THE EVOLUTION OF THRESHOLD LOGIC NETWORKS WHICH RECOGNIZE BINARY PATTERNS

T. A. Marsland, D. L. Johnson
Department of Electrical Engineering
University of Washington
Seattle, Washington 98105

Abstract

Any binary pattern can be represented by a partially specified Boolean function. These functions in turn can be recognized by a network of trainable threshold logic units. A heuristic program for recognizing Boolean functions is discussed and a method of classifying functions according to the distribution of minterms is suggested.

Introduction

One of the basic aims of this study was to examine ways in which artificial intelligence techniques can be used to solve difficult problems. When studying a simple process, one of the hazards is that the experimenter will bias his solution methods by implicitly assuming the form of the solution. However, when examining an unsolved problem the experimenter's experience is limited to that obtained through interaction with his own program. In a very real sense this becomes a joint learning process, the task of the machine being to enhance the discovery capabilities of the user.

For our model, a doubly interactive scheme was chosen. On the one hand, the program itself was inquisitive and tried to predict the nature of its environment and base its decisions on that prediction. On the other hand, the researcher was allowed to interact with the program during its execution; to alter some of the decision-making parameters. Naturally, in order to improve the performance of the system, it was also necessary to make evolutionary changes to the whole program. However, while making those changes, the long-term memory (experience) of the system was not disturbed.

The specific problem that we wish to discuss is that of recognizing binary patterns through the use of a network of adaptable elements, a network which is capable of growth and decay. Any finite sequence or string of binary digits can be regarded as a partially specified Boolean function, provided a one to one correspondence is drawn between each binary digit in the sequence and the minterms - see Phister[1] - of the Boolean function. When displayed on a Veitch-Karnaugh map or on the vertices of a N-dimensional cube, these sequences are simply binary patterns. By such means a N-variable Boolean function can represent a 2^N bit pattern.

Terminology

A trainable Threshold Logic Unit (TLU), of which Figure 1 is an example, is the basic element in our network. It is well known that a single trainable TLU can be made to recognize a restricted set of Boolean functions, and many algorithms have been developed for that purpose. A discussion of the error-correcting training procedures is to be found in Nilsson[2] while the linear programming approaches are conveniently tabulated by Ho and Kashyap[3].

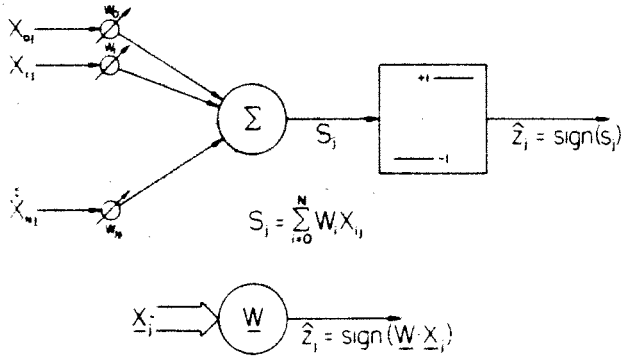

Figure 1

The basic TLU shown in Figure 1 is characterized by a weight vector $\underline{W} = (w_0, w_1, \ldots, w_N)$. Upon excitation by a set of ($\pm$1) binary inputs, $\underline{X}_j = (x_{0j}, x_{1j}, \ldots, x_{Nj})$, the device responds with a ($\pm$1) binary output $\hat{z}_j$ according to the equation

$$\hat{z}_j = \text{sign}\left(\sum_i w_i x_{ij}\right) = \text{sign}(\underline{W} \cdot \underline{X}_j),$$

where sign (0) = -1. In this formulation x_{0j} is set equal to -1 so that w_0 may correspond directly to the "threshold" of the device. The other components of the input vector $\underline{X}_j$ are either +1 or -1 and are obtained from the corresponding minterm.

For illustration purposes a special ordering of the minterms was chosen so that the 2, 3 and 4-variable Veitch diagrams could appear imbedded in an obvious manner in the 5-variable Veitch. The full need for this feature is explained in Reference 9, but is amply illustrated in Figure 2, which shows the ordering of

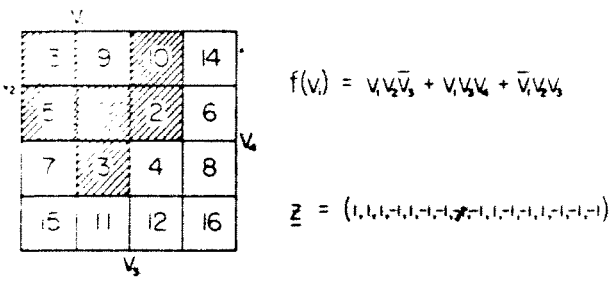

Figure 2

the minterms in a 4-variable Veitch. The octal number representation for the function shown in Figure 2, obtained from the vector $\underline{Z}$, is $011023)_8$ for which the corresponding decimal integer is 4631. The components of $\underline{Z}$ are represented by z_j and are the values of the function.

