Empirical Learning of Boolean Functions Using Two-Level Logic Synthesis

Arlindo L. Oliveira **Alberto Sangiovanni-Vincentelli**
Dept. of EECS
UC Berkeley
Berkeley CA 94720

July 9th, 1991

Abstract

Most approaches to the design of networks that learn from examples don't address the architecture design problem except as a side issue. Logic synthesis techniques can be used to derive both the structure and the connection patterns for a network that matches the examples in the training set while minimizing some cost function such as the size of the network. This cost function yields a very effective solution for the classification of examples not present in the training set. Thus, minimizing the size of the network can be considered as an effective generalization principle in accordance with the *Occam's razor* paradigm.

In this paper we restrict our attention to the use of two-level *and-or* networks. An efficient algorithm for the synthesis of networks of this type is proposed and the quality of the generalization performed by the network is evaluated against alternative approaches.

The algorithm was tested in several test problems taken from the machine learning literature. Results show that, in most cases, the proposed approach outperforms other popular methods, both in accuracy and in the number of examples needed for accurate generalization.

1 Introduction

In the standard model of supervised learning, the task of the learning algorithm is to induce, from a set of positive and negative instances (the training set), a rule that will accurately predict the class of future instances (the test set).

Many supervised connectionist models use gradient descent in the error [19] to derive the pattern of weights between units. The problem with this approach is that the solution is not guaranteed to generalize well, since many solutions that fit the training set will not give the correct results in the test set.

In general, the smaller the network, the better the generalization obtained. Theoretical [2, 5, 17] and practical [11, 12] results show that the quality of the generalization improves with a reduction in the size of the network. However, smaller networks are usually more difficult to train [9]. Ideally, one would like a network just large enough to learn the mapping. Several procedures were proposed to achieve simple networks, either by adding extra units to small networks [1, 7] or by adding an extra term to the cost function that helps to minimize the number of *active* weights [20]. This algorithms, however, still consider the architecture design more like a side issue than as one of the major problems to be solved in the design of a practical neural network.

A different way to view the problem is to consider the design of the architecture as a fundamental part of the problem. This is the approach that is taken in the field of integrated circuits design. In general, a specified input-output specification exists for a given number of input patterns, and architectures are designed to perform the desired mapping in the most economical way.

Therefore, a strong motivation exists to apply some of the vast array of techniques used in traditional circuit design in the definition of architectures for general networks. In this paper, the possible uses of two-level logic synthesis techniques in empirical learning are investigated. Under this restriction, the algorithm is closely related to induction systems based on set covering algorithms, like, for example, AQ15 [13]. Viewing the problem as a logic synthesis task, however, opens new directions for extensions of the techniques used, either by considering other types of gates [14] or by using multi-level design techniques [4] to generate networks with more flexible architectures.

One of the advantages of this approach is that a hardware implementation of the network using standard IC technology is straightforward. The implementation is also very small and extremely fast as compared to other neural network solutions for the problem.

2 Basic concepts

2.1 Concepts

We will use the usual setting for the problem of learning from examples in an attribute-based description language. Let B_N be the set of N objects in the instance space. Concepts and hypothesis are subsets of B_N. The task of learning from examples can be described as follows: given m objects, chosen from B_N according to some arbitrary distribution and labeled positive or negative according to whether they belong or not to a given concept C, derive an hypothesis H which is a good approximation to the concept C. How closely H approximates C is usually measured by testing the generated hypothesis in an independent set of examples drawn from the same distribution.

2.2 Cubes and Covers

Let f be a function of n boolean variables, $\{x_1, ..., x_n\}$. Formally, f is a mapping from $\{0,1\}^n \rightarrow \{0,1\}$. A literal is a variable or its negation. A cube (or monomial) is a conjunction of l literals $(1 \leq l \leq n)$, where no two literals corresponding to the same variable appear.

A cube with n literals is called a minterm, and it corresponds to a vertex in the n dimensional hypercube that represents the input space[1]. A cube c_1 is said to cover another cube c_2 if $c_2 \Rightarrow c_1$, i.e., if the truth values defined in c_2 make c_1 true. If $c_1 \neq c_2$, then c_1 properly covers c_2.

A single output boolean function f, is described by the following disjoint sets of minterms: f_{ON}, f_{OFF} and f_{DC}. The f_{ON} (f_{OFF}) set specifies which minterms should turn the output on (off). Non trivial boolean functions should have non-empty f_{ON} and f_{OFF} sets. The f_{DC} (don't care) set contains all minterms not contained in the previous sets.

