MaxSet: An Algorithm for Finding a Good Approximation for the Largest Linearly Separable Set

Leonardo Franco, José Luis Subirats, and José M. Jerez

Departamento de Lenguajes y Ciencias de la Computación,
Universidad de Málaga,
Campus de Teatinos S/N, 29071 Málaga, Spain
{lfranco, jlsubirats, jja} @lcc.uma.es

Abstract. Finding the largest linearly separable set of examples for a given Boolean function is a NP-hard problem, that is relevant to neural network learning algorithms and to several problems that can be formulated as the minimization of a set of inequalities. We propose in this work a new algorithm that is based on finding a unate subset of the input examples, with which then train a perceptron to find an approximation for the largest linearly separable subset. The results from the new algorithm are compared to those obtained by the application of the Pocket learning algorithm directly with the whole set of inputs, and show a clear improvement in the size of the linearly separable subset obtained, using a large set of benchmark functions.

1 Introduction

We propose in this work a new algorithm for finding a linearly separable (LS) subset of examples for a non-linearly separable problem. Related versions of this problem appears in different contexts as machine learning, computational geometry, operation research, etc. [1,2,3,4]. In particular, the problem is very relevant in the field of neural networks given that individual neurons can only compute linearly separable functions. Many constructive algorithms for the design of neural architectures are based on algorithms to find a maximum cardinality set of linearly separable examples [4]. Within the neural networks community, the very well known perceptron algorithm [5], that works well for finding the synaptic weights for a linearly separable problem, has been adapted several times in order to be applied to the case of non-linearly separable problems [6,7]. Among the different modifications, the most popular is the Pocket algorithm introduced by Gallant [6]. The problem of finding the largest LS subset is NP-hard and thus different approximate solutions and heuristics have been proposed to deal with it [3,4,8]. In this paper we introduce an algorithm for finding an approximation to the largest linearly separable subset that is based on finding a set of unate examples. The algorithm then use this subset to train a perceptron using the Pocket algorithm. The new and original contribution of the algorithm

J. Marques de Sá et al. (Eds.): ICANN, Part I, LNCS 4668, pp. 648–656, 2007.
© Springer-Verlag Berlin Heidelberg 2007

consists in a method for finding an unate approximation of the input function to speed up and improve the process of obtaining a LS set, working as a kind of preprocessing step before applying the Pocket algorithm,. In agreement with this previous idea, it has been shown by Jacob & Mishchenko [9] that obtaining a unate decomposition of a function improves its synthesis procedure in logic design. We have compared the results of the MaxSet algorithm against the most popular of the used algorithms, the Pocket algorithm, trained with the whole sets of examples.

2 Mathematical Preliminaries

A Boolean function of n variables is defined as a mapping $f : \{0,1\}^n \rightarrow \{0,1\}$. A Boolean function is completely specified by the output of the function, $f(x)$, in response to each of the 2^n input examples.

A Boolean function is unate if all its variables are unate. A variable is unate if it is only positive or negative unate but not both. A variable is positive or negative unate according to the change on the output of the function that a change in the variable produces, i.e., if a variable x_i is positive unate then $f(x_0, \ldots, x_i = 0, \ldots, x_{n-1}) \leq f(x_0, \ldots, x_i = 1, \ldots, x_{n-1})$ for all x_j with $j \neq i$ (conversely, for a negative unate variable). The influence of a variable x_i is defined as the number of inputs vectors $x \in \{0,1\}^n$ such that when the i^{th} component is negated (flipped from 0 to 1 or viceversa) the output of the function changes its value. Unate variables have positive or negative influences but not both, case in which are named binate.

It has been demonstrated that all threshold functions are unate but the converse is not true (for example, the function $F(x_0, x_1, x_2, x_3) = x_0\ x_1 + x_2\ x_3$ is unate but it is not threshold).

