FACTA UNIVERSITATIS (NIŠ)
Ser. Math. Inform. 16 (2001), 97–108

PARALLELIZING DESIGN OF APPLICATION TAILORED NEURAL NETWORKS

Zoran Obradović and Rangarajan Srikumar

Abstract. In a companion paper, a constructive approach for designing feedforward neural networks using genetic algorithms is proposed [7, 8]. The algorithm constructs networks with close to optimum size growing hidden layer units in a problem specific manner and has very good generalization properties. In this paper, in order to make the constructive design algorithm computationally efficient, a two stage speed up method is proposed: (1) parallel genetic search for hidden layer units construction; and (2) the dynamic pocket algorithm for learning the hidden to output layer weights. The proposed parallel method achieves significant computational speed-up over the sequential method and is suitable for distributed implementation. In addition, the dynamic pocket algorithm can be used to speed up various other neural network constructive design methods.

1. Introduction

Determining the appropriate size of a neural network is one of the most difficult tasks in its construction. One of the main drawbacks of many neural network learning algorithms (e.g. backpropogation [10]) is that the architecture of the network must be pre-specified. Even for feedforward neural networks with a single hidden layer it is quite challenging to pre-specify the appropriate number of hidden units (usually decided based on the developers intuition).

An attempt to overcome the fixed architecture problem are constructive learning algorithms that grow or shrink the network in an application specific manner. In [7, 8] a two step constructive learning method is proposed. First

Received November 26, 1998.
2000 *Mathematics Subject Classification.* Primary 68Q70; Secondary 08A60, 08A70, 20M35.

a genetic algorithm is used as a tool for constructing hidden layer units along with the appropriate connection strengths. The pocket algorithm [3] is then used to learn the connection strengths between the hidden and the output layer. The algorithm generates a small network (close to optimum number of hidden units) with good generalization abilities. A drawback of this algorithm is that it is highly computation intensive, which makes it inappropriate for large scale problems. The goal of this work is to speed up the existing algorithm by parallelization, and thereby make it applicable to large real life problems.

A short description of the sequential algorithm is given in Section 2. A parallel version of the algorithm is proposed in Sections 3 and 4. The parallelization of the hidden layer construction is discussed in Section 3, and that of the hidden to the output layer learning in Section 4.

2. The Sequential Algorithm

The goal of the constructive learning algorithm proposed in [7, 8] is to design a small feedforward neural network with one hidden layer of threshold units that classifies well training set examples for a given m dimensional binary classification problem. Alternatively, it can be interpreted as a two step process: (1) construction of a small set of hyperplanes in Re^m (corresponding to hidden units of a neural network with a single hidden layer) which partition the training examples into regions each containing only training examples from a single class; and (2) learning of the hidden to output layer weights.

In the first step of the design process, pairs of training examples are formed, each pair consisting of two examples from opposite classes. Obviously, for perfect classification of training examples, for each of those pairs there must be a hyperplane separating the pair. The distance between points in a pair is discretized to k equal segments. One can visualize a slide that can occupy one of k discrete positions; each position from the midpoint of one segment to the midpoint of the next. For a perfect classification there must be a position of the slide such that the separating hyperplane separates the slide. For practical purposes the hyperplane can be assumed to pass through the center of the slide (for sufficiently large resolution k, size of the slide is sufficiently small). A hyperplane in Re^m is uniquely determined by m points. So, the equation of a separating hyperplane is defined by m appropriately positioned slides. We use a genetic algorithm to determine m appropriately positioned slides that define a good separating hyperplane

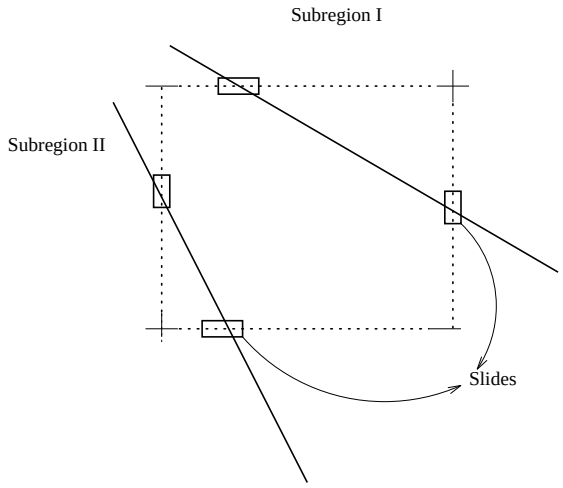

FIG. 1: The exclusive OR function realized by a neural network with two hidden units. The figure shows slides on the dotted lines between the points from the opposite classes. The slides can be in one of the k different positions on the dotted line.

(see Figure 1).

