Partial Logic Synthesis via Training a Topologically similar Binarized Neural Network

CHAOYI JIN　MASAHIRO FUJITA[†1]

In this paper, we present a new technique and its experimental results for solving partial logic synthesis problems through training a topologically similar binarized neural network. Partial logic synthesis means that most parts of the logic circuit are known whereas the missing portions must be logically synthesized from specification. By replacing 2-input logic gates with neural model and inheriting the topological structure of the logic circuit, we are able to recover the missing parts with back propagation and discrete training. To check the effectiveness of the representation techniques with neural networks, we have compared the two models with large number of independent experiments. Furthermore, we have compared two types of partial training approaches with different candidate-selecting strategies. We have also performed experiments on larger circuits.

1. Introduction

Logic synthesis is a process turning an abstract specification of desired circuit behavior into a design implementation in terms of network of logic gates. Partial logic synthesis is to retrieve a set of missing portions in a structured network composed of logic gates through known specification. Partial logic synthesis, serving as a subproblem for logic synthesis, draws interests in its application into debugging logic bugs, adjusting timing errors and various optimization problems. The typical format of the problem can be illustrated as shown in Figure 1, where a number of vacant parts C1, C2, …, Cn are to be determined. [1] The answer to the problem may not be unique but have to satisfy the overall specification which maintains the desired functionality of the target circuit.

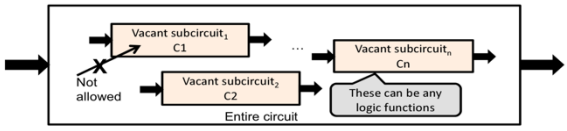


Figure 1. Partial Logic Synthesis problem

Format of solution to partial logic synthesis varies according to the specific requirement of the application. One common format is to assume all vacant parts as two-input one-output logic gates while the whole structure of the circuit stays the same. The process of solving the problem is equivalent to finding a correct set of gate types. Such gate level designs are essentially the same as PPC (Partially Programmable Circuit). In PPC, the gates are usually replaced by LUTs (Look-up Table) to realize the target functionality. In partial logic synthesis, by substituting vacant parts with LUTs, the problem can be formulated as two-level QBF (Quantified Boolean Formula). [2] By utilizing ideas from CEGAR (Counter Example Guided Abstraction Refinement), QBF can be effectively solved by repeatedly applying SAT solvers. [3] A further approach was later put forward to quickly find a solution within small numbers of iterations, avoiding wasting time searching large input space. [4]

On the other hand, artificial neural network is a graph-based algorithm modeled loosely after the human brain, that are designed to help recognize and classify data. The concept of neural network has been raised for decades, but it didn't arouse extensive discussion until in 2012 that AlexNet with astonishing recognition capability was discovered through training deep neural network on GPU. [5] The mysterious capability of capturing complex hyper-features has made it popular in various fields.

Though majority of researches in logic synthesis fields rely on methods with symbolic logic reasoning, in this article, a completely new approach is discussed from a perspective of connectionism. A logic circuit is represented by a neural network composed of numeric weights and homogenous units. Early trial can be found in Zhang et al.'s research [6], where he tested stuck-open faults in CMOS combination circuits using Hopfield Neural Network and help generate robust test pattern in 1992. However, history turned out that Hopfield Neural Network is too computationally expensive and impractical to implement. Instead, we use the idea of nested perceptron to simulate the circuit. To sustain the behavior of gate, each neural unit standing for gate is sealed with binarized function, leading to binarized activations. A candidate solution is no longer logically reasoned about but achieved through feeding data from specification and training via back propagation.

The paper is organized as follows. In Section 2, we explain the neural model for gate simulation. In Section 3, we show the entire framework of the algorithm using a topologically inherited neural network. The detailed algorithm and techniques are also discussed. In Section 4, we show that such method is effective by simulating different scale of circuits. Finally in section 5, we make a brief conclusion on the new method and discuss the new understanding of applying connectionist models to logic synthesis topic.

2. Gate simulation with neural model

2.1 Perceptron model

In machine learning, "perceptron" is an algorithm specialized for binary classification. [7] It leads by a linear combination operation, followed by a nonlinear activation function. For a two-input case with sign function as the activation function, it can be

†1 Department of Electrical Engineering and Information Science, Graduate School of Engineering, The University of Tokyo

expressed as follows.

