An Error Reducing Approach to Machine Learning Using Multi-Valued Functional Decomposition

Craig M. Files Marek A. Perkowski
Department of Electrical Engineering
Portland State University
Portland, OR 97207-0751, USA
email: cfiles@ee.pdx.edu

Abstract

This paper considers minimization of incompletely specified multi-valued functions using functional decomposition. While functional decomposition was originally created for the minimization of logic circuits, this paper uses the decomposition process for both machine learning and logic synthesis of multi-valued functions. As it turns out, the minimization of logic circuits can be used in the concept of "learning" in machine learning, by reducing the complexity of a given data set. A main difference is that machine learning problems normally have a large number of output don't cares. Thus, the decomposition technique presented in this paper is focused on functions with a large number of don't cares. There have been several papers that have discussed the topic of using multi-valued functional decomposition for functions with a large number of don't cares. The novelty brought with this paper is that the proposed method is structured to reduce the resulting "error" of the functional decomposer, where "error" is a measure of how well a machine learning algorithm approximates the actual, or true function.

1. Introduction

In the 1950s, Ashenhurst presented a Boolean multi-level logic minimization method called functional decomposition [3]. From the 1950s until the late 1980s, not much was done in the area of creating efficient programs for functional decomposition primarily because of the large computational procedures that are required in the decomposition process. In the late 1980s, functional decomposition was re-introduced as an application to the synthesis of Field Programmable Gate Arrays (FPGAs) [10].

Since then decomposition has been applied to many aspects of Boolean and multi-valued logic synthesis. Several papers written on decomposition of multi-valued networks have shown an interest in using their decomposers in the field of machine learning [7, 16, 14, 13]. Each of these papers had a slight twist to the decomposition process and the type of data that they were able to handle. The motivation was always the same: given some measure of complexity, find the solution with the minimum complexity. This is analogous to reducing the size, structure, number of gates, or number of terms in a logic circuit. The first ideas to apply decomposition to machine learning appeared in [12] and the first successful machine learning programs based on logic synthesis were reported in [15].

The main goal of using a multi-valued functional decomposer is to reduce the complexity (smaller circuit size, simpler description) of a given problem. This is analogous to the assumption in machine learning that the simplest formula derived from the data is the best formula. This is thought of as a natural constraint given that our own brains think in terms of reduced complexity. For example, a child might be given a "connect the dot problem", and from dot to dot, the child will draw straight lines. A straight line in this instance is the simplest way of connecting the dots, and very rarely will the child use lines of large complexities (e.g., sinusoids, circles, crossing over the same line).

This paper explores multi-valued functional decomposition as a logic synthesis method and as a machine learning tool. The decomposition algorithms for these two methods are similar, but there are some significant differences. The biggest difference is that most circuit-related multi-valued logic problems are nearly completely specified, while functions in machine learning tend to be 99.9% unspecified in their respective learning domains.

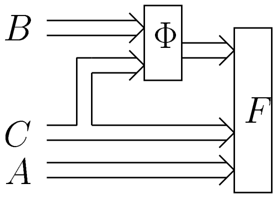
The paper is structured as follows. In Section two, a background of machine learning is given. In Section three, multi-valued decomposition is discussed. In Section four, the concept of building an *error reducing* multi-valued decomposer is presented. In Section five, an algorithm is presented that is based on the concepts of low complexity. In Section six, results of the algorithm are given. Section seven concludes the paper by remarking on the results of the algorithm.

2. Machine Learning

The similarities between machine learning and logic synthesis are based on minimizing circuit designs for size, number of gates, or product terms, and is equivalent to the machine learning concept of reducing complexity. In machine learning the idea is to find patterns in the data, such that the data can be partitioned into smaller concepts (data blocks), which correspond to the sub-blocks of the decomposition. The principle of using decomposition in machine learning is to reduce a given function specified by a set of care minterms (*samples* or *examples*) to a composition of smaller functions (concepts). The result is a set of expressions that describe suitable *intermediate concepts*. Each of these *intermediate concepts* can then be decomposed further, leading to expressions that form a more comprehensible description of the *learned* concepts. The advantage of using decomposition to obtain useful *intermediate concepts* is that it leads to a result specified as a hierarchy of compositions. This produces the description of the original function as a hierarchy of sub-functions and variables, which leads to *learning* that is faster, involves smaller error and gives better explanation of the *learned* concepts.

