Learning Boolean Circuits from Examples for Approximate Logic Synthesis

Sina Boroumand
Imperial College London
London, UK
s.boroumand18@imperial.ac.uk

Christos-Savvas Bouganis
Imperial College London
London, UK
christos-savvas.bouganis@imperial.ac.uk

George A. Constantinides
Imperial College London
London, UK
g.constantinides@imperial.ac.uk

ABSTRACT

Many computing applications are inherently error resilient. Thus, it is possible to decrease computing accuracy to achieve greater efficiency in area, performance, and/or energy consumption. In recent years, a slew of automatic techniques for approximate computing has been proposed; however, most of these techniques require full knowledge of an exact, or 'golden' circuit description. In contrast, there has been significant recent interest in synthesizing computation from examples, a form of supervised learning. In this paper, we explore the relationship between supervised learning of Boolean circuits and existing work on synthesizing incompletely-specified functions. We show that when considered through a machine learning lens, the latter work provides a good training accuracy but poor test accuracy. We contrast this with prior work from the 1990s which uses mutual information to steer the search process, aiming for good generalization. By combining this early work with a recent approach to learning logic functions, we are able to achieve a scalable and efficient machine learning approach for Boolean circuits in terms of area/delay/test-error trade-off.

CCS CONCEPTS

• **Hardware** → **Combinational circuits**; **Combinational synthesis**; • **Computing methodologies** → *Supervised learning*.

KEYWORDS

logic synthesis, information theory, approximate computing

ACM Reference Format:
Sina Boroumand, Christos-Savvas Bouganis, and George A. Constantinides. 2021. Learning Boolean Circuits from Examples for Approximate Logic Synthesis. In *ASP-DAC 2021: ACM Asia South Pacific Design Automation Conference, June 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Applications from embedded systems and internet of things (IoT) devices to large-scale data centers used to perform massive scientific computing, financial analysis, and social media computation, demand huge processing and storage load. As a result, it is essential to design power-efficient and high performance integrated circuits for these applications, which becomes a challenging task. To tackle this, it is possible to design specialized hardware accelerators, as opposed to general-purposed processors, to achieve energy efficiency. Hardware accelerators, as Boolean circuits, are attracting considerable interest in part to the growth of of machine learning for high-performance computing [4]. Besides, there exists intrinsically error resilient applications which can tolerate a limited amount of computational inaccuracy. Due to this fact, hardware accelerator designers are able to exploit *Approximate Computing* techniques to relax the accuracy in order to reduce area, delay, and/or energy consumption [6].

In the past decade, many *Approximate Logic Synthesis* (ALS) techniques have been proposed [10, 12, 13]. ALS is the process of finding a Boolean circuit that may introduce some error compared to the original circuit or specification without violating a pre-specified error tolerance. The circuit behaviour descriptions which are typically used in logic synthesis can be classified into two categories: 1) descriptions with *known* structure, such as RTL (register transfer level) specification, and 2) descriptions with *unknown* structure, such as truth table presentation. For the first category, ALS techniques use the knowledge of the structure to synthesize an approximate version of the golden circuit specification that adheres to the error and/or hardware resource constraints, *e.g.* [12, 13]. For the second category, which is the main case in this paper, since the (fully/partially specified) truth table is the only information, classic logic synthesis methods can be employed to construct the Boolean network. However, accuracy relaxation cannot be explicitly dictated to these methods. In fact, this problem can be viewed as the generalization of the *Minimum Circuit Size Problem* (MCSP) [7], which in contrast to MCSP, it determines whether a circuit with at most a certain size and accuracy exists to compute a full/partial truth table of a Boolean function.

This paper's problem can also be considered as a supervised machine learning problem, where the task is to find an approximate Boolean circuit that maps inputs to outputs based on a set of input/output examples. This can be represented as logic synthesis of incompletely-specified Boolean functions. Of course, incompletely-specified Boolean functions are not new: they are well-known within logic synthesis. The classic Espresso synthesis tool [9] already provides a way to describe incompletely-specified functions via 'don't-care' conditions. This suggests a possible use in learning from examples: can we consider all truth table entries outside the training data to be don't-care conditions? Such an approach would allow the reuse of classic techniques for supervised learning, but

ASP-DAC 2021, June 18–21, 2021, Tokyo, Japan

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

since they have not been designed with *generalization* in mind, it is unclear to what extent circuits learnt in this way would generalize to unseen examples. We answer this question in this paper.