Suppose $x_1, x_2, y \in \{-1,1\}$, define $f: \boldsymbol{x} \mapsto y$, we have,

$$z = g(\boldsymbol{x}) = w_1 x_1 + w_2 x_2 + b \qquad (1)$$

$$y = f(\boldsymbol{x}) = sign(z) = \begin{cases} 1, z \geq 0 \\ -1, z < 0 \end{cases} \qquad (2)$$

Three parameters need to be trained in this case. Extending to more complex situation, such perceptron can be recognized as the simplest format of neural network, though it is probably unable to learn the non-linearity of most data in practical due to its simplicity. Nested structure of perceptrons is mainly accepted for most neural network. Complex non-linearity is learned through stacking the linear and non-linear function alternatively.

2.2 Kernel Perceptron

Nested perceptrons are definitely not the only way. Even before the renaissance of neural network, SVM is partly believed as a top player in the realm of classification. The reason for its effectiveness is the introduction of kernel function [8] by projecting the input space into a higher dimension, which is believed to capture the non-linearity in a guided way. Suppose the input space is $\mathcal{X}$. For any $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{X}$, certain function $k(\boldsymbol{x}, \boldsymbol{x}')$ can be expressed as an inner product in another space $\mathcal{V}$ with higher dimension. The function k is often referred as kernel function. The computation can be made much simpler if the kernel can be written in the form of feature map $\varphi: \mathcal{X} \to \mathcal{V}$ which satisfies $k(\boldsymbol{x}, \boldsymbol{x}') = \langle \varphi(\boldsymbol{x}), \varphi(\boldsymbol{x}') \rangle_{\mathcal{V}}$. Here the notation $\langle \cdot, \cdot \rangle_{\mathcal{V}}$ means proper inner product. A kernel perceptron is a special perceptron that uses such kernel technique to achieve non-linearity. Such concept does not require nested structure which vastly introduces complexity.

2.3 Two ways of simulating logic gates

To build a computable network classified by threshold about logic gates, we need to replace the gates with classification model. In this paper, we only focus on two-input gate. Totally, there are 16 different logic functions with two inputs which are classified to 4 groups with NPN equivalence, as shown in Table 2. Though we commonly use 0 and 1 to represent False and True in logic, we use -1 and 1 to express False and True instead in our classification model. We seal the black box of the gate like Figure 2 shows. Inside the model, $f(\boldsymbol{x})$ function serves as a type of core calculation and result is fed into a binarized function giving -1 or 1 decision.

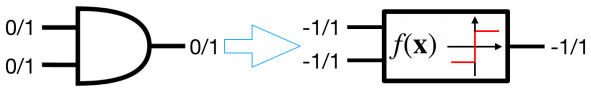


Figure 2. The logic gate is replaced by a sealed classification model, where f(x) serves as a kind of linear or non-linear calculation, followed by a binarized function as classification.

One direct way to build the model is to adapt the perceptron concept. Using the linear combination shown in Function (2), 14 gates can be represented except for XOR and XNOR. As shown in Figure 3, four points on the map are impossible to correctly classified into two categories within a single line.

To cope with that, one solution is to adapt a 2-1 nested perceptron structure. It should be noted that the first two perceptrons do not use a binarized function as activation function, but better uses an approximate function.

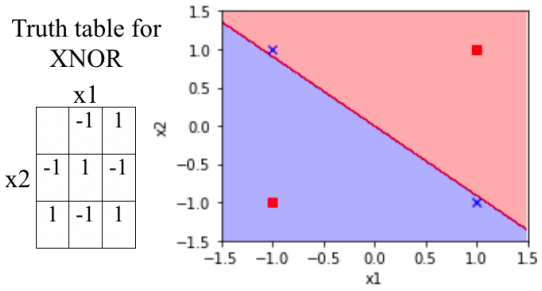


Figure 3. It's impossible to classify the XNOR gate with a linear combination function. Red squared point means True and blue crossing point means False.