The central idea in machine learning is to *learn* or gain information from a given data set. This is done by using the care terms (*examples*) in the data to set the values that are not given in the data (output *don't cares*). The objective is to set the *don't cares* to values that follow the same patterns as the care terms. In terms of logic synthesis, this setting of *don't cares* is done by creating a network of multi-valued input, multi-valued output blocks by decomposing the original function into multi-leveled blocks. A machine learning algorithm is evaluated on it effectiveness of learning by how it reduces the *error* of the resulting network. *Error* is how well the algorithm in question set *don't care* terms to care terms. This evaluation is done by selecting a *training set*, which is a random sampling of the original *test* function. The result of learning the *training set* is then compared to the *test* function. If the resulting expression has a high error then it does not approximate the *test* function well and, thus, is not a useful way of describing the function.

This can be a very difficult task given that practical databases that machine learning algorithms are used on are usually very large. To help handle this difficulty, the assumption of Occam's Razor and the introduction of explicit domain knowledge are used to reduce the search space [4]. Thus, machine learning is not an exact methodology for solving certain problems, but is an attempt at *learning* based on heuristics and probabilities. The hope is that these *learning* algorithms perform better than using chance, i.e, a resulting *error* that is less than 50%.



$$f{=}F(\Phi(B,C),A,C)$$

Figure 1. Basic Decomposition

To help solve some of the problems in machine learning, methods have been proposed to involve logic synthesis. This includes creating complexity measures that are general enough to be used in minimizing both circuits and concepts.

3. Multi-Valued Functional Decomposition

This section presents the basic principles of the decomposition of multi-valued functions. Note that the decomposition of multi-valued functions is just an extension of the decomposition of Boolean functions. See [3, 6, 18] for more information on Boolean based decomposition.

Definition 1 Given a multiple-valued variable x_i, the set of values that x_i may assume is $C_j = \{0, 1, \ldots, c_{j-1}\}$. Then an n-input, m-output, multi-valued function is defined as the mapping: $f(x_0, x_1, \ldots, x_{n-1}) = C_0 \times C_1 \times \ldots \times C_{n-1} \rightarrow D_0 \times D_1 \times \ldots \times D_{m-1}$, where D_j=$\{0, 1, \ldots, (p-1), -\}$, with p equal to the number of output values represented in the p-valued logic and "-" represents a *don't care* value.

Definition 2 A function $f(x_0, x_1, \ldots, x_{n-1})$ is decomposable under *bound set* $\{x_0, \ldots, x_{i-1}\}$ and *free set* $\{x_{i-l}, \ldots, x_{n-1}\}$, $0 < i < n, 0 \leq l$ if and only if f can be represented as the composite function $F(G_0(x_0, \ldots, x_{i-1}), \ldots, G_{j-1}(x_0, \ldots, x_{i-1}), x_{i-l}, \ldots, x_{n-1})$, where $0 < j < i-l$. If l equals 0 then f is said to be *disjunctively decomposable*, otherwise, it is known as *non-disjunctively decomposable* [6, 18].

The principle idea of the decomposition using the notations from Definition 2 is shown in Figure 1. Given that a function is decomposable under a given bound set, the function may be separated into two newly created functions. This is known as a simple decomposition because one decomposition was done on the function. To fully decompose the function, an iterative process is used. First the function is decomposed, then the sub-functions that are created from the initial decomposition are decomposed. This iterative process continues until a given function cannot be decomposed further under given complexity measures.

Note, that for an n-input function, the number of simple *disjunctive decompositions* is 2^n, while the number of simple *non-disjunctive decompositions* is 3^n. Thus, evaluating all possible partitions is an NP-complete problem when trying to fully decompose a function. Because of the exponential size of simple *non-disjunctive decomposition*, most heuristic methods only attempt at finding *disjunctive decompositions*. This reduces the search complexities of finding partitions in decomposition, but this still does not mean that the problem of finding *non-disjunctive decompositions* becomes trivial, since it is still an exponential search problem.