Thereupon, we proposed **L**ogic **S**ynthesis from **E**xamples (LSE) which employs notions of information theory and Boolean function learning to progressively build Boolean circuit using training examples. To make comparison, we evaluated our results in comparison with existing works. We show that, in this supervised learning setup, in most cases our technique outperforms them in terms of area/delay versus accuracy trade-off.

The rest of this paper is organized as follows: Section 2 gives an overview on related works in the literature. The third section provides the preliminaries of LSE, such as mutual information. In Section 4, we define the problem and then describe our proposed methodology, and in Section 5 we report the experiments. Finally, we conclude the paper with a brief discussion in Section 6.

2 RELATED WORKS

Various approaches have been proposed to synthesize Boolean circuits from a fully or partially specified truth table. Espresso [9] is among the logic synthesis and optimizer tools that are capable of synthesizing incompletely-specified Boolean functions. However, Espresso is a two-level logic minimizer and cannot be used directly to produce multi-level Boolean circuits. Espresso reads the multi-valued Boolean function and splits it into ON-set, OFF-set, and DC-set, and uses procedures such as: *reduce* and *expand* to iteratively minimize a cost function. The Espresso strategy reduces the implicants to non-prime cubes (product terms) and then expands cubes to prime implicants and finally extracts a minimal subset of the prime implicants [9] — An implicant is a cube which does not contain any minterm in the OFF-set of the function. This procedure is iterated until no reduction in the number of cubes is achievable.

On the other hand, most well-known logic synthesis tools, due to input format limitations, are unable to handle truth table descriptions with large number of explicitly declared don't-care conditions, such as ABC [1] and SIS [11]. Nevertheless, Chang *et al.* [2] re-purposed techniques from synthesis tools to synthesize incompletely-specified functions to multi-level Boolean circuit. This method is a fast logic synthesis technique that synthesizes a truth table with extensive numbers of don't-care conditions to a multi-level circuit. CleanSlate uses an *expand* function, which is similar to the expand function of Espresso, to generate a two-level netlist, and this netlist is then fed to ABC for further logic optimization in which the final product is a multi-level Boolean circuit.

In contrast, Oliveira *et al.* [8] proposed an algorithm to learn and generate Boolean networks from examples. This algorithm obtains both the topology of the Boolean network and the functionality of each node within the network using concepts such as mutual information. Alternatively, Chatterjee achieved generalization by memorizing the training examples using a neural network of lookup tables (LUTs) with random interconnections [3]. This method could also be re-purposed for approximate logic synthesis problem; however, in order to achieve a good test accuracy, a relatively deep and large network is required.

3 PRELIMINARIES

The following sections are devoted to the preliminaries of the proposed method of this paper; logic synthesis from examples (LSE): *mutual information* and *learning a single lookup table.*

3.1 Mutual Information

For two random variables X and Y with values over the space $\mathcal{X}\times\mathcal{Y}$, the mutual information $I(X;Y)$ measures the average reduction in uncertainty about X that results from learning the value of Y; or vice versa [5]. This can be expressed in terms of *Shannon entropy* as:

$$I(Y;X) = H(X) - H(X|Y), \tag{1}$$

where $H(X)$ is the entropy of X and $H(X|Y)$ is the conditional entropy of X given Y. The conditional entropy measures the average uncertainty that remains about X when Y is known:

$$H(X|Y) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(x,y)\log\left(\frac{P(x,y)}{P(x)}\right), \tag{2}$$

where $P(x,y)$ is the joint probability of x and y, and $P(x)$ is the marginal probability of x.