Another way to model the gate is to use kernel perceptron. As the kernel function is an in-advance made module designed for expanding dimension, which may contain potential features, the kernel function should be carefully chosen. In this paper, a multiplier item is added in the linear combination Function 1 to add one more dimension to the input space. The feature map can be expressed as:

$$\varphi((x_1, x_2)) = (x_1, x_2, x_1 x_2). \qquad (3)$$

The inner function then uses four parameters to simulate the gates as follows,

$$z = k(\boldsymbol{x}) = w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + b. \qquad (4)$$

Interestingly, the effect of kernel trick happens to be similar to simulation in algebraic analysis of logic circuit. As we can be seen from [8], polynomials of Boolean gates give identical format of equation.

To clarify the real model used to replace gate in the paper, the detailed specification is listed as below.

***Model 1**: 2-1 nested perceptron structure using Function (1) as the key function. The first two perceptrons use approximate function in Table 3 as activation function. The second level perceptron uses binarized function.*

***Model 2**: Kernel perceptron structure adapts Function (4) as the key function. Activation function uses binarized function.*

3. Training the topology-inherited network

3.1 Topology inheritance

With the models for replacing gates, we can connect the models according to the architecture of the logic circuit. It gives us a classification model holding the same input and output dimension as the initial circuit so that the topology is completely inherited.

3.2 Binary Activation and Discrete Learning

A special technique called discrete training is used here to avoid the non-differential problem of sign function during back propagation. The algorithm was first tried by Courbariaux [9] when he studied efficient and valid training algorithm for binarized neural networks. Similarly, our synthesized network can be judged as a binarized neural network due to the seal of

perceptron and binary activation, and so does the usage of discrete training.

Discrete training includes two crucial concepts which are not common in training normal neural networks. First, it uses an approximate function as activation function during backward computation. Table 2 displays a set of candidate for activation function, where the detailed formulae of Tanh and Htanh are expressed below,

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (5)$$

$$Htanh(z) = clip(z, -1, 1) = \begin{cases} -1, & \text{if } z < -1. \\ z, & \text{if } -1 \le z \le 1 \\ 1, & \text{if } z > 1. \end{cases} \quad (6)$$

The second technique relates to clip function as well. In order to keep gradients from explosion along propagation, the weights should be clipped in a range during the update.

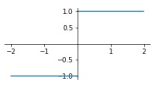
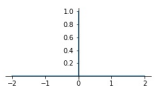
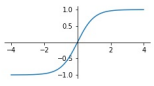
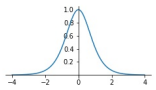
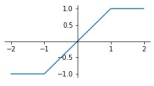
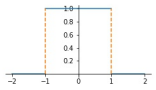
Function	Function plots	Derivative plots	Usage
Sign(x)			Forward
Tanh(x)			Backward
Htanh(x)			Backward

Table 1. Plots on functions used in discrete learning and their corresponding derivative functions.

3.3 Partial Training and Template initialization

Although the whole network is composed and a way to apply back propagation is developed, it is impractical to directly train the whole network from normal initialization. As partial logic synthesis is a process to retrieve just portions of missing gates to satisfy a set of specification, most gates are known and fixed. It is very dangerous to let the whole network execute training. Therefore, majority of the network are untrainable and initialized from template.

This paper proposes two strategies to decide which gates should be allowed to be trainable. To make the explanation explicit and clear, two ways are listed as follows.

***Strategy A**: Only training the unknown gates and initialize all other gates from template.*

***Strategy B**: Training the unknown gates and the ones lying on the path of fan-out of unknown gates. Initialize the rest from template.*

In strategy B, the fan-out of certain gate means the route from that gate to the primary outputs. Figure 4 shows an example of one-bit full adder circuit network in node format. As the topology is inherited, the nodes inside the figure can refer to either gates or a perceptron model. In this figure, red node 2 is chosen as the unknown gate. According to strategy A, only node 2 is trainable, while according to strategy B, node 5, 8, 6, 9, 11 are all trainable as they are on the fan-out cone of the target node. In the implementation, those nodes lying on the fan-out path shall not have a changeable logic behavior. Therefore, a fairly low learning rate is given to those nodes when using the strategy B.