Definition 3 Given a k-valued, completely specified function f, with a *bound set* B, and *free set* A, then for the partition $A|B$, a *partition matrix representation of* f is defined as a rectangular array, where the *columns* correspond to the variables in the *bound set*, and the *rows* correspond to the variables in the *free set*.

Using Definition 2 and 3 the following is found:
1. The array has $k^{|B|}$ columns and $k^{|A|}$ rows.

2. Given a k-valued function f, with a bound set B, then the corresponding partition matrix has l distinct columns, where l is known as the *Column Multiplicity* of a partition.

3. The *Column Multiplicity* for the function can be reduced if the function is incompletely specified, by finding columns that are *compatible* and combining the two columns by setting *don't care* values. By compatible, for every row, the possible output value-sets of the first column (a number or a *don't care*) intersect the non-empty sets of the corresponding output value-sets of the second column.

4. Thus, to represent f as a composite function in the form: $f = F(G_0(), \ldots, G_{j-1}(), x_i, \ldots, x_{n-1})$, where each G function has inputs $(x_0, \ldots, x_{i-1})$ then $j = \lceil log_k l \rceil$, G functions are needed.

The concept of decomposition is fairly general, and because of this generality it can be applied to any type of logic structures or elements, and takes into account various constraints and requirements. Of course, this might be a difficult task, but none-the-less, it is possible. Further, multi-valued network decomposition has the advantage that it is not based on any pre-specified set of operators or gates.

4. Reducing Error Through Partition Selection

The biggest difference between logic synthesis and machine learning is the difference in the number of care terms in logic synthesis versus the number of care terms in machine learning problems. There is another factor to this that must be noted. The *don't cares* in machine learning problems should actually be thought of as *don't knows*. By *don't knows* we mean that the value is not known and that caution must be used in setting a *don't know* to a value.

With the concept of a *don't know* in mind, this section presents a methodology for selecting partitions for machine learning based decompositions. The formulation for finding partitions and evaluating is based on *don't knows* and the complexity issues found in trying to decompose functions with many variables.

Observation 1 From the Section 3, given that the function is completely specified, the only way that two columns can be compatible is if for every row, the output value of the first column is equal to the output value of the second column (in the same row). Obviously, if the function is p-valued, completely specified, and each value is equally likely, then for every row the probability of matching one column to another is $\frac{1}{p}$.

From Observation 1, for a column with B rows, the probability of finding a pair of *compatible* columns is $\left(\frac{1}{p}\right)^B = \frac{1}{p^B}$. The result is that the more rows in a column, the less likely it is to find a *compatibility* between two columns. The problem with Observation 1 is that it doesn't take *don't cares* into consideration and the fact that machine learning data sets are 99% unspecified.

Observation 2 Assume the probability of having a *don't care* is $P(X) = 0.99$, then the probability of having any other value in the function is $P(care) = \frac{1-0.99}{p} = \frac{0.01}{p}$, given that the function is p-valued and each value is equally likely. The only way to have an incompatibility between two columns occurs only when, for a given row, the first column's value is some value and the second column's value is a value that is not equal to the first column's value. Thus, the probability of having an incompatibility between two columns is $0.01\frac{p-1}{p}$.

It results from this observation that evaluating small bound sets (small number of columns and a large number of rows in a partition table) should result in a larger probability of incompatible columns. The premise of this paper is that by increasing the probability of incompatible columns we have decreased the amount of *error* in the resulting network. This premise is true because the possibility of combining a *don't care* with an incorrect care term is reduced. This is based on the probability of combining two columns is much smaller, and thus, the probability of combining a *don't care* with a value is also smaller.

For example, given a 3-valued function with 100 inputs (each input is also 3-valued), if the bound set has 2 variables then there are 98^3 rows in partition table. Whereas, if the bound set has 50-variables then there are 50^3 rows in the partition table. From Observation 2, the probability of having incompatible columns is $P(incomp)$ and the probability of having compatible columns is $1 - P(incomp)$. In comparison, the probability of finding compatible columns with 2-variables, $(1-P(incomp))^{98^3}$, is much smaller than the probability of finding compatible comlumns with 50-variables, $(1 - P(incomp))^{50^3}$. The results show that it is highly unlikely that two columns will be found to be compatible in the case of 2-variable bound sets. The only way that they will be compatible is if there is some pattern in the data that allows the combination of the two columns. As for the case of the 50-variable bound set, the probability seems like a small enough , but note that there are 50^3 columns for this partition.