It now follows that any sequence of (0,1) binary digits, represented by a decimal integer, can be equated to a partially specified Boolean function which in turn corresponds to an incomplete binary pattern on a Veitch diagram. The pattern shown in Figure 2 might be thought of as the letter Y. In using a TLU network to recognize such a binary pattern, one has to remember that the output from these units is a binary quantization of the sum of weighted inputs and is therefore a non-linear function of those inputs. This non-linearity makes it very difficult to solve the general synthesis problem of realizing an arbitrary Boolean function.

Network Configurations

There are three distinct network configurations that are commonly used in attacking this general synthesis problem. In one category we have the Multithreshold Logic Element, as expounded by Haring[4]. In another we see the double layer of Dot Product Units of Nilsson[2] or the Learning Matrices of Steinbuch and Piske[5]. The third category, which is slightly more general, consists of the Feed-Forward Networks of Hopcroft[6] and of Hughes[7]. In addition, networks which include feed-back loops are being examined; at this stage it is difficult to say whether the increased generality and reduction in network size is worth the added complexity.

In a recent paper by Mow and Fu[8] multithreshold threshold element realizations for the 221 NPN equivalence class leaders of the 4-variable Boolean functions were tabulated. (An NPN class leader is one which is invariant with respect to Negation of the function, Permutation of variables and Negation of individual variables.) That table shows that with these elements, which generate parallel separating hyperplanes, the most awkward Boolean function requires five threshold detectors. However, by using the more general feed-forward network, a considerable reduction in elements can be achieved. For example, in our own report (reference 9) a complete tabulation of the 4-variable NPN class leaders has been presented through their 5-variable Self Dual (SD) representatives, of which there are eighty-three. The tabulation is summarized and compared with the multithreshold threshold logic element realizations for the NPN class leaders of the 4-variable Boolean functions in Table 1.

Number of threshold detectors required.	Feed-Forward network.	Multi-threshold threshold element.
1	14	14
2	174	64
3	33	85
4		56
5		2

Table 1: A Comparison of Threshold Element Requirements between Feed-forward Networks & Multi-Threshold Networks.

Although the results in Table 1 are interesting in themselves, it is not the purpose of this paper to make any real comparisons between the two configurations on a basis of generality, complexity or efficiency. Rather, we shall concentrate on the pattern recognition and pattern discovery features of the computer program that generated the results.

Evolutionary Networks

The technique that was used to create our minimal element feed-forward networks was one of numerical exploration. In many respects the approach consisted of bringing together a number of partially correct hypotheses, supported by certain heuristic ideas to form an adaptive (learning) program which performed a task that appeared impractical by other means. These strategies were applied to a general interconnection of four TLUs which formed a feed-forward network, as illustrated in Figure 3.

The output, $x_{N+k,j}$ from the k^{th} element of such a network, is given by

$$x_{N+k,j} = \text{sign}(\underline{W}_k \cdot \underline{X}_j^k),$$

where $\underline{W}_k$ is a $(N + k)$ component weight vector and $\underline{X}_j^k$ represents $\underline{X}_j$ augmented by $(k - 1)$ additional components. In other words,

$$\underline{X}_j^k = (\underline{X}_j, x_{N+1,j}, x_{N+2,j}, \cdots, x_{N+k-1,j})$$

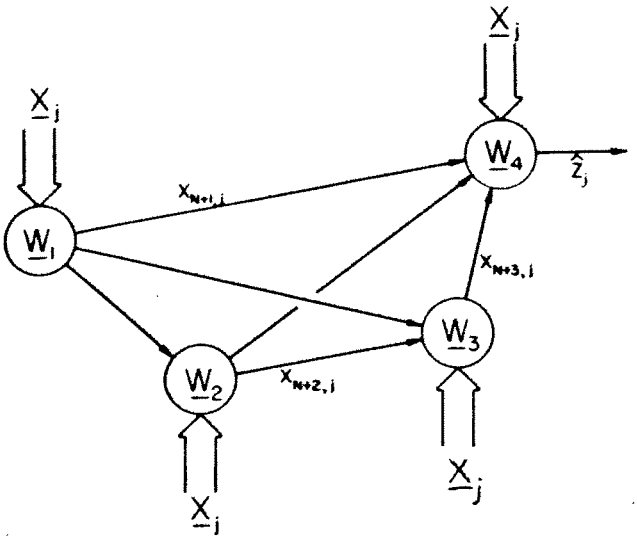

Figure 3

and w_{ik} is the weight associated with the input line x_{ij}, for i = 0, 1, ..., N+k-1. It was not only for its generality but also for its computational convenience that this formulation was chosen.