3 The MaxSet Algorithm

The general idea of the algorithm is to construct an unate approximation of the input function that is as similar as possible to the original function. In order to construct this approximation the algorithm selects a variable at a time, adding to the unate set pair of examples those that produce a positive (or negative) influence. The selection of the variable and the sign of the influence is decided according to the largest number of pairs available at that time with the same influence, in order to maximize the cardinality of the selected set. The optimal solution for obtaining the unate approximation, consists in maximizing the whole set of variables simultaneously, and it is a NP-hard multiple optimization problem. Our method, instead, select variables each at a time, searching for the largest possible subset at each step.

For a clear explanation of the algorithm we will follow a flow diagram shown in Fig. 2 and an example of the application of the algorithm on a non-LS function of 4 variables. The Boolean function selected for the example is shown in the

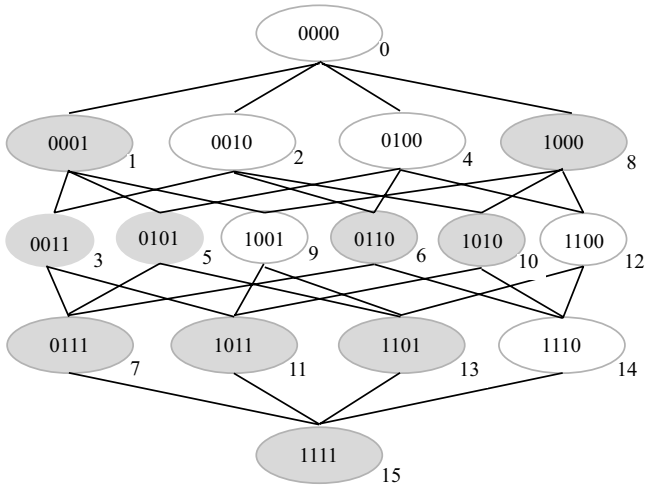

Fig. 1. The partially ordered set (Poset) of the Boolean function used as an example in the text to illustrate the application of the algorithm. The input bits of the examples are shown inside the ovals and the corresponding outputs are indicated by the color of the oval (white corresponds to 0 and grey indicates 1). The decimal value of the inputs is indicated outside the ovals, as it is used in the text. Links between examples connect pair of nearest neighboring examples, whose input bits are at Hamming distance of 1.

partially ordered set (Poset) depicted in Figure 1. The Poset shows the examples ordered in levels according to the weight of the input (number of 1's) where links between nearest neighboring examples are also shown. In the figure (Fig. 1), the output value of each of the examples is indicated by the color of the boxes (white:0, grey:1). It is also shown below each of the examples, the decimal order of the examples used to refer to the examples.

The first step of the algorithm consists in allocating all pairs of nearest neighboring examples (pairs of examples at Hamming distance 1) to one of the 4 variables of the function (the one in which the examples of the pair differ) and to one of the three possible sets : positive, negative and neutral influence sets. The pair is included in a given set according to the influence of the variable in which the pair differs. For example, the pair of examples with decimal indexes 0 and 1 (inputs 0000 and 0001), with output value 0 and 1 respectively will be included in the positive set corresponding to the the last bit variable, x_0. This is because the pair of examples defines the variable x_0 as having positive influence, because a change in the variable from 0 to 1 makes the output of the example to change in an increasing way (positive).

As mentioned, the algorithm starts with the allocation of all possible pairs of neighboring examples to a table in which every pair is assigned to a variable and to a category out of the three possible ones: positive, negative and neutral, indicated by $(+, -, N)$. The allocation of the pairs of neighboring examples for the case of the function that we are using to illustrate the method is shown in Table