3.2 Learning a Single Lookup Table

We adapted a method suggested by Chatterjee [3] to learn the functionality of a single lookup table from a set of training examples. Let $X = (x)$ be a sequence of k-bit training input words and $Y = (y)$ be the corresponding sequence of 1-bit outputs. The aim is to learn the function $f : \mathbb{B}^k \to \mathbb{B}$ over the training example set $\{(x,y)\}$. A lookup table with k-inputs has 2^k possible input patterns and p_i programming bits, where $i = 0,\ldots,2^k-1$ corresponds to each input pattern. In the learning process, we count the number of times the output of each input pattern a_i in the training examples has been seen as 0, denoted by $c_{i,0}$, and 1, denoted by $c_{i,1}$. Next, we construct the k-input lookup table by setting each programming bit p_i as follows:

$$p_i = \begin{cases} 1 & \text{if } c_{i,0} < c_{i,1}, \\ 0 & \text{if } c_{i,0} > c_{i,1}, \\ b & \text{if } c_{i,0} = c_{i,1}, \end{cases} \tag{3}$$

where $b \in \{0,1\}$ is selected uniformly and randomly.

4 LOGIC SYNTHESIS FROM EXAMPLES (LSE)

Let $X = (x_i)$ be a sequence of training inputs, where x_i denotes the i-th input sample drawn from possible values $\mathcal{X}$, and $Y = (y_i)$ the corresponding sequence of labels, where y_i denotes its corresponding 1-bit label (desired output) drawn from possible values $\mathcal{Y} = \{0,1\}$. The aim of the learning algorithm is to find an *approximate circuit* implementing a function $f : \mathcal{X} \to \mathcal{Y}$, mapping X to Y approximately, where

$$f = \arg\max_{\tilde{f}} \; I\big(Y, \tilde{f}(X)\big) \quad \text{s.t.} \quad \text{Area}(\tilde{f}) \le A_{\max}. \tag{4}$$

$I(\cdot)$ denotes the mutual information between Y and $\tilde{f}(X)$, with $\tilde{f}$ applied element-wise, and $A_{\max}$ is the maximum acceptable area.

Algorithm 1: The LSE synthesis flow.

Input : Training example set X, Y
Output: Netlist f
1 **for** $i = 1$ *to* m **do**
2 $\quad f^{(i)} \leftarrow$ `muesliModified` $(X, y^{(i)})$
3 **end**
4 Build f by combining all $f^{(i)}$
5 Logic optimization of f using ABC
6 **return** The optimized netlist f

4.1 Methodology

We proposed an algorithm for LSE which aims to approximately synthesize a Boolean circuit from a set of training examples by learning the functionality and the topology. Given a set of training examples, let X be a sequence of n-bit binary inputs, each drawn from possible values $\mathcal{X} \subseteq \mathbb{B}^n$, and Y be the sequence of corresponding m-bit binary outputs, each drawn from possible values $\mathcal{Y} \subseteq \mathbb{B}^m$. The LSE algorithm, shown in Algorithm 1, individually searches for single-output Boolean circuits for each output bit of Y. $y^{(i)}$ denotes the i-th output bit of Y. These m Boolean circuits are then combined into one Boolean circuit $f : \mathcal{X} \rightarrow \mathcal{Y}$ with n input and m output bits, and then optimized and technology mapped using ABC synthesis tool.

The LSE algorithm uses a function to search for single output Boolean circuits. This function, for each output bit $y^{(i)}$, progressively builds a Boolean network, denoted as $f^{(i)}$, by learning Boolean nodes — *i.e.* lookup tables, that locally maximize the mutual information with the output to globally maximizes the mutual information (Eq. 4). This function is directly based on [8], only differing in Line 9 and Lines 14–18. We describe the algorithm here in full, outlined in Algorithm 2, for completeness. The function, named `muesliModified`, keeps a list of *candidate* nodes C, which is initially populated with n primary inputs. During iterations of the algorithm, each newly learned LUT is added to the candidate list. The algorithm proceeds by selecting active nodes from C, storing them in the *active list* A. Variable act is defined as the number of nodes in A and also points to the least informative node in the list. The algorithm starts with act = 0 and sup = init_sup, and selects (act + sup − 1) nodes from C. The active list is sorted in decreasing order of mutual information of candidate nodes with the output. Given a candidate nodes list C, the active list $A = (a_0, \ldots, a_{j-1})$ with j items is selected as follows:

$$
\begin{aligned}
a_0 &= \arg\max_{v_0 \in C} I(v_0; y^{(i)}), \\
a_1 &= \arg\max_{v_1 \in C \setminus \{a_0\}} I(v_1 \| a_0; y^{(i)}), \\
&\vdots \\
a_j &= \arg\max_{v_j \in C \setminus \{a_0, \ldots, a_{j-1}\}} I(v_j \| a_0 \| \ldots \| a_{j-1}; y^{(i)}),
\end{aligned}
\tag{5}
$$

where $\|$ denotes the concatenation into a multi-bit variable.

