Evolutionary Design of Application Tailored Neural Networks

Zoran Obradović and Rangarajan Srikumar

***Abstract*— First, an evolutionary algorithm for designing a single hidden layer feedforward neural networks is proposed. The algorithm constructs a problem tailored neural network by incremental introduction of new hidden units. Each new hidden unit is added to the network by linear partitioning of the hidden-layer representation through a genetic search. Second, a two stage algorithm speed-up is achieved through: (1) distributed genetic search for hidden layer units construction along with the appropriate input to hidden layer weights; and (2) the dynamic pocket algorithm for learning the hidden to output layer weights. Finally, promising experimental results are presented on fast construction of small networks having good generalization property.**

I. Introduction

Research on using genetic algorithms for neural networks learning is increasing. Typically, genetic search is used for the weights optimization on a prespecified neural network topology (for survey, see [8]). However, determining the appropriate size of a neural network is one of the most difficult tasks in its construction. An attempt to overcome the fixed architecture problem are constructive learning algorithms that grow or shrink the network in an application specific manner [3, 4]. An interesting system can be obtained by combining an existing constructive learning algorithm and the weights optimization of neural networks using genetic algorithms [5, 7]. However, the space defined by a combination of two techniques might be difficult for the genetic search. Another previously suggested approach to neural networks design using genetic algorithms is architecture optimization only[6]. In those applications once the networks are constructed, slow backpropogation algorithm is used for learning appropriate interconnection weights.

In the evolutionary algorithm that we propose here there is no need for additional adaptation of connections from input to the hidden layer after an architecture is designed. The algorithm designs a neural network growing one hidden unit at a time, each constructed using genetic search on a relatively simple space. Each hidden unit is immediately assigned appropriate connection weights from the input layer. The pocket algorithm[3] is then used to learn the connection weights between the hidden and the output layer. Experiments indicate that the algorithm generates small networks with good generalization ability. The drawback of this algorithm is that it is highly computation intensive when implemented on a sequential machine, which makes it inappropriate for large scale problems. The proposed algorithm is speeded-up by parallelization of the genetic search and the pocket algorithm to the level that it is applicable to large real life problems.

A description of the sequential algorithm is given in Section 2. A parallel version of the algorithm is proposed in Section 3. The theoretical analysis and experimental results are discussed in Section 4.

II. The Evolutionary Algorithm

For a given domain $D \subset R^m$ we define a *region* by its bounding set of hyperplanes in R^m. A region is said to be *resolved* if almost all training examples (high percentage) belonging to that region are of one class, otherwise the region is *unresolved*.

Let us assume that the given problem can be represented by a feedforward neural network of a single hidden layer with units computing threshold functions of their weighted input sum (usually called hard limiter units). Each hidden unit in such neural network can be interpreted as a hyperplane through the problem domain $\Re^m$. Hyperplanes corresponding to the network's hidden units partition the domain into resolved regions. Consequently, learning goal can be interpreted as a construction of a small set of hyperplanes (corresponding to hidden layer units) which partition the training set into resolved

This work was supported in part by the National Science Foundation under grant IRI-9308523.

Z. Obradović's address is the School of Electrical Engineering and Computer Science, Washington State University, Pullman WA 99164-2752, USA. He is also affiliated with the Mathematical Institute, Belgrade, Yugoslavia.

R. Srikumar's address is Microsoft Corporation, 3219 Building 16, One Microsoft Way, Redmod WA 98052-6399.

regions.

The outline of the proposed constructive learning algorithm follows:

(1) Start from a single unresolved region of all training examples, and with a neural network without hidden units.
(2) Generate a new hidden unit (inter-connection weights and threshold) using genetic search and add it to the current network.
(3) Partition the current unresolved region(s) further with the generated unit. Discard resolved regions.
(4) Go back to step 2 till all regions are resolved.
(5) Learn hidden to output layer inter-connection weights.

Let us form pairs of training examples, each pair consisting of two examples belonging to different classes. Let A and B be such a pair of training examples. For perfect classification of training examples by a single hidden layer neural network, there must be a hyperplane corresponding to a hidden unit separating A from B. Assume that the line connecting A and B is completely covered by k disjoint *building blocks* all of the same length. For a perfect classification there must be a building block between A and B such that the hyperplane separating A from B passes through it. For large k the block is small and consequently the hyperplane can be assumed to pass through the center of it. A hyperplane in $\Re^m$ is uniquely determined by m points. So, the equation of the hyperplane separating A from B is defined by m building blocks appropriately positioned between m pairs of training examples similar to A and B. A genetic search can be used to determine m such appropriately positioned building blocks that define a desired separating hyperplane in $\Re^m$ (for $m = 2$ see an example on Figure 1).