Another complexity issue that comes up is determining column compatibility. One of the most promoted methods for finding column compatibilities with many *don't cares* is the use of graph coloring. Since graph coloring is an NP-complete problem (exponential in complexity) itself, then the more columns then the longer the computation time. Thus, it is reasonable to assume that small bound sets with a small number of columns will have a much faster run time than medium-sized (half the variables in the bound set, the other half in the free set) bound sets. In the case of medium or large bound sets, the probability of two columns being incompatible is much smaller than in the case of small bound sets. Hence, the number of compatible columns can be very large. In fact, because there are so many possible coloring in the graph coloring algorithm, selecting compatible columns becomes a random process. Thus, the combination of columns is done randomly with no basis on probabilities. This also shows that having bound sets that are medium or large will result in greater *error*.

What does this mean in terms of complexity. From the Section 3, we know that for an n-input function there are 2^n partitions that can be evaluated or O(2^n). If we restrict the size of partitions, then there are $\binom{n}{2} = \frac{n\times(n-2)}{2}$ partitions that can be evaluated or O(n^2). In the case of medium sized bound sets there are $\binom{n}{n/2} \approx \frac{2^n}{10}$ partitions that can be evaluated or O(2^n). For instance, given that there are 100 input variables, then there are $\approx 1\times10^{29}$, 50-variable bound sets. This is a very large search space, much too large to go through all of the partitions with 50 bound set variables. If the bound set is small, then search becomes much smaller. Given 100 input variables there are $\binom{100}{2} = 4950$, 2 variable bound sets. Therefore, based on complexity, it is much easier to evaluate all the small bound sets.

In fact, in terms of logic synthesis (recall the small percent of *don't cares*), the possibility of finding a good decomposition with small bound sets would probably not work. But, from the analysis above, the algorithm presented here is based on the premise that small bound sets are the best way to decompose machine learning problems.

5. Algorithm Implementation

This section presents an algorithm that uses multi-valued functional decomposition as a machine learning algorithm. The section is broken down into subsections which describe some of the important aspects of the presented algorithm.

5.1. Machine Learning Complexity Measure

The complexity measure selected is a measure proposed in [1]. For more on complexity measures see [7, 8, 16, 17]. This complexity measure, called Decomposed Functional Cardinality (DFC) is the sum of the cardinalities of the component functions in a combinational representation, when the sum has been minimized. Any function is allowed as a component, but its cost goes up with its *cardinality*. The *cardinality* of an n-input, m-output binary function is $2^n * m$. Thus, any n-input gate has the same cardinality as any other n-input gate, regardless of the functionality of that function.

Definition 4 A function $f(x_0, x_1, \ldots, x_{n-1})$ with k-valued logic and m-outputs, has *cardinality* $k^n * m$. DFC is defined as the sum of the function cardinality of each decomposed partial block. A non-decomposable function is defined as a function that has a DFC$>k^n * m$.

5.2. Data Structure of the Decomposer

The representation of a function can sometimes make or break the algorithms that are applied to it. The importance of data structure has been shown in [10] in which a Binary Decision Diagram (BDD) [11, 5, 2] based decomposition proved very efficient in the decomposition of logic functions for FPGAs. The use of Multi-value Decision Diagrams (MDD) was also shown to be effective in the decomposition of multi-valued functions with many *don't cares* [7]. The problem with the MDD package is that it does not allow variables of different valued-ness. Also, it was found that when the function implemented in the MDD has many values, the MDD package acted more like a decision tree than a BDD, that is, there was very little compression within the nodes of the MDD.

Taking the ideas from [19] of mapping a MDD onto a BDD, we created a new MDD package that is built on top of our existing BDD package[1]. By mapping a MDD onto a BDD package, the MDD now has the ability to have variables of different sizes.