In each iteration of algorithm, function `bestFunction` tries to find a Boolean function f', with sup inputs assigned with A[act : act + sup − 1], and if the condition below holds, f' is added to the candidate nodes:

$$
I(A[0 : \text{act}]; y^{(i)}) < I(A[0 : \text{act} - 1] \cup f'; y^{(i)}). \tag{6}
$$

In other words, it tries to substitute the least informative node in the active list with a new Boolean node that increases the gain in predictability of the output, given the active nodes.

Since, for large sup values, enumeration of all possible functions to find the functionality of f' is infeasible, we used the technique described in Section 3.2 to learn the functionality of a single lookup table. The complexity of learning a LUT using the method proposed in [3] grows linearly with the number of instances in the training examples, whereas the complexity of searching all possible functions grows exponentially with the size of sup — which is $O(2^{2^{sup}})$. Oliveira *et. al.* [8] used a partitioning algorithm to find the functionality of f' with a large sup; however, in their paper there is no description to demonstrate this adaptation.

If a desirable Boolean function f' is successfully observed, the algorithm inserts it into C, as well as re-initializing act to 0 and sup to init_sup. Otherwise, it tries again by incrementing *act* until no gain is achievable or act = max_act. Thereby, the algorithm increments sup (up to max_sup) to expand the search space. This loop continues until function `notDone` interrupts it or the size of sup exceeds max_sup. Function `notDone` returns true if $\varepsilon(v, y) < \varepsilon'$, for some $v \in C$, and false otherwise. $\varepsilon(v, y) = \frac{I(v;y)}{H(y)}$ is the mutual information between v and y, normalized to the entropy of y, and ε' is the bound value to end the learning process.

Finally, in case of success, `muesliModified` returns a node from C that maximizes mutual information with the output. This node can either be one of the primary inputs forwarded directly to the output, or latent generated LUTs. As mentioned earlier, all $f^{(i)}$ functions are combined into one Boolean function f with n input bits and m output bits (Line 4 of Algorithm 1). In the last step of

Algorithm 2: Modified Muesli

1 **Function** `muesliModified` $(X, y^{(i)})$
2 $\quad$ initialize max_act, max_sup, init_sup
3 $\quad$ initialize C with primary inputs
4 $\quad$ sup ← init_sup
5 $\quad$ **while** `notDone()` ∧ sup < max_sup **do**
6 $\quad\quad$ act ← 0
7 $\quad\quad$ **do**
8 $\quad\quad\quad$ $A \leftarrow$ `selectActs` $(C, \text{sup}, \text{act})$
9 $\quad\quad\quad$ $f' \leftarrow$ `bestFunction`$(A, X, y^{(i)})$
10 $\quad\quad\quad$ $C \leftarrow C \cup f'$
11 $\quad\quad\quad$ *success* ← $(f' \neq \varnothing)$
12 $\quad\quad\quad$ act + +
13 $\quad\quad$ **while** success = FALSE ∧ act < max_act
14 $\quad\quad$ **if** success = TRUE **then**
15 $\quad\quad\quad$ sup ← init_sup
16 $\quad\quad$ **else**
17 $\quad\quad\quad$ sup + +
18 $\quad\quad$ **end**
19 $\quad$ **end**
20 $\quad$ **return** $\arg\max_{f^{(i)} \in C} I(f^{(i)}(X); y^{(i)})$
21 **end**

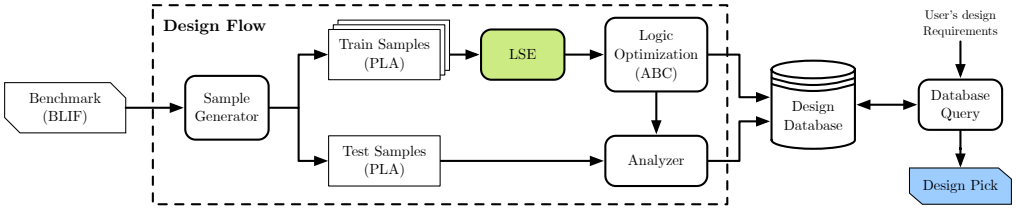

Figure 1: Proposed tool framework flow of learning a netlist from HDL description of a benchmark and reports design metrics: Area, delay, and accuracy. Design points are stored in a database and can be chosen according to the user's design requirements.