In the proposed evolutionary algorithm, the population is a set of hyperplanes. In each generation each of the hyperplanes in the population is evaluated for its fitness. If τ_i is the percentage of all the training examples from class i correctly classified by the hyperplane, then its fitness is defined as the sum of τ_i over all classes. A *region* is set of training examples defined by a bounding set of hyperplanes. A region is said to be *resolved* if all training examples falling in that region are of the same class, otherwise the region is *unresolved*. Initially we start from a single unresolved region which is our problem domain. Genetic search, for a pre-specify number of generations is performed to construct the best hyperplane which is used to partition the existing regions further. All the resolved regions are ignored in future generations because we already have a set of hyperplanes that can classify those regions correctly. The unresolved regions are maintained in a linked list. We continue to generate hyperplanes using genetic search until all the regions are resolved (the linked list is empty). The hidden layer units and the connections from the input to the hidden layer can be easily generated from the constructed hyperplanes.

The second step of the design process is to learn the connection strengths between the hidden and the output layer. This task is performed using the

pocket algorithm [3] which is a modification of the perceptron algorithm able to produce the optimal separation between non linearly separable classes.

3. Distributed Genetic Search

In the genetic search of the algorithm briefly summarized in Section 2 species in population correspond to hyperplanes in the problem domain and their fitness is the percentages of examples of each class classified correctly by the corresponding hyperplane. Genetic algorithm used in construction of separating hyperplanes creates species with better generalization capability from one generation to another. Thus, if a population consists of n species, in each generation the fitness has to be evaluated for all n species. making this evaluation the most expensive step in the algorithm. Our experiments with n ranging between 50 to 200 show that in the sequential implementation of the algorithm more than 80% of the time is spent computing the fitness. It also appears that the cost of this evaluation grows exponentially faster as compared to other costs. Fortunately, the estimation of the fitness value of the species are independent of one another, and this makes it an appropriate candidate 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 (see Figure 2). In the proposed distributed algorithm the Main node process

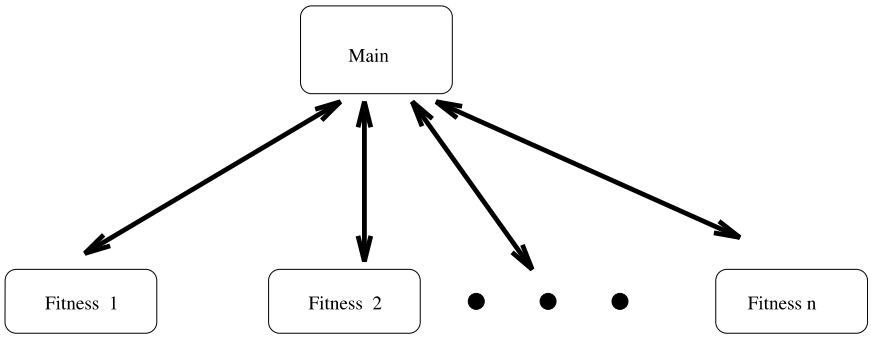

FIG. 2: Distributed implementation of the constructive genetic algorithm

initially broadcast the unresolved region (the training set) to all Fitness nodes. The Main node process executes the algorithm sequentially until there is a need to evaluate species based on their fitness. At that point, Main node process distributes n species (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 until 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. It is easy to show the following:

Theorem 3.1. *Let T_s be the time spent in executing the algorithm described in Section 2 on a sequential machine, and $c * T_s$ $(0 < c < 1)$ the time spent computing the fitness. Then parallel execution time T_p on a distributed environment of $n+1$ processors is given by*

$$T_p = T_s\Big(1 - c\Big(1 - \frac{1}{n}\Big)\Big) + \varepsilon,$$

where ε is the communication overhead.

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

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 once. Fraction of the sequential time c spent computing fitness depends on n, and our experiments for n ranging from 50 to 200 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 a speed up by a factor close to 5 can be attained by parallelizing the genetic search as proposed. In fact, then

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

4. Dynamic Pocket Algorithm

In the sequential algorithm of Section 2 weights from hidden to the output layer are learned using the pocket algorithm. Reason for this choice is the

fact that the hidden layer problem representation constructed by the genetic algorithm is not necessarily linearly separable.

In non-linearly separable problems no set of weights in perceptron can correctly classify all training examples. For such problems, it is desirable to reach so called optimal set of weights which gives smallest number of misclassification. The perceptron learning algorithm [2, 6, 9] is not well behaved for non-linearly separable problems. While it will eventually visit an optimal set of weights, it might not converge to any set of weights at all. Even worse, the algorithm can go from an optimal set of weights to a worst-possible set in one iteration, regardless of how many iterations have been taken previously. The pocket algorithm [3] is a modification of the perceptron algorithm, which makes perceptron learning an optimal set of weights with high probability even for non-linearly separable problems.