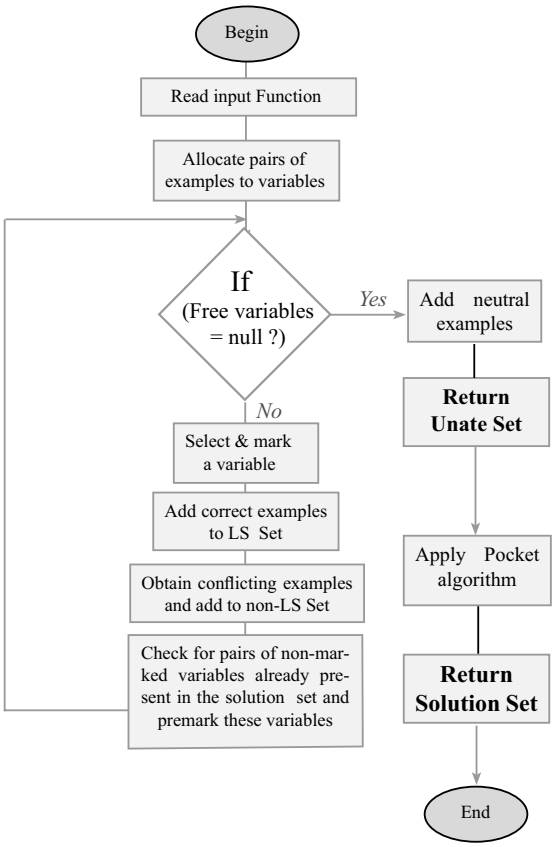

Fig. 2. Flow diagram of the MaxSet algorithm introduced in this work to find an approximation to the largest LS subset of a non-LS input function

1. The function is defined by the truth vector (0,1,0,1,0,1,1,1,1,0,1,1,0,1,0,1), that contains the output values of the function in response to the 16 inputs arranged in decimal order. The table has 4 columns corresponding to the four variables of the function and three rows for the three types of influences.

After all the pairs have been allocated into the table, the algorithm continues by selecting a set of examples belonging to a positive or negative category with the largest cardinality. The neutral sets are only considered at the end of the algorithm after all variables have been defined as positive or negative. For the function that we are considering, the first selected set is the examples with positive influence corresponding to the variable x_0, as this is the largest set containing 10 examples. The selected examples are then included in the LS set. For the function of the example, this step means that examples 0,1,2,3,4,5,12,13,14 and 15 are included in the LS set. After a set of examples have been selected, the algorithm normally checks if there are possible conflicting examples, but this

checking step is only applied after a second variable has been selected and the process is described below. Thus, the algorithm continues by selecting a second variable in the same way as it was done for the first one, selecting the largest subset corresponding to a negative or positive non-selected variable. For the example under analysis, the new selected subset is the positive examples corresponding to variable x_1 that will add to the LS set a total of 3 **new** examples (the examples 6,9 and 11, as example 4 is already included). After the second subset (and variable) have been selected, a checking procedure is applied in order to avoid possible future conflicts. The checking procedure analyzes the non-selected pairs of example with opposite influence of the selected sets that have an example included in the LS set. For the Boolean function of the example, the variables x_0 and x_1 have been already selected with positive influence, implying that we should look for the negative sets of these two variables for probable conflicting examples. The way to look for conflicts is to search for pairs included in these non selected sets of variables x_0 and x_1 in which one of the examples of the pair belongs to the LS set. For example, the pair $8-9$ with negative influence on variable x_0 contains the example number 9 that has been already included in the LS set. The checking procedure marks example 8 as a forbidden example and includes it in the Non-LS set (avoiding its future selection), because an eventual selection of example 8 will imply that the variable x_0 would have both positive and negative influence, and this is not possible for a unate or a LS set of examples. The algorithm also checks for pairs of examples already included in the LS set, that belong to non-marked variables. In the example, the pairs 3-6 and 9-13 belonging to the variable x_2 with positive influence are already included in the solution set, and also the pairs 1-9 and 6-14 corresponding to the variable x_3 with negative influence. On the contrary, the example 8 that has been marked as a conflictive example, makes the algorithm to eliminate the pairs 8-12 and 0-8 in the negative x_2 variable and in the positive x_3 variable respectively. The main loop of the algorithm is applied again, but this time can only consider the pre-marked variables, and thus the pre-marked positive part of variable x_2 is included to the LS set. No new examples are added to the LS set as the examples of the pairs with positive influence for variable x_2 are already part of it. Any of the variables x_2 or x_3 could have been selected as they contain the same number of possible examples, but the algorithm is set to select the first one, in this case the x_2. The checking procedure now marks example 10 as forbidden example and thus the pair 2-10 is deleted from the positive part of variable x_3. Lastly, the algorithm marks the only left variable, x_3 as negative as it has been already pre-marked, and the main loop of the procedure finishes. The unate set contains the examples $0, 1, 2, 3, 4, 5, 6, 9, 12, 13, 14, 15$, while the forbidden examples 8 and 10 belong to the non-LS set. Examples 7 and 11 are not yet selected, meaning that we can include them in the unate set, as they only involve neutral influence pairs, and thus the final solution set includes the 14 examples $0, 1, 2, 3, 4, 5, 6, 7, 9, 11, 12, 13, 14, 15$.

