).

2. Whitley, D., 'Applying genetic algorithms to neural network problems', *International Neural Networks Society*, p.230, (1988).

3. Montana, D. and Davis, L. Training feedforward neural networks using genetic algorithms, *BBN Systems and Technologies*, Cambridge, MA, (1989).

4. Whitley, D., and Hanson, T. Optimizing neural networks using faster, more accurate genetic search, *3rd Intern. Conference on Genetic Algorithms, Washington D.C.*, Morgan Kaufmann, pp. 391-396, (1989).

5. Miller, G., et al. Designing neural networks using genetic algorithms', *Proceedings of the 3rd International Conference on Genetic Algorithms*, Morgan Kaufmann, (1989).

6. Whitley, D., and Bogart, C. The evolution of connectivity: Pruning neural networks using genetic algorithms , *International Joint Conference on Neural Networks, Washington D.C.,* 1. Hillsdale, NJ: Lawpence Erlbaum, pp. 134-137, (1990).

7. Bornoholdt S. and Graudenz, D. General asymmetric neural networks and structure design by genetic algorithms, *Neural Networks*, Vol. 5, pp327 - 334, (1992).

8. Happel, B., et al. Design and evolution of modular neural network architectures, *Neural Networks*, Vol. 7, pp. 985 - 1004, (1994).

9. Billings, S.A., and Zheng, G.L. Radial basis function nerwork configuration using genetic algorithms, *Neural Networks*, Vol. 8, No. 6, pp. 877-890, (1995).

10. Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs, Second Extended Edition.* Springer-Verlag, 1994.

11. Rabani, Y.,Rabinovitch, Y. and Sinclair, A. A computational View of Population Genetics (preliminary version), Proceedings of the 27th Annual ACM symposium on Theory of Computing, pp. 83-92, 1995.

12. Diaz, J., Serna, M. and Spirakis, P. Sampling matchings in Parallel, submitted, November 1996.