5.3. Inessential Variables

One of the most important procedures in a machine learning algorithm is the removal of non-essential(*vacuous*) variables from the function. There are actually three different classes of variables as defined in [9]. A variable is either *essential*, *inessential* or *vacuous*. *Essential* and *vacuous* variables are easily defined as variables that either have an impact on the output, or do not have an impact on the output. An *inessential* is a variable that by setting *don't cares* in a certain way, the variable either becomes *essential* or *vacuous*. Because of the structure of the MDD, if a variable is vacuous it is no longer included in the database, so they are automatically removed from the function. For each *inessential* variable, *don't cares* are forced to make the variable vacuous.

To determine which variables can be made vacuous a compatibility graph algorithm is used. Each node in the graph represents an *inessential* variable, and each vertex between two nodes designates that both variables can be forced vacuous at the same time. From this graph a maximum clique algorithm is used to determine the largest clique of *inessential* variables that can be forced to become *vacuous*.

5.4. The Decomposition Algorithm

The algorithm is broken down into the following steps:
1. Run graph algorithm to find all *inessential* variable cliques. Evaluate the resulting maximal cliques by calling the decomposer on each of the resulting functions.

2. Decomposer algorithms:
2a. Determine if the current function's DFC is larger than the best DFC found so far. If it is, then return (this is a method of pruning, such that not all functions are decomposed).
2b. Evaluate all two-variable bound sets.
2c. Determine which partitions could lead to a decomposition. This is done by evaluating the column multiplicity of each of the partitions and selecting those with higher ranks. The partitions with the smallest column multiplicity are ranked higher than those with higher column multiplicities.
2d. From the list of possible partitions from 2b, systematically decompose each function f into sub-blocks F and Φ. Recursively call step 2 with F and Φ, separately.

[1]See: http://www.ee.pdx.edu/~cfiles/bdd/

3. Done. Return the function with the best DFC.

6. Results

The results obtained demonstrate how the algorithm can reduce the complexity of a given problem while keeping the *error* low. The testing of the algorithm was done on completely specified functions with eight input variables. There are several reasons for using these small functions: the functions are completely specified which makes the comparison of error exact, the smallest DFC of the functions are known, so analysis of the decomposers heuristics can be performed, and many training samples may be run.

The training sets are made from randomly selecting care terms from a given function. The selection is done in steps of 8 in the range [8,248]. At each step, 20 different data sets were found by randomly selecting P care terms, where P is equal to the current step. The reason for selecting 20 different data sets is to get a good distribution so the output could be fully analyzed.

The test functions that were used come from the Wright Patterson's Pattern Theory Group set of benchmarks[2]. Of course, this is just a measure to see how well a function can handle data which is highly unspecified. But, notice that the minimum number of care terms evaluated is 8 care terms and 248 *don't cares*, which means that the function is 96.875% incompletely specified.

The results are shown in the Figures 2, 3 and 4. For each function the following are shown in the figures: the *error* of the training output to the original KDD function, the DFC that was found on the training function, and the time that it took for the algorithm to find a solution. The title of each graph designates what is actually being displayed on the $Y - axis$ (*error*, DFC, Time), while the $X - axis$ is the number of training samples. For each function, there is some error when the number of training samples is quite small, but as the number of training samples gets larger, the amount of error goes to zero. Also note that the algorithm usually finds the correct function once it finds the DFC of the original function. That is, as the number of training samples grows the algorithm is able to find the original function, and the DFC of the original function. The last thing to note is the time to a solution. We found it quite interesting that the time to find a solution was much longer for small training samples. The reason for this is the number of possibilities that exist with a large number of *don't knows*.

Each of the Figures 2, 3 and 4 was selected out of the 54 possible functions in the benchmark set. These three functions, in general, show the execution of the decomposer. The first two functions are small examples of Knowledge

[2]benchmarks available at http://www.ee.pdx.edu/polo/function/

Discovery in Databases, while the third is a character recognition function.