The procedure then finishes by applying the pocket algorithm with the selected unate set that as a result will give a linearly separable set. If the unate

test set was, indeed, linearly separable the pocket algorithm will then indicate so but if the unate set happened to be unate but not linearly separable the pocket algorithm will select only some examples of this candidate set. The procedure ends by giving as output the LS separable set found by the pocket algorithm. If a LS function is needed instead of a LS subset, all that is needed is just to test the examples not already included in the LS set to find their outputs. In the case of the function of the example, the unate set found is also linearly separable and thus the pocket algorithm is able to learn it all.

Table 1. An example of the application of the MaxSet algorithm to a non-LS Boolean function of 4 variables. All the 32 possible pairs of neighboring examples are included in the table. (See text for details).

	x_0	x_1	x_2	x_3
$+$	$0-1$ $2-3$ $4-5$ $12-13$ $14-15$	$4-6$ $9-11$	$3-6$ $9-13$	$0-8$ $2-10$
$-$	$8-9$		$8-12$ $10-14$	$1-9$ $6-14$
N	$6-7$ $10-11$	$0-2$ $1-3$ $5-7$ $8-10$ $12-14$ $13-15$	$0-4$ $1-5$ $4-7$ $11-15$	$3-11$ $4-12$ $5-13$ $7-15$

4 Testing the Algorithm on a Set of Benchmark Functions

The algorithm described above was implemented in $C\#$ and tested on different sets of Boolean function. The efficiency of the algorithm to find large LS sets and the computational time required for the implementation was compared to the results obtained using the pocket algorithm applied to the original set of examples. An initial test was carried out with all the 65536 Boolean functions of 4 variables with the aim of checking that the implementations of the algorithms used in the test was working properly, and also to see how the new algorithms perform with LS functions. The test confirmed that both algorithms found the correct solution (i.e., the whole set of input examples was LS) for the 1881 LS functions of 4 variables that exist in this case. The average size of the LS subset across all the functions of 4 variables was 12.72 (79.5 % of the 16 examples) for the new algorithm, while 10.79 (67.4 %) examples resulted from the application of the Pocket algorithm (16 examples constitute the whole set of inputs for functions of 4 variables). The Pocket algorithm was implemented for this case setting a maximum of 200 iterations each time a new set of weights was found.