A set of cubes is called a cover of f if all minterms in the f_{ON} set of a function are covered by at least one cube, and no cube covers any minterm in the f_{OFF} set. Given a cover, there is a trivial realization of a two level network that realizes the desired function: simply allocate one *and* gate for each cube and connect them all to an *or* gate in the second level.

A cover is said to be irredundant if the removal of any cube from the cover produces a set that is no longer a cover. For a given function, a cube c is called a prime cube if there is no other cube that does not intersect the f_{OFF} set and properly contains c. A cover consisting only of prime cubes is called a prime

[1] By extension, a minterm also defines a truth assignment of all the input variables.

cover. If a prime cube is contained in all prime covers of a function, then such a prime is essential. The existence of essential primes greatly simplifies the task of finding a minimum cover. In particular, if all primes are essential, such a task can be achieved in a time polynomial in the number of f_{ON} minterms.

In general, the problem of finding a minimum cover for a given function is NP-complete [8]. Hence, most algorithms are heuristic and find only approximate solutions using reasonable amounts of time. Unlike other training algorithms, the resulting solution is always guaranteed to represent a network that performs the right input/output mapping, even if the absolute minimum was not found.

2.3 Expanding and Reducing cubes

Most heuristic algorithms are based on the incremental modification of a cover obtained by applying one of the following two operations to the cubes in that cover:

- Expand : Drop one literal from the cube. This operation is only legal if the resulting cube does not contain any minterm in the f_{OFF} set. The resulting cube properly contains the original one.
- Reduce : Add one literal to a cube. This operation is only legal if the f_{ON} minterms no longer covered by this cube are still covered by some other cube in the cover. The resulting cube is properly contained in the original one.

3 The synthesis procedure

The objective of the synthesis procedure described here is to derive a network that performs the input-output mapping specified by the set of examples while minimizing the overall size of the network, as measured by the number of gates. In logic synthesis notation, the positive examples define the f_{ON} set, the negative ones the f_{OFF} set and the rest of the minterms will be in the f_{DC} set.

The algorithms used in logic synthesis, like, for instance, ESPRESSO [3], could conceptually be used in our approach. However, the type of boolean functions that is common in machine learning problems is relatively unimportant in classical synthesis. In particular, the presence of very large don't-care sets[2] is quite uncommon in traditional logic synthesis problems. Finally, the size of the input in some cases is far too large to be handled by general purpose logic minimizers.

[2] When compared to the sizes of the f_{ON} and f_{OFF} sets.

3.1 Finding a small cover

As described before, from the list of positive and negative examples in the training set, a two-level *and-or* network that performs the desired mapping can be readily derived. The starting point is simply an *or* of minterms. This network (or the cover it corresponds to) is then reduced in size until no further improvement is obtained.

We use a branch and bound algorithm to search for a more compact cover. A search tree is built with a predefined branching factor r, and, at each step, the most promising node in the tree is expanded.

Every node n_i in the search tree corresponds to a set s_i of cubes and has a value given by $v_i = Ku - m$, where u is the number of uncovered minterms, m is the number of minterms covered more than once and K some large constant. If v_i is non-positive, then node n_i represents a solution with one less cube than the root. A new node n_j is created by modifying the cover of a given node in the tree n_i (the parent node). Set s_j is obtained by modifying the set s_i of the parent node with one of the following two operations:

1. Removing one cube from the set.
2. Changing one cube so that it covers a new minterm.

Children of the root node are created applying operation 1. Since operation 1 removes one cube from the cover, any solution that is found is going to have one less cube than the original solution. All other nodes (except the root) are created using operation 2. Operation 2, the creation of children of node n_i by changing one of the cubes in the cover is performed by the following algorithm:

$m \leftarrow$ pick_one_uncovered_minterm(n_i, f_{ON});
for $i = 1$ to r do $\{$
    $c_{old} \leftarrow$ choose_next_cub(m);
    $g_{ON} \leftarrow$ build_on_set(m, n_i, c_{old});
    $c \leftarrow$expd_cube(c_{old}, g_{ON}, f_{OFF});
    $n_j \leftarrow$ create_child($n_i, s_i \cup \{c\} \setminus \{c_{old}\}$);
    $v_j \leftarrow$ evaluate(n_j);
$\}$

The expansion of a node works by choosing the r closest candidate cubes that can be expanded to cover a given (and still uncovered) on set minterm.