In contrast to the direct analytical method of solution proposed by Hopcroft[6], which has the potential disadvantage of becoming reduced to an exhaustive search, our approach was quite indirect and experimental. Three basic strategies were followed throughout the evolution of the computer programs for this study:

(1) The whole problem of realizing minimal threshold element solutions for the NPN equivalence class of 4-variable Boolean functions should be treated as a single pattern recognition and learning problem.

(2) The program should gather its experiences on a sequence of increasingly complex functions. Those functions which were not realized satisfactorily on the first attempt were assumed to be more "difficult" or "complex" and were examined more closely. If all else failed they could be added to the current list of problems and re-attempted later. In addition, since the networks for some functions could be used to generate others, this list of problems should be tried several times in an attempt to find "better" solutions.

(3) As far as possible, only algorithms with guaranteed convergence properties should be used.

The formulation of the computer model was designed to be very reminiscent of a common human decision-making process, when confronted with a time-varying environment. Our model tries to predict the nature of its environment, based on experience gathered under similar conditions.

A Heuristic Training Algorithm

Let us now examine the problem of training an L-element network of TLUs to recognize a Boolean function. Although the network size is not initially known, an error-correcting training procedure can still be constructed which will adapt the network until it realizes the function. For instance, whenever an error occurs (that is, the actual output $\hat{z}_j$ is not equal to the desired output z_j), the weights of one of the elements can be modified. If the output is still in error, another element can be similarly treated until the correct response is achieved. Such a scheme leaves open three questions:

(a) What initial values shall be assigned to the weight vectors $\underline{W}_1, \underline{W}_2, \ldots, \underline{W}_L$?

(b) In case of error, which element shall be selected for modification?

(c) What should be done if the training scheme enters a loop?

Since we are using a ($\pm$1) binary system, it is necessary that the component values of any weight vector be either all even or all odd. One convenient unbiased starting point, which allows proper propagation through the network, assigns zeros to the components of $\underline{W}_1$. For the other weight vectors all the components are also zero, except for the last component which has value 2. Later on, when the program is able to use the experience it gains in solving similar problems, an alternative choice of initial conditions is allowed.

Whenever an error occurs during the training period the weights of an element of the network must be selected and modified. Clearly the last element in the network can always be chosen, but equally obviously that scheme would only work on those functions which require a single element. On the other hand choosing an element for modification from near the front of the network influences all those elements which follow it and may cause too great a change in the system. As a reasonable compromise it was decided to examine the prequantized outputs of each element. Beginning with the front element $\underline{W}_1$, select for modification the first one whose prequantized output has smallest absolute value and whose sign is opposite to that desired.

Finally, since the size of the network is measured by the last "non-zero" or "non-initial" element and not by the maximum permissible size of the network, it is possible for the algorithm to enter a loop. Usually this means that the network must be allowed to grow and include an extra element. Certain exceptional cases are discussed in Reference 9.

Network Evolution

So far we have only discussed how one might train a TLU network to recognize an isolated binary pattern. However, since we are interested in the possibilities for machine learning based on experience, it is necessary to make the total task consist of a number of similar sub-problems. Unfortunately, the simple heuristic algorithm suggested in the previous section is not in itself sufficient for our task, which was to find minimal threshold element realizations for the 221 NPN characteristic Boolean functions of four variables.

Our basic approach consists of supplying these problems to our computer model several times. Essential to that model is its long term memory, which gathers statistics about the problems for use when it next sees them. However, the long term memory is not disturbed, even if evolutionary changes are made to the model to improve its performance, unless those changes are so drastic as to force re-initialization. Typical of the non-destructive changes to the program were the incorporation of a redundant element elimination scheme and a "loop breaking" mechanism. The need for these changes became apparent when the solution path for the functions which had been classified as "difficult" (that is, required more than the maximum number of elements allowed) were examined.