Table 2. Results of the implementation of the MaxSet and Pocket algorithms on a set of 54 non-LS Boolean functions used in logic synthesis (See text for details)

Function	Inputs	Fraction LS MaxSet	CPU Time (Seconds)	Fraction LS Pocket	CPU Time (Seconds)
cm82af	5	0.75	0	0.50	1
cm82ag	5	0.75	0	0.50	1
parity5	5	0.50	0	0.50	1
z4ml25	7	0.75	0	0.50	1
z4ml26	7	0.75	0	0.52	1
z4ml27	7	0.75	0	0.50	1
f51m44	8	0.80	0	0.50	1
f51m45	8	0.80	0	0.54	1
f51m46	8	0.80	0	0.53	1
f51m47	8	0.78	0	0.47	1
f51m48	8	0.81	0	0.53	1
f51m49	8	0.88	0	0.56	1
f51m50	8	0.75	0	0.50	1
9symml52	9	0.82	0	0.74	1
alu2k	10	0.53	0	0.54	1
alu2l	10	0.51	0	0.52	1
alu2m	10	0.75	0	0.50	1
alu2p	10	0.75	0	0.72	1
x2l	10	0.88	0	0.79	1
x2p	10	0.94	4	0.80	4
x2q	10	0.83	1	0.82	1
cm85am	11	0.65	3	0.96	3
cm152al	11	0.55	1	0.58	1
cm151am	12	0.79	5	0.79	5
cm151an	12	0.75	4	0.77	4
alu4o	14	0.50	59	0.46	60
alu4p	14	0.56	70	0.51	71
alu4r	14	0.53	64	0.50	65
alu4s	14	0.75	42	0.51	43
alu4u	14	0.78	90	0.73	91
alu4v	14	0.38	85	0.94	87
cm162ao	14	0.92	56	0.84	57
cm162ap	14	0.92	69	0.85	70
cm162aq	14	0.93	63	0.86	64
cm162ar	14	0.93	87	0.87	89
cup	14	0.81	41	0.88	43
cuq	14	0.81	41	0.87	43
cuv	14	0.94	380	0.93	382
cux	14	0.83	118	0.96	120
cm163aq	16	0.92	757	0.83	773
cm163ar	16	0.93	774	0.79	793
cm163as	16	0.93	793	0.86	811
cm163at	16	0.94	931	0.86	948
parityq	16	0.50	983	0.50	1000
pm1a0	16	0.94	741	0.88	758
pm1c0	16	0.97	788	0.94	806
tcona0	17	0.75	5645	0.76	5712
tconb0	17	0.79	7441	0.75	7513
tconc0	17	0.76	5294	0.75	5368
tcond0	17	0.75	6212	0.75	6287
tcone0	17	0.72	5256	0.75	5331
tconf0	17	0.75	7476	0.75	7548
tcong0	17	0.76	5877	0.75	5952
tconh0	17	0.75	5952	0.75	6015
Average		0.77	1041	0.70	1054

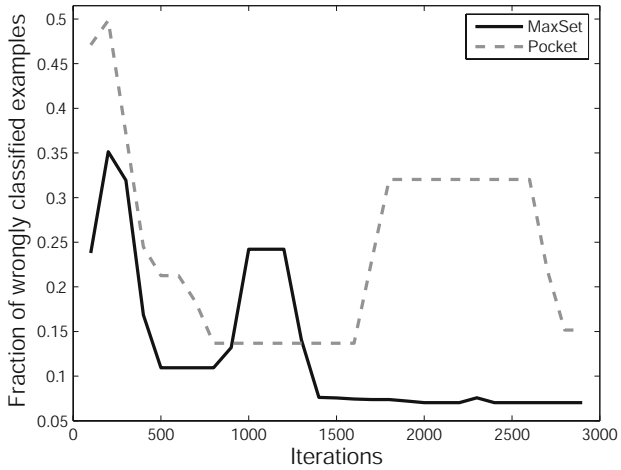

Fig. 3. The fraction of incorrect classified examples vs. the number of iterations for the function cm163aq using the MaxSet and Pocket algorithms