Function build_on_set creates the g_{ON}[3] set by adding minterm m to the list of

[3] In general, this g_{ON} set will not be equal to the f_{ON} set.

minterms chosen to be covered by c. Procedure create_child simply adds one more children to node n_i, with a set of cubes appropriately modified.

Function expd_cube returns the maximally reduced cube covering all minterms in the g_{ON} set and none in the f_{OFF} set. If no such cube exists it reports failure and no child node is created.

Readers familiar with two level logic simplifiers will notice that the algorithm does indeed perform the normal steps, i. e. expansion and reduction of cubes. However, unlike these algorithms, the expansion and reduction phases are not performed in batch, but individually for each cube, in order to make some particular cube redundant. Empirical comparisons [15] have shown that this procedure is more efficient and less time consuming. However, it depends on the fact that all examples are minterms, and cannot be used as a general two level logic minimizer.

As an example, consider the function defined by its f_{ON} and f_{OFF} sets shown in figure 1:

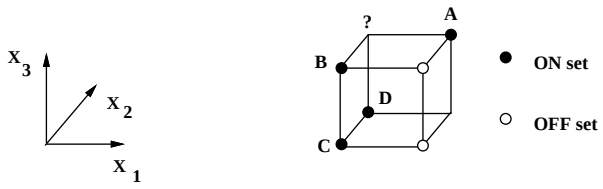

Figure 1: *On* and *Off* sets of function f.

An hypothetic search tree is shown in figure 2. The initial solution in level 0 consists simply of a cover formed by all the f_{ON} minterms of function f. The first level nodes in the tree are constructed by removing one cube from the cover. Cubes 2 and 4 were selected for illustration purposes. In the left branch, this leaves the minterm D uncovered and cubes 1 and 3 are chosen to be expanded. A similar situation exists in the right branch. In this trivial example, all 4 level 2 nodes in the tree are solutions, but, in general, deeper search trees are needed, since removal of one cube may leave several minterms uncovered. The network in figure 3 represents the actual implementation of solution S_4.

Figure 2 also illustrates how generalization takes place. If solution S_4 is taken as the final solution, the f_{DC} point marked with **?** in figure 1 would have been considered as belonging to the concept that generated the positive examples that define the f_{ON} set.

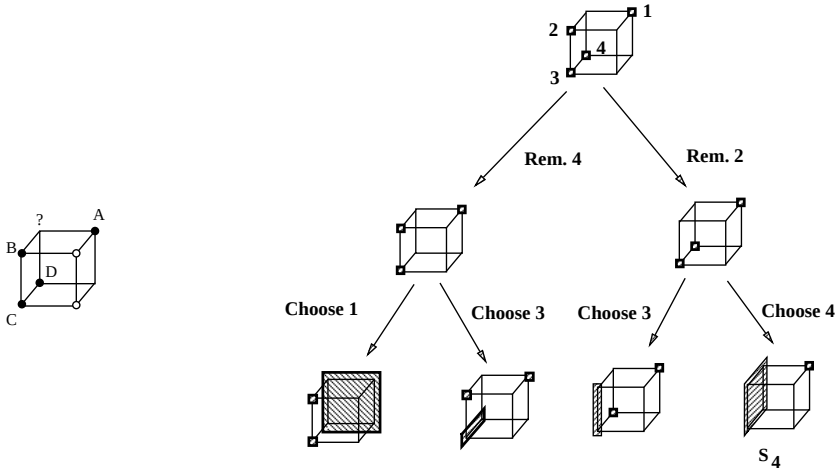

Figure 2: Search tree used to find a compact cover for function f.

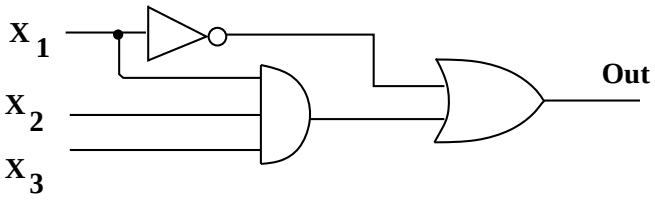

Figure 3: Implementation of solution S_4

4 Empirical evaluation

4.1 Experimental setting

The performance of the algorithm was tested in three sets of problems: the first was proposed by Pagallo and Haussler in [18] and consists of 8 different concepts that accept compact DNF descriptions. The second test set is extracted from chess game endings and was proposed by Quinlan in [16]. Finally, the third consists in the recognition of segmented and normalized handwritten digits using the ZIP code database provided by the US Postal Service.