LSE, this function is passed to the ABC for combinational logic optimizations, the product circuit is then technology mapped and returned as the final netlist.

The computation time of LSE is non-deterministic. Therefore, we present the complexity of each iteration instead. The bottleneck of the algorithm is in function `selectActs`, which the complexity is $O(2^{(\text{act+sup})}(|C| - \text{act} - \text{sup})|X|)$.

5 EXPERIMENTAL RESULTS

To evaluate the proposed method, we have selected various benchmark examples from MCNC logic synthesis and optimization benchmarks [14]. Table 1 illustrates the characteristics of each benchmark. These benchmarks are randomly selected from combinational multi-level examples with logic label on account of the error metric we utilize in this paper. In addition, we performed an experiment on the MNIST data set to check the applicability of the proposed algorithm to a real machine learning application — Section 5.4. We compare our method with two existing methods, Espresso [9] and *CleanSlate* [2] which, as noted in the introduction, can be re-purposed in a form of supervised machine learning for logic synthesis from examples. In the following sections, we describe the area/delay model and the quality metric that is used in the reports.

5.1 Area and Delay Model

The resource utilization reported in this paper is estimated based on a network of k-LUTs. The delay is computed by the longest path in the circuit and area by sum of node areas. For a single k-LUT, with k as the number of inputs, the area is modeled by 2^k and the delay is modeled by $k - 1$.

Table 1: Characteristics of benchmarks. Area and delay numbers are based on the model in Section 5.1.

Benchmark	#In/Out	Area	Delay
cu	14/11	154	7
cm138a	6/8	64	4
cm150a	21/1	184	9
cm151a	12/2	92	7
cordic	23/2	188	10
comp	32/3	328	11
cc	21/20	154	7
c8	28/18	418	8
cm162a	14/5	116	8
cm163a	16/5	116	7

5.2 Error Metric

To evaluate the quality of the synthesized circuits, we employ the test error rate. For a given test example set $T = \{(x, y)\}$ with t samples, let $X = (x)$ be the sequence of test input drawn from $\mathcal{X}$, and $Y = (y)$ be the sequence of corresponding outputs drawn from $\mathcal{Y}$. For an approximate circuit f, the *test error rate* is defined as follows:

$$\text{Test Error } (\%) = \frac{\left|\{(x, y) \in T \mid f(x) \neq y\}\right|}{t} \times 100, \tag{7}$$

which is the total number of times in the test sample set that the predicted output by f differs from the actual output, over the total number of test samples.

5.3 Logic Synthesis

Our logic synthesis flow is presented in Figure 1. In this flow, for each benchmark, we generated multiple random training and test samples, and simulated them using exact circuit descriptions to acquire exact outputs. All input sample patterns are generated using a uniform pseudo-random number generator. Since in CleanSlate and Espresso there is no explicit control on the level approximation, we used various size example sets, in terms of the number of truth table rows, to implicitly conduct approximate synthesis. Morover, for logic synthesis methods, circuits synthesized from truth tables with a higher number of don't-care conditions are expected to be smaller. On the other hand, for the LSE method, it is possible to explicitly regulate approximation by changing ε', which terminates the algorithm when the normalized mutual information of a node in the candidate nodes with the output reaches ε'.