In the genetic algorithm, population represents a set of hyperplanes each being a candidate for the next hidden unit that we want to add to the neural network. Each individual hyperplane is represented by fixed length binary string. For m dimensional input space, each string consists of m concatenated substrings of equal length. Each of those substrings encodes a 4-tuple: a region, pair of a positive and a negative training examples both belonging to that region, and the index (one of k possibilities) of a building block between those two examples that defines a point on the hyperplane.

Here, genetic search consists of a sequence of steps (called *generations*) where in every generation a better population of candidates for the next hidden unit to be added to the network is created. For that matter an incremental static population model is used where the population is ranked according to fitness [10]. The two best ranked strings from previous generation are copied into the next generation. The rest $n-2$ strings of the new generation are the result of crossover and mutation on the n strings of the previous generation. This model ensures that the best strings are not destroyed.

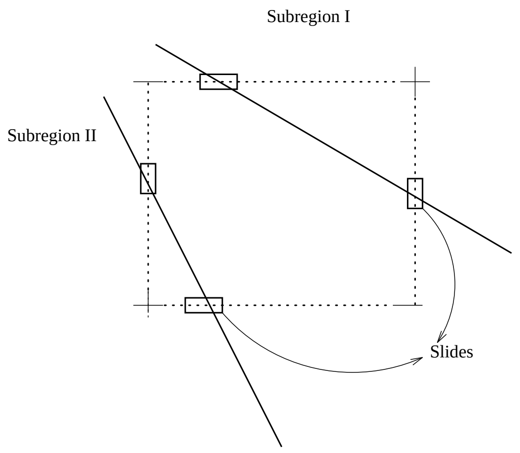

Figure 1: The XOR function realized by the net with two hidden units. The figure shows building blocks on the dotted lines between the examples belonging to the opposite classes (between $(0,0)$ and $(0,1)$; $(0,0)$ and$(1,0)$; $(0,1)$ and $(1,1)$; $(1,0)$ and $(1,1)$). Each block can be in one of the k different positions on the dotted line.

In each generation the hyperplanes in the population are evaluated for their fitness. If τ_i is the percentage of all training examples from class i correctly classified by the hyperplane, then its *fitness* is defined as the sum of τ_i over all classes. The crossover occurs with equal probability between any two adjacent bits.

Initially, the problem domain is a single unresolved region. Genetic search for a pre-specified number of generations is performed and the hidden unit corresponding to the generated hyperplane is added to the existing network hidden layer. This hidden unit is used to partition the unresolved regions further. All resolved regions can be ignored in future constructive steps of adding new hidden units because a set of hidden units that can classify those regions correctly is already designed. The unresolved regions are maintained in a linked list. The process continue construction of new hidden units, each using new genetic search, till all the regions are resolved.

The hidden layer units and the connections from the input to the hidden layer can be easily generated from the constructed hyperplanes. The final step in the algorithm is to learn the connection weights between the hidden and the output layer. This task is performed using the pocket algorithm[3], a modification of the perceptron algorithm able to produce the optimal separation between non-linearly separable classes.

III. Construction Speed-Up

A speed-up of hidden layer construction is possible through distributed genetic search. Additional speed-up is achievable through parallelization of the hidden to the output layer learning.

A. Distributed Genetic Search

In the genetic search algorithm from Section 2 the most expensive step in a genetic cycle is strings fitness evaluation. If a population consists of n strings, in each generation the fitness will have to be evaluated for all n strings. In practice n ranges between 50 to 200. Our experiments show that in the sequential implementation of the algorithm more than 80% of the time is spent computing the fitness.

The estimation of the fitness value of the strings are independent of one another, and this makes it appropriate for distributed computing. Given a network of $n+1$ processors (Main and n Fitness nodes), estimation of the fitness could be performed concurrently, each on a different processor in a distributed environment.