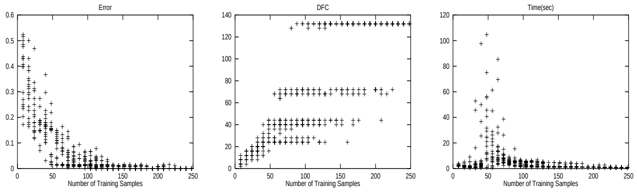


Figure 2. Finding the error, DFC, and time of the decomposer on the function KDD5

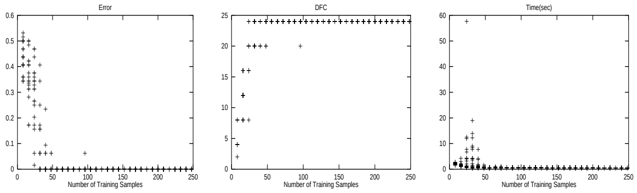


Figure 3. Finding the error, DFC, and time of the decomposer on the function KDD10

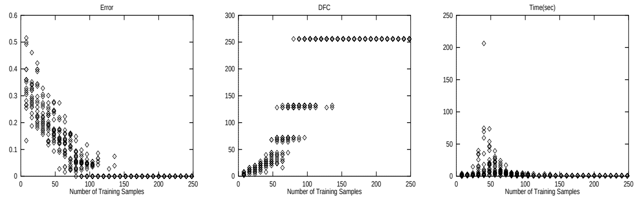


Figure 4. Finding the error, DFC, and time of the decomposer on the function CH52f4

7. Conclusion

This paper presented a new method for the decomposition of highly unspecified multi-valued output functions. The main goal of this paper was to present a new method of decomposition that is focused on reducing the error in a solution. Finding error is actually a test of an algorithm to see how well it will behave on real-world data. In this paper, the proposed algorithm was tested on a KDD based benchmark test set. From this data analysis, the program did very well on reducing the error between the original function and the solution.

References

[1] Y. Abu-Mostafa. *Complexity in Information Theory*. Springer-Verlag, New York, 1988.

[2] S. Akers. Binary decision diagrams. *IEEE Trans. on Computers*, 6:509–516, June 1978.

[3] R. L. Ashenhurst. The decomposition of switching functions. *International Symposium on Theory Switching Functions*, pages 74–116, 1959.

[4] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information Processing Letters*, pages 377–80, 1987.

[5] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computer*, 8:677–691, August 1986.

[6] H. A. Curtis. *A new approach to the design of switching circuits*. Van Nostrand, Princeton, NJ, 1962.

[7] C. Files, R. Drechsler, and M. Perkowski. Functional decomposition of MVL functions using multi-valued decision diagram. *International Symposium on Multi-Valued Logic*, pages 27–32, May 1997.

[8] C. Files and M. Perkowski. Multi-valued functional decomposition as a machine learning method. *International Symposium on Multi-Valued Logic*, May 1998.

[9] S. Hight. Complex disjunctive decomposition of incompletely specified boolean functions. *Transactions on Computers*, pages 103–110, 1973.

[10] Y. T. Lai, M. Pedram, and S. B. K. Vrudhula. BDD based decomposition of logic functions with application to FPGA synthesis. *Design Automation Conference*, pages 642–647, 1993.

[11] C. Y. Lee. Binary decision programs. *Bell System Technical Journal*, 4:985–999, July 1959.

[12] G. G. Lendaris and G. Stanley. On the structure-dependent prperties of adaptive logic networks. Technical report, GM Defense Research Laboratories, Sata Barbara, California, July 1963.

[13] T. Luba. Decomposition of multiple-valued functions. *International Symposium on Multi-Valued Logic*, pages 256–261, 1995.

[14] T. Luba and R. Lasocki. Decomposition of multiple-valued boolean functions. *Applied Math and Computer Science*, 4(1):125–138, 1994.

[15] R. Michalski. Discovering classification rules using variable-valued logic system vl1. In *Third International Conference on Artifical Intelligence*, pages 162–172, Stanford University, 1973.

[16] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. S. Zhang. Decomposition of multiple-valued relations. *International Symposium on Multi-Valued Logic*, May 1997.

[17] T. Ross, M. Axtell, M. Noviskey, and D. Gadd. Pattern theory paradigm for system design. *Midwest Symposium on Circuits and Systems*, 1993.

[18] J. P. Roth and R. M. Karp. Minimization over boolean graphs. *IBM Journal of Research and Development*, pages 227–38, 1962.

[19] A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. *International Conference on CAD*, pages 92–95, 1990.