4.2 Concepts with a compact DNF representation

A detailed description of the test concepts used in this section can be found in [18], including the process by which the terms in the randomly generated functions were obtained.

Some information about the test concepts is given in table 1. The number of terms is the number of monomials in the most compact DNF representation of the concept. The number of examples was selected as in [18], according to the formula $\frac{K*\log_2(N)}{\epsilon}$, where N is the number of attributes, K the number of literals needed to write down the smallest DNF description of the target concept and ϵ, the upper limit on the error, was set to 0.1.

name	description	# attributes	# terms	# examples
dnf1	random DNF	80	9	3292
dnf2	random DNF	40	8	2185
dnf3	random DNF	32	6	1650
dnf4	random DNF	64	10	2640
mx6	6-multiplexer	16	4	720
mx11	11-multiplexer	32	8	1600
par4	4-parity	16	8	1280
par5	5-parity	32	16	4000

Table 1: Target concepts

The first four are randomly generated DNF functions, with varying numbers of terms and variables. The N-multiplexer examples are taken from the multiplexer domain. For an N-multiplexer, the first $\lfloor \log_2 N \rfloor$ bits act as selectors for one of the

following $N - \lfloor \log_2 N \rfloor$ bits, which gives the value of the function. Finally, the last two ones are taken from the exor domain, and the value of the function is given by the *exclusive or* of the first N bits. A given number of random bits (corresponding to irrelevant attributes) is added to each of the last 4 examples (respectively 10, 21, 12 and 27). A complete description of the minimum DNF representations of these functions can be found in [18].

The examples were generated using *random*, a pseudo-random number generator available in most unix systems. The number of nodes in the search tree was restricted to 1000, although a much smaller number was actually used in all examples but the last.

We compare the results obtained by our algorithm with the best results described in [18]. These results were obtained by choosing the tests to be made in each node of the decision tree according to a specific heuristic, named *fringe*.

Table 2 lists the error incurred by each one of the algorithms in a test set of 2000 examples. The results shown are the average of 10 independent runs for each example.

target	LogSynt		Fringe	
concept	Aver. Err (%)	Aver. # terms	Aver. Err (%)	Aver. Tree Size
dnf1	0.0	9.0	0.0	9.0
dnf2	0.015	8.0	0.5	7.6
dnf3	0.04	6.1	0.3	6.1
dnf4	0.0	10.0	0.0	10.0
mx6	0.0	4.0	0.0	4.9
mx11	0.0	8.0	0.0	11.6
par4	0.0	8.0	0.0	4.9[4]
par5	0.0	16.0	22.1	120.7

Table 2: Results

Also interesting is to analyze how the performance of the classifiers varied with the number of examples used. The evolution of the performance with the size of the training set size is shown in figure 4 for one of the examples, *dnf4*. For reference, we also show the performance of a non-modified decision tree algorithm and a gradient descent algorithm[5]. We used a conjugate gradient method [10] that

[4] This value, reported in [18] is probably a misprint, since the minimal DNF description of this concept requires 8 terms.

[5] Data for *fringe* was obtained from graphical information on [18].

performed better than standard back-propagation in this example. We first selected the optimal number of nodes in the intermediate layer (for a training set size of 2640) and then used that network for different training set sizes. Computation was stopped after convergence was obtained or 2 days of CPU time in a DECstation 3100 was reached. Due to the computational requisites, only 3 different experiments were performed with each training set size for this learning algorithm.

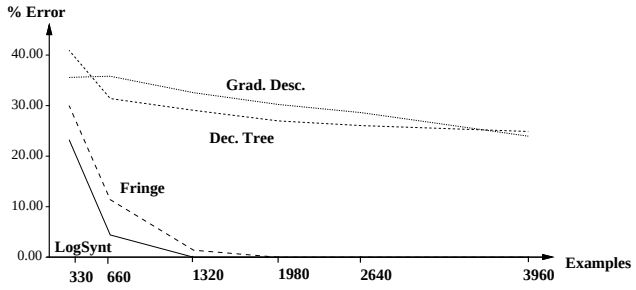

Figure 4: Error x Training Set Size for *dnf4*

4.3 Chess endings

The previous examples all accepted compact DNF descriptions and the good generalization results obtained by a method that searches for compact DNF realizations are somehow to be expected. It is interesting to analyze how the method compares with alternative approaches in cases where no compact DNF representations are known to exist.