In a distributed algorithm that we propose Main node process initially broadcast the unresolved region (the training set) to all Fitness nodes. Main node process executes the algorithm sequentially till there is a need to evaluate strings based on their fitness. At that point, Main node process distributes n strings (population) each to one of n Fitness nodes that work in parallel, where each of them computes fitness of a string assigned to it. Fitness values are computed using current list of unresolved regions and returned back to the Main node process which then broadcast the fittest hyperplane back to all the Fitness nodes. On receiving the fittest hyperplane each of the Fitness nodes concurrently modifies its current list of unresolved regions, partitioning unresolved regions further and discarding the regions resolved by this new hyperplane. At the same time the Main process continues sequential computation till there is a need to compare fitness values of strings in the next generation. The process terminates when all regions are resolved. If less than $n+1$ processors (workstations) are available then the estimation of fitness is evenly distributed among available processors.

For an efficient distributed algorithm, communication among processors should be minimized since it can be very expensive. Observe that in the proposed parallelization each message is of the same size and it is indeed quite small (one string rather then the whole population). In addition, the genetic search is performed for a pre-specified number of generations (a small constant), and consequently the total cost of message exchanges per each hidden unit construction is a small constant (one communication per generation).

B. Dynamic Pocket Algorithm

The basic idea of the pocket algorithm is to run the perceptron algorithm while keeping a backup hypothesis (weights assignment) "in pocket" [3]. Whenever the perceptron hypothesis has a better performance it replaces the pocket hypothesis. The final pocket hypothesis is the output of the algorithm. The drawback of the algorithm is that the processes of estimating the better of the two hypothesis (perceptron and pocket) is computationally extremely costly. This is especially true when the training set is large.

To speed-up the pocket algorithm we propose replacement of the existing pocket memory with another special perceptron called *Slave.* In the *dynamic pocket algorithm* the perceptron, here called *Master*, and the *Slave* run in parallel on the same input. The Slave is devoid of power to update its current hypothesis, but in contrast to the original pocket algorithm it evaluates the quality of the pocket hypothesis concurrently with the evaluation of the Master's hypothesis.

The *Slave* in addition to its current hypothesis π, keeps φ which is current number of consecutive correct classification by π, and ϕ the maximum φ so far. The *Master* on the other hand has its current hypothesis Π, and keeps Φ which is current number of consecutive classification of the training samples by Π. Both the *Master* and the *Slave* start of with randomized Π and π. The indices ϕ and Φ indicate the respective goodness of π and Π at any particular moment. When Φ becomes greater that ϕ, an estimate of *goodness* of π and Π are made. If Π is found to be better, then previous π is replaced by Π. The training procedure goes on till the training samples are classified correctly or a predetermined number of iterations are completed. The Monitoring subsystem is responsible for estimating the quality of the Master and the Slaves hypothesis (i.e., Π and π). If the Master hypothesis is found to be better, the Monitor subsystem replaces Slaves weights by that of the Master.

Using this dynamic approach the threshold ϕ for evaluation of the current hypothesis Π is subject to dynamic update unlike the original pocket algorithm (i.e., ϕ increases to φ if $\varphi > \phi$). Thus, a significant

Problem	**Tower**		**Evolutionary**	
	U_{gen}	P_{test}	U_{gen}	P_{test}
Monk1	6	81%	3	95%
Monk2	8	82%	8	87%
Monk3	5	83%	6	99%

Table 1: The Tower and the evolutionary algorithm comparison on the Monks problems. Here, U_{gen} is the number of units generated and P_{test} is the performance on the testing set.

computing time is saved by reducing the number of useless quality evaluations of the current master's hypothesis. The experimental results of Section 4 are consistent with this observation.

IV. Results

First, convergence and size of the constructed network and quality of its generalization are studied. Second, time needed for the network design is discussed.

A. Network Generalization and Complexity

The following theorem proves the convergence of the proposed network design algorithm.

Theorem 1: The proposed constructive neural network design algorithm will always converge.

Proof: With the addition of each hidden unit (hyperplane) one of the following will occur: (1) number of training examples belonging to unresolved regions decreases; (2) number of existing unresolved regions increases. Consequently, all regions will be eventually resolved since both the number of training examples belonging to unresolved regions, and the number of unresolved regions are bounded by the number of examples. □

In general the algorithm uses genetic search with an aim to discover the best hidden unit that we then add to the existing network. The constructed network complexity and generalization is experimentally tested on a number of problems. For the XOR problem the evolutionary algorithm constructs the minimum configuration of 2 hidden units shown in Figure 1. All generalization and complexity results presented in this paper are obtained by averaging over ten experiments and rounding to the nearest integer. The experiments are identical in all parameters except for the seed of the random number generator. In all experiments a region is considered to be resolved only if it is 100% pure. The intention was to test if this oversimplification will result in an overfitt. However, in our experiments the obtained networks generalized quite well as demonstrated by the following results.