The basic idea of the pocket algorithm is to run the perceptron algorithm while keeping a backup hypothesis (weights assignment) "in pocket". 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 pocket algorithm is that the processes of estimating the better of the two hypothesis (perceptron and pocket) is extremely costly in terms of computation time. 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 our 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 (see Figure 3).

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 until 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

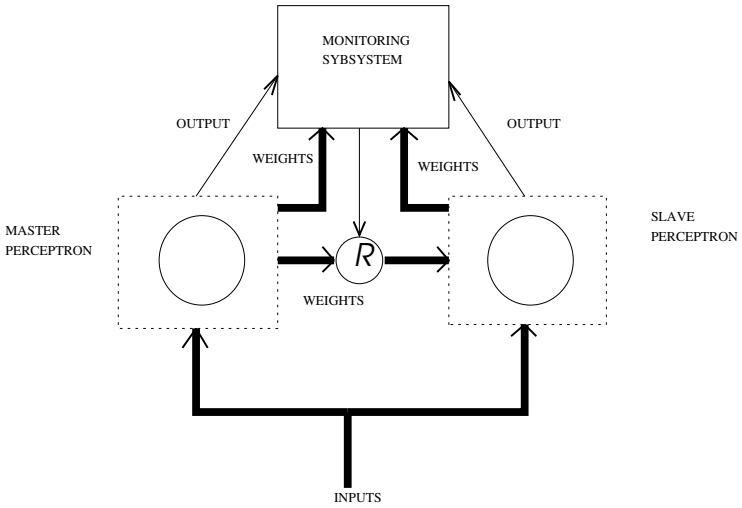

FIG. 3: Schematic representation of the Dynamic Pocket Algorithm

Master and the Slaves hypothesis (i.e., Π and π). If the Master hypothesis is found to be better, the Monitor subsystem sends a reset signal R which causes the replacement of Slaves weights by that of the Master. The dynamic pocket algorithm can be described more formally as follows.

DYNAMIC POCKET ALGORITHM

Input: A set of training examples $T = \{(\mathbf{x}^k, y^k)\}$. Here $\mathbf{x}^k \in \{-1, 1\}^m$ is the vector of features including bias, and $y^k \in \{+1, -1\}$ is the desired response.

Temporary Data:

Π = Master hypothesis, $\Pi \in Z^m$.

π = vector of Slave hypothesis, $\pi \in Z^m$.

Φ = current number of consecutive correct classifications using Master hypothesis Π.

φ = current number of consecutive correct classifications using Slave hypothesis π.

ϕ = maximum number of consecutive correct classifications using Slave hypothesis π.

num_ok_Π = total number of training examples that Π correctly classifies.

num_ok_π = total number of training examples that π correctly classifies.

Algorithm:

(1) initialize π and Π to random set of weights.

(2) set $\varphi = \Phi = num\_ok_\Pi = num\_ok_\pi = \phi = 0$.

(3) randomly pick a training example $\mathbf{x}^k$ (with corresponding classification y^k).

(4) if Π correctly classifies $\mathbf{x}^k$, (i.e, $sgn(\Pi \mathbf{x}^k) = sgn(y^k)$) then

(4a) set $\Phi = \Phi + 1$.

else

(4b1) set $\Phi = 0$.

(4b2) (*Update step*) Form a new Master hypothesis as $\Pi = \Pi + y^k \mathbf{x}^k$.

(5) if π correctly classifies $\mathbf{x}^k$ then

(5a1) set $\varphi = \varphi + 1$

(5a2) if $\varphi > \phi$ then

(5a2a) set $\phi = \varphi$

(5a2b) if all training examples are correctly classified (i.e $\phi = |\{\mathbf{x}^k\}|$) then stop; the training examples are separable.

else

(5b) set $\varphi = 0$

(6) if $\Phi > \phi$ then

(6ba) randomly select a subset of $0.6T$ training examples.

(6ba) compute num_ok_Π by checking every selected training example.

(6bb) if $num\_ok_\Pi > num\_ok_\pi$ then

(6bba) set $\pi = Pi$

(6bbb) set $\varphi = \phi = \Phi$

(6bbc) set $num\_ok_\pi = num\_ok_\Pi$

(7) end of this iteration. If the specified number of iteration has not been performed then go to step 3. Otherwise output π.

It is easy to to show the following result:

Theorem 4.2. *Given a finite set of input vectors* $\{\mathbf{x}^k\}$ *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 [4]. □

Note that in the dynamic pocket algorithm the threshold ϕ for evaluation of the current hypothesis Π is subject to dynamic updation unlike the original pocket algorithm (i.e., ϕ increases to φ if $\varphi > \phi$). Thus, a significant computing time is saved by reducing the number of useless quality evaluations of the current master's hypothesis. The experimental results consistent with this observation are reported as follows.