The (white) king-rook versus (black) king-knight chess ending was proposed by Quinlan in [16] as a test case for the ID3 algorithm, an induction system that implements a classification procedure based on decision trees.

Every position in this chess ending can be considered as either lost for black or drawn[6]. In particular, a position is considered as *lost N-ply* iff the best possible sequences of no more than N moves lead to a position where

- Black king is in checkmate, or
- The black knight was captured, the white rook is safe and the position is not stalemate.

We used the *lost 3-ply* positions as a test bed. With the codification used by Quinlan, a total of 551 different vectors with 39 attributes each exist. We randomly

[6] For simplicity, the extremely rare cases where black can mate are ignored.

selected training sets of sizes from 100 to 400 and compared the results with the ones obtained by a decision tree algorithm, using, as the test set, the full set of vectors. Table 3 shows these results. The first two columns show the results obtained by the decision tree algorithm and by the LSAT algorithm when a network with one output is used. Although these results show that the error rate is significantly lower for the logic synthesis algorithm, even more interesting results are obtained when a network with 2 outputs is synthesized. One of the outputs signals a position that is considered lost and the other one signals a position that is considered drawn. When none of the outputs is *on* or when both are *on*, we consider the example as unclassified. The last two columns of table 3 give the percentage of unclassified inputs and the error rate in the classified ones.

Training	Dec. Trees	Synt. 1 output	Synt. 2 outputs	
set size	Error (%)	Error (%)	Error (%)	Unclass. (%)
100	18.3	20.0	4.5	23.8
200	12.7	8.5	2.4	11.1
300	10.5	3.8	0.6	6.2
400	6.7	2.5	0.2	4.5

Table 3: Results in the chess endings problem.

4.4 Handwritten ZIP code digits

A problem much harder than the previous ones is the recognition of handwritten characters. To test the algorithm we used the data provided by the US Postal Service that data consists of approximately 10000 digits. Each digit was segmented and normalized to a 16x16 binary grid. We then synthesized a network with 256 inputs and 10 outputs that gives the correct output for every one of the 8000 digits selected to be included in the training set. The resulting network was tested in an independent test set consisting of approximately 2000 digits.

As described in the previous section we considered as unclassified all the digits in the test set that turned on zero or more than one of the 10 outputs. We obtained a 2.0% error by rejecting as unclassified 30% of the characters. These results are worst than the ones reported in [6] (1% error and 9% rejection) but were obtained without using any field-specific knowledge.

4.5 Analysis of the results

The results show that the synthesis of minimal two-level circuits is an efficient alternative approach to the problem of learning from examples.

Although the method is computationally more expensive than decision trees, it is still much more efficient than gradient descent algorithms. For the larger examples, the synthesis algorithm is faster by at least two orders of magnitude than a gradient descent method. No data is available for the computation times involved in the use of modified decision trees that use complex heuristics to generate higher level attributes, like, for instance, *fringe*.

Finally, the comparatively bad results obtained in the problem of handwritten character recognition are probably a sign that no compact two-level solutions exist for these problems. In [6], a much deeper network was used, together with extensive knowledge about the topology of the problem and the relevant features to extract. For the algorithm to be successful in these harder problems, more flexible architectures and network constraints based on field-specific knowledge have to be used.

5 Conclusions

We have presented and evaluated a novel approach for the problem of learning from examples based on the synthesis of minimal networks.

Although currently restricted to the synthesis of two-level networks, there are several interesting directions for future work in this area: synthesis of multi-level networks, synthesis of multi-valued networks and generalization of the cost functions used.

Two level architectures are, in general, restrictive, versus multi-level networks. Logic synthesis algorithms may realize the same logic function using less gates if allowed to use multi-level structures. Several important problems are unlikely to accept compact solutions in the form of two level networks. Since multi-level techniques are based on the extraction of common factors between different functions in the network, they are likely to be an efficient method for the choice of useful higher level features in problems that don't accept compact two-level solutions. The challenge is to efficiently use the don't-care points in the minimization process since current multi-level synthesis algorithms use don't care information in a very limited way.

The applicability of these techniques is not restricted to problems where the attributes are boolean. If the attributes have more than two possible values but can be discretized in a finite number of discrete values, multi-valued synthesis algorithms

can be applied, giving, as an additional result, an appropriate codification for the multi-valued variables.