The number of truth table rows for training samples are selected from a variety of values $\{100, \ldots, 10000\}$, and for test samples, 100,000 random truth table rows are generated. These samples are generated multiple times in order to demonstrate the variability of the results as a function of sample size. These samples are synthesized using LSE, and generated Boolean circuits are passed to ABC for logic optimizations and technology mapping. Then, the accuracy of each netlist is evaluated using the test samples through the *Analyzer*, which calculates the test error through simulating test samples. For CleanSlate and Espresso results, we used a similar flow to synthesize the aforementioned logic synthesis benchmarks out of randomly generated training examples. Figures 2 and 3 report the area and delay, versus test error (%) of each synthesized benchmark for the different methods, respectively. In these figures, only Pareto frontier design points are shown — dominated designs are not shown, nor design points with test error larger than 10%.

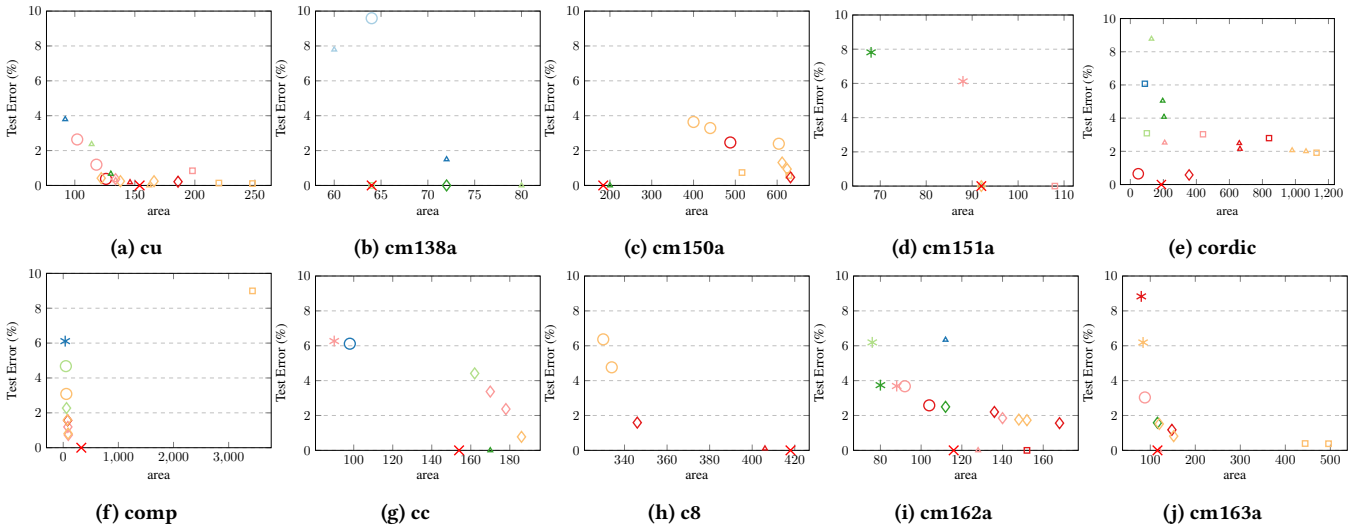

Figure 2: Area report of Pareto-efficient design curves of LSE with different ε': 0.7 (∗), 0.8 (○), and 0.9 (◇), CleanSlate (□), and Espresso (△). The different sample sizes are depicted in different colors. The red cross (×) shows the area of the original circuit.

As expected, for all benchmarks, results show that Espresso outperformed CleanSlate. The reason is that CleanSlate's expand is covered by the Espresso's algorithm. Furthermore, since the problem is to optimize a two-level Boolean function, and Espresso is equipped with multiple procedures for the two-level optimization in particular, the output is minimized thoroughly. In addition, we used ABC multi-level optimizations to produce the final netlists, which means all further multi-level optimizations are done through ABC, and is shared between these two methods. However, in most cases, CleanSlate performed faster than Espresso and LSE.

Nonetheless, for benchmarks with a large number of inputs, neither CleanSlate nor Espresso were able to synthesize a circuit, such as *comp* with 32 input bits. In terms of test accuracy, almost all circuits generated by CleanSlate in this way have very low test accuracy. The results show that, in almost all cases, the Pareto curve of LSE dominates the other two methods, demonstrating that exploiting mutual information helps to achieve better generalization. However, for some benchmarks, Espresso was able to reproduce the expected exact circuit with zero test error. We also reported the Pareto-efficient frontier of all design points of the three algorithms together, and we counted the number of design points associated with each method collectively and report them in Table 2. The table shows that LSE was able to build more Pareto-efficient design choices, compared to other methods.