The NPN characteristic functions were ordered so that those with fewest true minterms appeared first and no function had more than eight true minterms. This ordering was introduced so that the program could make use of the solutions to these "smaller" problems as starting points for the "bigger" problems which follow, as an alternative to the "zero" starting point mentioned earlier.

In addition to the long term memory, which contained information about the solutions found for all the problems attempted so far, such as the minimum network size and the time taken to find the solution, two short term memories were employed. Both of these memories were used to reduce the storage and searching requirements for the program. One was a very local memory containing detailed information about the last four problems tackled. The other memory was of variable length and was used to reduce the searching time required to detect whether the training algorithm was in a loop. For instance, one might not need to begin checking for a loop in the current problem until MIN adaptions had been made and then not check for a loop longer than MAX iterations, where MIN and MAX are statistical values computed from information stored in the other local memory and are related to values actually required for the preceding problems. The loop data that needs to be checked is quite extensive, consisting mainly of the states (weight vectors) of the network. For simplicity, the loop memory was circular and of length modulo MAX. Furthermore, simple tests were incorporated into the program to make the searching scheme "fail safe".

Aside from the statistics gathered about the nature of the solution, data was collected to show the effectiveness of the various solution methods. In the final version of the program, achieved after three complete attempts to solve all of the sample problems, two distinct extensions of the training algorithms mentioned earlier had established themselves as most satisfactory for further development. The major differences between these algorithms were in the way they re-initialized the network when an element was added after a loop was detected. Further discussion of the operation of these algorithms is to be found in Reference 9. Our interest here is in the scheme used to select the appropriate algorithm for the current problem. The basic mode of operation was as follows.

If an algorithm generated a network containing no more than the predicted number of elements, the next problem was taken. Otherwise the problem was assumed to be "difficult" and another algorithm tried. The gathering of statistics to measure the relative merits of the various algorithms was bound by the premise that such statistics should only be collected on difficult problems and only when one algorithm provides a better solution (that is, one requiring fewer elements) than another. If the success rate of an algorithm dropped too low, it was temporarily eliminated from service. However, at regular intervals and on especially difficult problems, the eliminated algorithms were re-introduced. The mode of operation was that one algorithm was maintained as the primary method of solution until another provided a better solution, whereupon the latter method became the one that was tried first. After a while the two best algorithms dominated the situation, but neither could provide acceptable solutions for all of the problems. However, under the actual test conditions of the final experiment, one algorithm was used about 58 per cent of the time, the other for the remaining 42 per cent. Minimal elements solutions were found for all but four of the problems. It was felt that in all of these cases minimal solutions would have been obtained after a fourth presentation of the complete set of problems. The computer programs for this study were written in

Burroughs Extended Algol and executed on the B-5500 in about forty minutes.

Prediction of Network Size

Critical to the success of this study was the method used to predict an upper bound on the network requirements for a given Boolean function. Before the program had any experience in solving the problems, a 4-element network was assumed. As problems were solved, the average size of the four previous networks was used and after the first complete pass, the number of elements required on the previous solution attempt became an upper bound. For the final program, however, a much better prediction scheme was developed. This scheme was based on the hypothesis that "similar" functions should require the same number of elements. It should be mentioned that two functions are said to be similar if the distributions of their true vertices on an N-dimensional cube represent similar patterns. For example, in terms of threshold element realizations, all functions made up of a single true vertex can be said to belong to the same threshold logic network class, since they are all 1-realizable. Nevertheless, they are in distinct NPN and SD classes. As another example, the three functions whose Veitch diagrams are shown in Figure 4 can be said to belong to the class of three connected minterms isolated from a single minterm, and all require two TLUs.

A special set of characteristic numbers can be computed for all Boolean functions which describe the pattern of the function as displayed on the Veitch map. Let us refer to these numbers as the H-vector, which is an M-tuple having $M=2^N$ components. Each component corresponds to a minterm and is a POSITIVE/NEGATIVE integer which measures the number of FALSE/TRUE minterms adjacent to the original minterm. The numbers shown on the Veitch diagrams of Figure 4 are the component values of the H-vector which correspond to the specified minterms. For characterization purposes the H-vector is ordered from largest to smallest, and under these conditions it becomes invariant with respect to permutation of variables and negation of variables. In addition, negating the whole function reverses the H-vector. Corresponding to the 221 NPN characteristic functions there are 195 distinct H-vectors. Despite this lack of uniqueness, a transformation of the H-vector produces an invaluable categorizer of Boolean functions in terms of its TLU network requirements. That transformation is designed to eliminate the effect of the number of variables in the function and is supplied by a new vector, referred to as the T-vector, whose components are given by

$$t_j = (N-|h_j|)\,\mathrm{sign}(h_j), \quad \text{for } j=1, 2, \ldots, M=2^N,$$