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

Table 4.1: Comparison between the Pocket and the Dynamic Pocket Algorithm

Window	**W**	Q_{dpoc}	U_{dpoc}	T_W/T_F
Soybean	0.49	45	6	1
Votes	0.46	9	8	0.84

Table 4.2: Window size experiments for the Dynamic Pocket algorithm

Testing Results of the Dynamic Pocket Algorithm

The efficiency of the Dynamic Pocket algorithm was tested on two standard benchmark problems. First test was on the Soybean Database which is a noisy domain consisting 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 the Votes Database. This data set includes votes for each of the U.S House of representatives congressmen on the 16 key votes identified by the Congressional Quarterly Almanac (CQA) [1]. The CQA contains nine different types of votes: voted for, paired against, voted

against, and announced against. The database consisting of 435 instances of 2 classes with each instance having 17 attributes.

The experimental comparisons between the pocket and the dynamic pocket algorithms are shown in Table 4.1. The number of stops for quality comparison between the current and the pocket hypothesis using the pocket and dynamic pocket algorithm is denoted by Q_{poc} and Q_{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. Reduction of the number of useless stops using the dynamic algorithm is such a big gain that in all our 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 seconds (CPU time on HP 9000/730) verses 71 seconds needed by the Pocket algorithm. In a parallel implementation the Master and the Slave run in parallel thus reducing time further. Speedup of parallel dynamic algorithm over the sequential dynamic algorithm is network dependent (communication overhead is a function of network capabilities), but in practice for non-trivial learning problems it is close to double.

The various experiments carried out with the dynamic pocket algorithm indicate that a smaller data window for Master and Slave hypothesis comparison gives a fairly good approximation of the quality obtained with the full training set. Using smaller data window each comparison takes less time and consequently additional speedup can be achieved. The experimental results on data window variations are shown in Table 4.2. 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. In practice, window size varying between half and three-fourths of the training set provides good prediction quality.

In summary, experiments from Tables 4.1 and 4.2 indicate that a speed up by a factor near 4 can be attained using parallel dynamic algorithm with window $W = 0.6T$ (where T is the size of the full training set) over the existing pocket algorithm.

5. Conclusion

A parallelization of our recent constructive neural network design algorithm is proposed in this paper. It is shown that an almost optimal speedup is achievable on parallelizing genetic search for hidden layer construction.

The additional speed up is achievable on the output layer learning using parallel dynamic pocket algorithm. The dynamic pocket algorithm can also be used instead of the traditional pocket algorithm in various constructive algorithms including tower algorithm [4] and tiling algorithm[5]. In those learning methods the pocket algorithm is used extensively to construct every unit of the network. So, the dynamic pocket could indeed speed up these existing constructive algorithms significantly. Further research is needed in order to characterize the level of speedup achievable for other constructive algorithms by using proposed dynamic pocket learning.

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: *Optimal linear discriminant.* In: Proc. Eight Int. Conf. Pattern Recognition (Oct. 28–31, 1986, Paris, France), pp. 849-852.
4. S. I. Gallant: *Perceptron-based learning algorithms.* IEEE Transaction on Neural Networks **1**, No. 2 (1990), 179–191.
5. M. Mézard and J. P. Nadal: *Learning in feedforward layered networks: The tiling algorithm.* Journal of Physics **21** (1989), 2191–2204.
6. M. Minsky and S. Papert: *Perceptrons: An Introduction to Computational Geometry.* Cambridge, MA: MIT Press, 1969.
7. Z. Obradovic and R. Srikumar: *Evolutionary design of application tailored neural networks.* In: Proc. IEEE Int. Symp. on Evolutionary Computation, Orlando, FL, 1994, pp. 284–289.
8. Z. Obradovic, R. Srikumar: (in review) *Constructive neural networks design using genetic optimization.*
9. F. Rosenblatt: *Principles of Neurodynamics: Perceptrons and the theory of brain mechanisms.* Washington, D.C: Spartan Press, 1961.
10. P. Werbos: *Beyond Regression: New Tools for Predicting and Analysis in the Behavioral Sciences.* Harvard University, Ph.D. Thesis, 1974; Reprinted by Willey & Sons, 1995.

School of Electrical Engineering
and Computer Science
Washington State University
Pullman, WA 99164-2752, USA
and
Mathematical Institute
Knez Mihailova 35
11000 Belgrade, Yugoslavia

Microsoft Corporation
3219 Building 16, One Microsoft Way
Redmond, WA 98052-6399, USA