Table 2: Number of all Pareto frontier design choices generated by each algorithm. P_{area} and P_{delay} are the number of points in Pareto frontier which for each circuit an algorithm dominates in area and delay, respectively. The greatest values for each circuit are in bold.

Benchmark	LSE		Espresso		CleanSlate	
	P_{area}	P_{delay}	P_{area}	P_{delay}	P_{area}	P_{delay}
cu	**8**	**3**	5	1	0	0
cm138a	1	**1**	**2**	0	1	0
cm150a	**4**	1	3	**3**	0	0
cm151a	**4**	**3**	0	0	0	0
cordic	**4**	**5**	0	0	0	0
comp	**10**	**2**	0	0	0	0
cc	**4**	**3**	1	1	0	0
c8	**6**	3	1	**4**	0	0
cm162a	**7**	**3**	1	1	0	2
cm163a	**7**	**3**	1	**3**	0	0

5.4 MNIST Binary Classification

In this section, we present logic synthesis from the MNIST dataset to evaluate the accuracy performance of the proposed algorithm on a real-world application. For binary classification of the MNIST database where digits from '0' to '4' are classified as the 0 class, and digits from '5' to '9' as the 1 class, for 1-bit image pixels, we studied the task of learning a Boolean function $f : \mathbb{B}^{28\times28} \to \mathbb{B}$ using the LSE and other methods. We tried this on Espresso and CleanSlate and both timed-out without generating any circuit.

In terms of accuracy performance, we compare our method with *Memorization* [3] technique. For Memorization, we set up a network of 12-input LUTs with 6 layers. For the proposed algorithm, we ran it for init_sup={8,10}, denoted as LSE-8 and LSE-10, respectively. Table 3 outlines the training and test accuracy of these two methods. As expected, the training accuracy of Memorization technique is higher than LSE method, since it aims to memorize all the training examples. In terms of test accuracy, LSE provides more accuracy compared to Memorization, which in fact is a result of its more complicated algorithm that uses mutual information to build the Boolean circuit progressively. This experiment shows that LSE is capable of handling tasks with large number of inputs and training examples, and provides acceptable training and testing accuracy. In terms of area and delay, LSE-8 contains of 86× 8-LUT and has