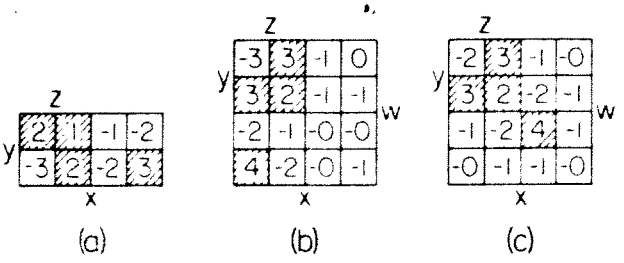

Figure 4

where N is the number of variables in the function. For the three functions shown in Figure 4 the T-vectors are

Ta = (0,1,1,2;-2,-1,-1,-0)

Tb = (0,1,1,2;-4,-4,-4,-4,-3,-3,-3,-3,-3,-2,-2,-1)

Tc = (0,1,1,2;-4,-4,-4,-3,-3,-3,-3,-3,-3,-2,-2,-2)

It was the positive part of such T-vectors as these which were used to help in the prediction of the network requirements for our 221 problems. It was hypothesized that any functions that were similar in the sense that they had equal positive parts of their respective T-vectors would require the same size of network. This kind of prediction was made 117 times of which 85% subsequently proved to be correct.

It should be emphasized that the set of characteristic numbers developed through the T-vector only provided a one way prediction of the form that functions with the same characteristic numbers tend to require the same number of threshold elements in their realization network. Even though not perfect, the T-vector was a deliberate attempt to classify Boolean functions through their threshold element requirements, rather than through NPN or SD classifications. In many respects the development of the T-vector could be regarded as an example of computer aided pattern discovery.

Conclusions

The successful technique in this study consisted of evolving an interactive adaptive program, one which was capable of making its own decisions about the actual solution method to use for a particular problem. The choice of decision was based on the program's experience in the successful and unsuccessful handling of similar problems. Such programs are often of questionable efficiency. Nevertheless they do allow a number of inelegant algorithms, partially correct hypothesis and heuristic ideas to be brought together so that the whole can perform a task impossible for any subset of its component parts.

In the field of TLU synthesis one of the aims is to classify Boolean functions in terms of their network size. Along these lines, the H and T-vectors generated by this research may be helpful since they categorize functions as patterns. These patterns, or distributions of true vertices on a N-dimensional cube, seem to bear some relationship to the network requirements of the function.

Acknowledgment

The research for this paper was jointly sponsored by the National Science Foundation under Grant GK-680, the Air Force under Grant AF AFOSR-468-65 and through a generous grant of computer time from the University of Washington Computer Research Center. The programs for the study were written in Burroughs Extended ALGOL for execution on a B-5500.

References

M. Phister, Logical Design of Digital Computers, Wiley (1962)

N. J. Nilsson, Learning Machines, McGraw Hill (196)

Y-C Ho and R. L. Kashyap, "A Class of Iterative Procedures for Linear Inequality", SIAM Journal of Control, pp. 112-115, Feb. 1966.

D. R. Haring, "Multithreshold threshold elements", IEEE Trans. on Electronic Computers, Vol. EC-15, pp. 45-65, Feb. 1966.

K. Steinbuch and V. A. W. Piske, "Learning Matrices and their Applications", IEEE Trans. on Electronic Computers, pp. 846-862, December 1963.

J. E. Hopcroft and R. L. Mattson, "Synthesis of Minimal Tnreshold Logic Networks", IEEE Trans. on Electronic Computers, Vol. EC-14, pp. 552-560, Aug. 1965.

G. F. Hughes, "Feed-forward Threshold Logic Nets for Digital Switching and Pattern Recognition", IEEE Trans. on Electronic Computers, Vol. EC-16, pp. 463-472, Aug. 1967.

C-W Mow and K-S Fu, "An Approach for the Realization of Multithreshold Threshold Elements", IEEE Trans. on Computers, Vol. C-17, pp. 32-46, Jan. 1968.

T. A. Marsland, "An Adaptive Computing System for the Synthesis of Threshold Logic Networks", Air Force Report for Contract AFOSR-468-65, August 1967, D. L. Jonnson, Principal Investigator. Available through Defence Documentation Center Number AD 663446 or the author.