In the first experiment of the constructed network complexity and generalization, the algorithm designed a network with two hidden units learning from 300 training examples of the votes database of the U.S House of Representatives congressmen on 16 key votes identified by the Congressional Quarterly Almanac [1]. This database of two classes consists of 435 examples each of 17 attributes, and has previously published classification generalization results of about 90 - 95% correct. The generalization performance of the evolutionary designed neural network on the test set of 135 randomly selected previously unseen examples is 97% correct.

The second test was on another standard benchmark, the Monk's problems relying on the artificial domain in which robots are described by six different attributes[9]. It comprises of three different tasks. Each learning task is a binary classification problem given by a logical description of a class. Robots belong either to a particular class or not, but instead of providing a complete class description to the learning problem, only a subset of 432 possible robots with their classification is given. The learning task is then to generalize using these examples. The results of the evolutionary algorithm on these problems are listed in Table 1, where the training set had 125, 170, 350 examples respectively. Evolutionary design algorithm performed better than another popular constructive technique called the Tower algorithm[3].

Finally, tests were conducted on learning two randomly generated networks with three and five hidden units. For both test problems the algorithm learned the minimal networks from a training set of 50 examples with an accuracy of 99% (see three units result on Figure 2).

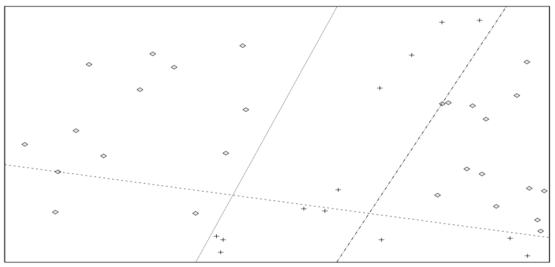

Figure 2: Test on a randomly generated network with 3 hidden units. A neural network with three hidden units was constructed from this partitioning, This is a minimal architecture.

B. Network Design Time

The speed-up of distributed genetic search is given by the following theorem.

Theorem 2: Let T_s be the time spent in designing a network's hidden layer using the algorithm described in Section 2 on a sequential machine, and $c * T_s$ $(0 < c < 1)$ the time spent computing the fitness. Then execution time T_p on a distributed environment of $n+1$ processors organized as described in Section 3 is

$$T_p = T_s(1 - c(1 - \frac{1}{n})) + \epsilon$$

where ϵ is the communication overhead.

Proof: Follows directly from the observation that the time spent computing the fitness in parallel is $\frac{1}{n}cT_s + \epsilon$. □

In practice, the communication overhead ϵ is small since genetic search consists of a constant number of generations and in each generation processes communicate just a single small message (a single string). Fraction of the sequential time c spent computing fitness depends on population size, and experiments indicate that $0.8 < c < 1$. Let us conservatively assume $c = 0.8$ (meaning that exactly 80% of sequential algorithm's time is spent computing the fitness). Then given $n+1$ processors and using genetic population of size n, a speed up of the hidden layer construction by a factor close to 5 can be attained by parallelizing the genetic search as proposed. In fact, then

$$T_p = \frac{1}{5}T_s(1 + \frac{4}{n}) + \epsilon.$$

It is easy to show the optimality of the dynamic pocket algorithm's generalization.

Theorem 3: Given a finite set of input vectors $\{\mathrm{x}^k\}$ and corresponding desired responses $\{y^k\}$ and a probability $P < 1$, there exists L such that after $l \geq L$ iterations of the dynamic pocket algorithm, the probability that the pocket weights are optimal exceeds P.

Proof: A straightforward extension of optimality result of [3]. □

The efficiency of the dynamic pocket algorithm was also tested on some standard benchmark problems. First test was on the Soybean database which consists of 307 instances belonging to 19 classes where each instance has 35 attributes. Based on the attribute values, the networks was trained to predict if the soybean crop suffered from one of the nineteen deceases. Second test was on already described Votes database.