A further test was carried out using a set of 54 benchmarks functions used in logic synthesis. The functions considered where all non-linearly separable, and have a number of input variables ranging from 5 to 17. In Table 2, we report the results for the 54 functions indicating in the first column the name of the function, followed by the number of input variables. The rest of the columns shows the results for the size of the linearly separable set found, indicated as a fraction of the whole set of examples, and the CPU time in seconds used for the calculations for the case of the new algorithm and for the application of the Pocket algorithm directly with the whole set of examples. In 40 out of the 54 cases considered, the MaxSet algorithm found the largest LS set, while in 11 cases the best results were obtained with the original dataset using the Pocket algorithm (In the remaining 3 functions the results were the same.) On average the application of the MaxSet algorithm leads to LS sets that cover on average 76.7% of the total set of examples, while the result from the application of the Pocket algorithm was 70%, this means a 10.00% improvement on the size of the LS subset found (equivalent to a 23% reduction on the average fraction of errors). The previous comparisons were done by selecting similar computational times for both algorithms (1041 and 1054 seconds on average per function).

We also shown in Fig. 3 an example of the learning process for both algorithms (MaxSet and Pocket) for the case of the 16 input variables Boolean function cm163aq. It can be shown from the graph, that the MaxSet algorithm performs better both in terms of size of the obtained set and in terms of the number of iterations needed. Fig. 3 shows the result of the fraction of wrongly classified examples as a function of the number of iterations, where the values have been averaged across 100 repetitions of the learning process to eliminate random fluctuations. Moreover, it can be seen that the dynamic of the learning process for the case of the MaxSet has a similar shape to the one obtained

from the Pocket algorithm (trained with the whole sets of examples), but it is a re-scaled version of it, where both the fraction of errors and the number of iterations are reduced approximately by half.

5 Discussion

We introduced in this work a new algorithm for the problem of finding a large subset of LS examples from a non-LS set. The new procedure works by finding first a unate set of examples (the new contribution of this work), to then train a perceptron through the Pocket algorithm with this unate set. The results over a large set of benchmark functions show that for similar computational times the MaxSet algorithm outperforms in most cases the standard Pocket algorithm and lead to an average 11.94% improvement on the size of the LS subset found (equivalent to a 24% reduction in the fraction of errors made). An important aspect of the introduced MaxSet algorithm, regarding the obtention of an unate subset, is that this part of the method can in principle be combined with any other alternative method in order to obtain the final linearly separable solution set.

The extension of the method to partially defined Boolean functions is not straightforward but it is being developed under similar lines and it will be reported elsewhere.

References

1. Johnson, D.S., Preparata, F.P.: The densest hemisphere problem. Theoretical Computer Science 6, 93–107 (1978)
2. Greer, R.: Trees and hills: Methodology for maximizing functions of systems of linear relations. Annals Discrete Mathematics 22 (1984)
3. Amaldi, E., Pfetsch, M.E., Trotter, L.E.: On the maximum feasible subsystem problem, IISs and IIS-hypergraphs. Mathematical Programming 95, 533–554 (2003)
4. Elizondo, D.: Searching for linearly separable subsets using the class of linear separability method. In: Proceedings of the IEEE International Joint Conference on Neural Networks, pp. 955–960. IEEE Computer Society Press, Los Alamitos (2004)
5. Rosenblatt, F.: Principles of Neurodynamics. Spartan Books, New York (1962)
6. Gallant, S.I.: Perceptron-based learning algorithms. IEEE Trans. on Neural Networks 1, 179–192 (1990)
7. Frean, M.: Thermal perceptron learning rule. Neural Computation 4, 946–957 (1992)
8. Marchand, M., Golea, M.: An Approximation Algorithm to Find the Largest Linearly Separable Subset of Training Examples. In: Proceedings of the 1993 Annual Meeting of the International Neural Network Society, vol. 3, pp. 556–559. Erlbaum Associates, Hillsdale, NJ (1993)
9. Jacob, J., Mischenko, A.: Unate Decomposition of Boolean Functions. In: Proceedings of the International Workshop on Logic and Synthesis, California (2001)