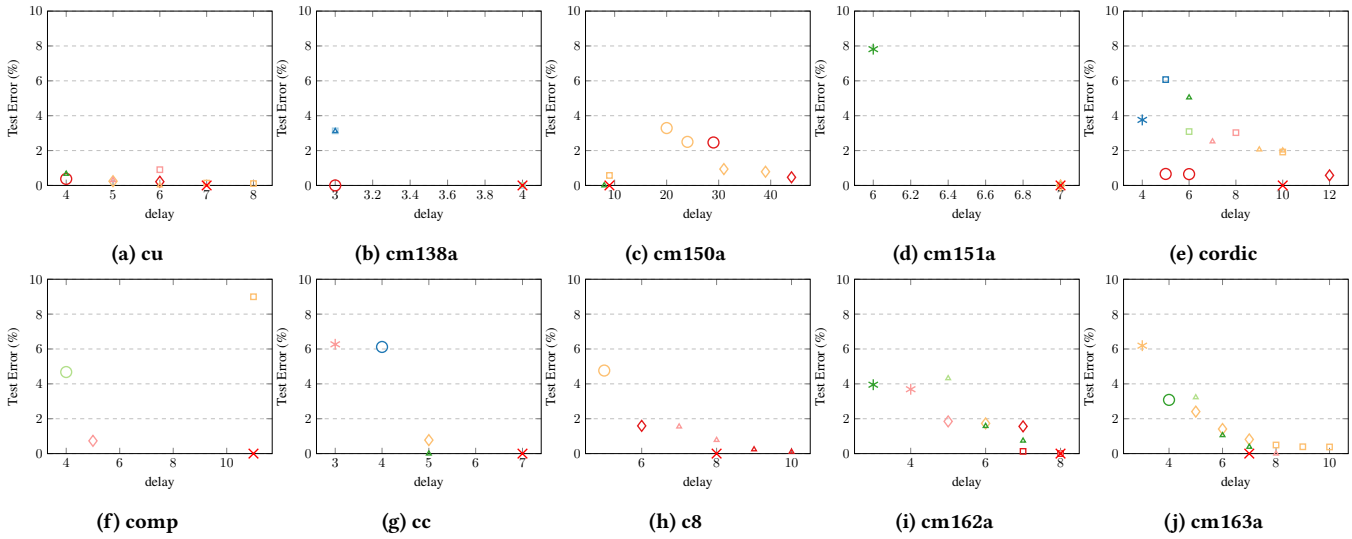

Figure 3: Delay report of Pareto-efficient design curves of LSE with different ε': 0.7 (∗), 0.8 (○), and 0.9 (◇), CleanSlate (□), and Espresso (△). The different sample sizes are depicted in different colors. The red cross (×) shows the delay of the original circuit.

Table 3: Accuracy performance of binary classification of MNIST with LSE in terms of training and test accuracy.

Method	Accuracy (%)	
	Training	Test
LSE-8	**94.68**	**92.76**
LSE-10	**94.80**	**92.56**
Memorization [3]	99.00	90.00
Random Guess	50.00	50.00

the delay of 574, and LSE-10 contains of 17× 10-LUT and has the delay of 153.

6 CONCLUSION

In this paper, we addressed the problem of finding an approximate Boolean circuit from a set of input/output examples without structure knowledge. This problem can be considered as a supervised machine learning problem, and we showed that it is possible to repurpose classic logic synthesis methods to work around it. However, since these methods are not meant for this purpose, they do not essentially provide a good generalization. Experiments show that the proposed method is able to proficiently steer the design search space, using mutual information, to build Boolean circuits from examples, that provide good generalization and lower area/delay utilization in comparison to the existing methods. We also studied our proposed method on a real application: MNIST binary classification. The results of this study imply that the proposed method has the potential to build Boolean circuits with high accuracy from large training example sets with a large number of primary inputs. However, this method is limited only to single-bit outputs. Further studies, which take the word-level search into account, will need to be undertaken, which helps to search effectively for Boolean circuits with multi-bit outputs.

REFERENCES

[1] Robert Brayton and Alan Mishchenko. 2010. ABC: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*. Springer, 24–40.
[2] Kai-Hui Chang, Valeria Bertacco, Igor L. Markov, and Alan Mishchenko. 2010. Logic Synthesis and Circuit Customization Using Extensive External Don't-Cares. *ACM Trans. Des. Autom. Electron. Syst.* 15, 3, Article 26 (June 2010), 24 pages.
[3] Satrajit Chatterjee. 2018. Learning and memorization. In *International Conference on Machine Learning*. 755–763.
[4] George A Constantinides. 2020. Rethinking arithmetic for deep neural networks. *Philosophical Transactions of the Royal Society A* 378, 2166 (2020), 20190051.
[5] David J. C. MacKay. 2002. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, USA.
[6] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* 48, 4, Article 62 (March 2016), 33 pages.
[7] Cody D Murray and R Ryan Williams. 2017. On the (non) NP-hardness of computing circuit complexity. *Theory of Computing* 13, 1 (2017), 1–22.
[8] Arlindo L Oliveira and Alberto Sangiovanni-Vincentelli. 1994. Learning complex boolean functions: Algorithms and applications. In *Advances in Neural Information Processing Systems*. 911–918.
[9] Richard L Rudell and Alberto Sangiovanni-Vincentelli. 1987. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6, 5 (1987), 727–750.
[10] Ilaria Scarabottolo, Giovanni Ansaloni, and Laura Pozzi. 2018. Circuit carving: A methodology for the design of approximate hardware. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 545–550.
[11] Ellen M Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R Stephan, Robert K Brayton, and Alberto Sangiovanni-Vincentelli. 1992. SIS: A system for sequential circuit synthesis. (1992).
[12] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. 2013. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1367–1372.
[13] Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. 2012. Salsa: systematic logic synthesis of approximate circuits. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 796–801.
[14] Saeyang Yang. 1991. *Logic synthesis and optimization benchmarks user guide: version 3.0*. Microelectronics Center of North Carolina (MCNC).