Database	**Pocket**		**Dynamic**	
	C_{poc}	U_{poc}	C_{dpoc}	U_{dpoc}
Soybean	85	6	20	6
Votes	11	9	9	8

Table 2: Comparison between the Pocket and the Dynamic Pocket Algorithm

Window	**W**	C_{dpoc}	U_{dpoc}	T_W/T_F
Soybean	0.49	45	6	0.9
Votes	0.46	9	8	0.84

Table 3: Window size experiments for the Dynamic Pocket algorithm

The experimental comparisons between the pocket and the dynamic pocket algorithms are shown in Table 2. The number of stops for quality comparison between the current and the pocket hypothesis using the pocket and the dynamic pocket algorithm is denoted by C_{poc} and C_{dpoc} respectively. The number of useful stops for comparison for the pocket and the dynamic pocket algorithm is denoted by U_{poc} and U_{dpoc} respectively. For example, on Soybean problem the dynamic pocket algorithm stops 20 times and 6 of the stops are useful. In contrast, the original pocket algorithm waist significant time by 79 useless stops. Reduction of the number of useless stops using the dynamic algorithm is such a big gain that in all performed experiments even a sequential implementation of the dynamic pocket algorithm runs faster than the original pocket algorithm. In particular, our sequential implementation of the dynamic pocket algorithm learns Soybean database in 29.4 secs (CPU time on HP 9000/730) verses 71 secs needed by the pocket algorithm. If implemented as processes on two workstations, the Master and the Slave run in parallel thus reducing time further. A speedup of parallel dynamic algorithm over the sequential dynamic algorithm is network dependent (communication overhead is a function of network capabilities). Our experience on non-trivial learning problems shows speed-up close to double even on a local network of workstations with slow communication link.

The various experiments carried out with the dynamic pocket algorithm indicate that a smaller window of training examples for Master and Slave hypothesis comparison still gives a fairly good approximation of the quality obtained suing the full training set. A smaller data window achieves additional speedup since then each comparison takes less time. The experimental results on data window variations are shown in Table 3. The fraction of training set considered for quality estimation is denoted as W. Last column in the table $\frac{T_W}{T_F}$ is the ratio of the time taken by the dynamic pocket algorithm using W fraction of the training set for quality estimation to the time taken using the full training set. Practical experience indicates that window size between one-forth and three-fourths of the training set provides good prediction quality.

In summary, experiments from Tables 2 and 3 indicate that a speed up by a factor near 4 over the existing pocket algorithm can be attained using two processors for dynamic algorithm with reduced data window.

V. Conclusion

A new evolutionary algorithm has been proposed for learning functions computable on a single hidden layer feedforward neural networks. The algorithms has a constructive nature and adds hidden units incrementally using a genetic search over a relatively simple search space. The algorithm speed-up is achieved through parallelization. The proposed parallel algorithm is suitable for a distributed system implementation. Promising experimental results are obtained for both the constructed network complexity and generalization. Further research will show if the approach is usefull for multi hidden layer architectures.

References

[1] Congressional Quarterly Almanac, 98th Congress, 2nd session 1984, *Volume XL: Congressional Quarterly Inc.*, Washington, D.C., 1985.

[2] R.O. Duda and P.E. Hart, *Pattern classification and scene analysis.* New York: Wiley, 1973.

[3] S. I. Gallant, "Perceptron-Based learning algorithms", in IEEE Transaction on Neural Networks, Vol. 1, No. 2, pp. 179-191, 1990.

[4] S.E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," in D.S. Touretzky *Advances in Neural Information Processing Systems 2*, Morgan-Kaufmann, pp. 524-532, 1990.

[5] N. Karunanithi et al, "Genetic cascade learning for neural networks," *Proc. Int. Workshop on Combinations of Genetic Algorithms and Neural Networks*, pp. 134-145.

[6] R. Keesing and D. G. Stork, "Evolution and learning in neural networks," in R.P. Lippman et al (eds) *Advances in Neural Information Processing Systems 3*, Morgan-Kaufmann, pp. 804-810, 1991.

[7] M.A. Potter, "A genetic cascade - correlation learning algorithm," *Proc. Int. Workshop on Combinations of Genetic Algorithms and Neural Networks*, pp. 123-133.

[8] J.D. Schaffer et al, "Combinations of genetic algorithms and neural networks: A survey of the state of the art," *Proc. Int. Workshop on Combinations of Genetic Algorithms and Neural Networks*, pp. 1-37, 1992.

[9] S. B. Thrun et al., "The Monk's Problems: A performance comparison of different learning algorithms," Carnegie Mellon University, Tech. report CMU-CS-91-197, Dec 1991.

[10] D. Whitley, "The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best," *Proc. 3rd Int. Conf. on Genetic Algorithms,* pp. 116-121, 1989.