Robustness to noisy exemplars is another topic that deserves further research. Currently, a small number of erroneous examples may seriously damage the performance of the system. This disadvantage is easily overcome by changing the cost function that one wants to minimize in order to accept a small number of misclassified examples if that helps to minimize the cost of the network.

Acknowledgments: This work was supported by the Air Force Office of Scientific Research (AFOSR/JSEP) under Contract No. F49620-90-C-0029 and the Portuguese INVOTAN committee.

References

[1] Ash, T. “Dynamic Node Creation in Backpropagation Networks”, Connection Science, 1:4, pp. 365-375, 1989.

[2] Baum, E. B., Haussler, D. “What Size Net Gives Valid Generalization”, Neural Computation 1:1, 1989.

[3] Brayton, R. K., G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli “Logic Minimization Algorithms for VLSI Synthesis”, Kluwer Academic Publishers, 1984.

[4] Brayton, R. K., Hachtel, G. D., Sangiovanni-Vincentelli, A. L. “Multilevel Logic Synthesis” Proceedings of the IEEE, Vol. 78:2, February 1990.

[5] Blumer A., Ehrenfeucht, A., Haussler, D., Warmuth, M. K. “Occam’s Razor”, Information Processing Letters, 24:377-380, April 1987.

[6] Le Cun, Y., Boser, B., Denker, J. S., Henderson, D., Howard. R. E., Hubbard, W. and Jackel, L. D. “Handwritten Digit Recognition with a Back-Propagation Network” in Touretzky, D., editor, Advances in Neural Information Processing Systems, Vol. 2, pp. 524-532, Denver, 1989.

[7] Fahlman, S. E., Lebiere, C. “The Cascade-Correlation Learning Architecture” in Touretzky, D., editor, Advances in Neural Information Processing Systems, vol. 2, pp. 524-532, Denver, 1989.

[8] Garey, M. R. and Johson, D. S. “Computers and Intractability: A Guide to the Theory of NP-Completeness”, W. H. Freeman, San Francisco, 1979.

[9] Hanson, S. J. and Pratt, L. Y. “Comparing Biases For Minimal Network Construction With Back-Propagation” in Touretzky, D., editor,

[10] Alan Kramer, A. Sangiovanni-Vincentelli, “Efficient Parallel Learning Algorithms for Neural Networks” in Touretzky, D., editor, Advances in Neural Information Processing Systems, vol. 1, pp. 40-48, San Mateo, CA, 1989.

[11] Le Cun, Y. “Generalization and Network Design Strategies” in Pfeifer, R., Schreter, Z., Fogelman, F., and Steels, L., editors, Connectionism in Perspective, Zurich, Switzerland, Elsevier.

[12] Le Cun, Y., Denker, J. S. and Solla, S. A. “Optimal Brain Damage” in Touretzky, D., editor, Advances in Neural Information Processing Systems, vol. 2, pp. 598-605, Denver, 1989.

[13] Michalski, R.S., Mozetic, I., Hong J., and Lavrac, N. “The multipurpose incremental learning system AQ15 and its testing application to threee medical domains”, Proceedings of the Fifth National Conference on Artificial Intelligence, pp. 1041-1045, Philadelphia, PA, 1986.

[14] Oliveira, A.L. and Sangiovanny-Vincentelli, A. “Learning Concepts by Synthesizing Minimal Threshold Gate Networks”, Proceedings of the Eigth International Workshop in Machine Learning, pp. 193-197, Chicago, IL, 1991.

[15] Oliveira, A.L. and Sangiovanni-Vincentelli, A. “LSAT - An Algorithm for the Synthesis of Two Level Threshold Gate Networks”, Proceedings of ICCAD-91, Santa Clara, CA, 1991.

[16] Quinlan, J. R. “Learning Efficient Classification Procedures and Their Application to Chess End Games” in Michalski, R., Carbonell J. and Mitchell, T., editors, Machine Learning - An Artificial Intelligence Approach, 1983.

[17] Pearl, Judea “On the Connection Between the Complexity and Credibility of Inferred Models”, Intern. Journal Gen. Systems, 4:255-264, 1978.

[18] Pagallo, Giulia. and Haussler, David “Boolean Feature Discovery in Empirical Learning”, Machine Learning, 5:71-99, 1990.

[19] Rumelhart, D. E. and J. L. McClelland, editors. “Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1, Foundations”, MIT Press, Cambridge, 1986.

[20] Rumelhart, D. E. “”Parallel Distributed Processing”, Plenary Lecture presented at the IEEE International Conference on Neural Networks, San Diego, July 1988.