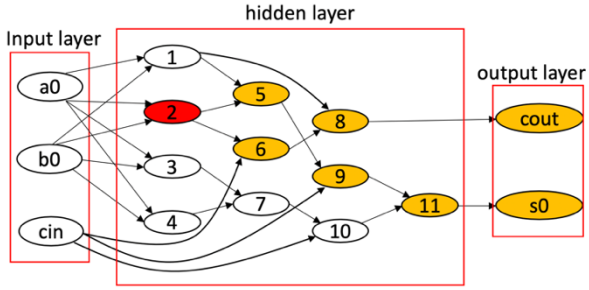


Figure 4. Example of partial training. The red node 2 is unknown. In strategy A, only red node gets trained, while in strategy B, both red nodes and orange nodes get trained.

Algorithm: Training a perceptron based neural network. C is the cost function for minibatch. $1_{Constraint}$ means the result is 1 only when constraint is satisfied, otherwise it equals to 0. The function $Binarize()$ specifies how to deterministically binarize the activations and weights, and $Clip()$, specifies how to clip the weights. The order from inputs to outputs is any acyclic path based on the original structure.

Require: knowing a minibatch of inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$, weights W_i, inputs $\boldsymbol{a}_i$, outputs b_i, learning rate η_i for node i. We suppose SGD (Stochastic Gradient Decrease) for optimizer here. Kernel function is φ. Assume that output of node is directly connected to input of next corresponding node.

Ensure: updated weights W_i^{t+1} for node i.

{1. Forward propagation:}

for i in all nodes from inputs to outputs **do**

Assert a_i is computed

$W_i^c \leftarrow Clip(W_i, -1, 1)$

$s_i \leftarrow \varphi(a_i) W_i^c$

$b_i \leftarrow Binarize(s_i)$

end for

{2. Backward propagation}(Computation for gradients)

Compute gradient $g_{b_{out}} = \frac{\partial C}{\partial b_{out}}$ knowing $\boldsymbol{y}$ and b_{out}, $out \in \{output\ nodes\ to\ the\ circuit\}$

for i in all nodes from outputs to inputs **do**

Assert g_{b_i} is computed

$g_{s_i} \leftarrow g_{b_i} 1_{|s_i| \le 1}$

$g_{a_i} \leftarrow g_{s_i} g_\varphi W_i^c$

$g_{W_i^c} \leftarrow g_{s_i}^\intercal \varphi(a_i)$

$g_{W_i} \leftarrow g_{W_i^c} \circ 1_{|W_i| \le 1}$

end for

{3. Update parameters:}

for i in all nodes from inputs to outputs **do**

if W_i is trainable:

$W_i^{t+1} \leftarrow Update(W_i, g_{W_i}, \eta_i)$

end if

end for

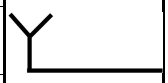
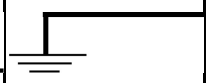
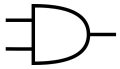
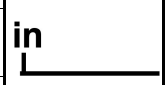
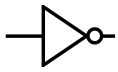
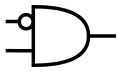
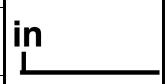
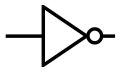
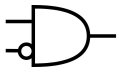
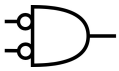
NPN equivalence class for Two-input gate and their template functions				
Name	**HIGH**	**LOW**	**AND**	**NAND**
Symbol				
Template f(x)	$f(\boldsymbol{x}) = 1$	$f(\boldsymbol{x}) = -1$	$f(\boldsymbol{x}) = x_1 + x_2 + x_1x_2 - 1$	$f(\boldsymbol{x}) = -x_1 - x_2 - x_1x_2 + 1$
Name	**BUF_1**	**NOT_1**	**AND2_NP**	**NAND2_NP**
Symbol				
Template f(x)	$f(\boldsymbol{x}) = x_1$	$f(\boldsymbol{x}) = -x_1$	$f(\boldsymbol{x}) = -x_1 + x_2 - x_1x_2 - 1$	$f(\boldsymbol{x}) = x_1 - x_2 + x_1x_2 + 1$
Name	**BUF_2**	**NOT_2**	**AND2_PN**	**NAND2_PN**
Symbol				
Template f(x)	$f(\boldsymbol{x}) = x_2$	$f(\boldsymbol{x}) = -x_2$	$f(\boldsymbol{x}) = x_1 - x_2 - x_1x_2 - 1$	$f(\boldsymbol{x}) = -x_1 + x_2 + x_1x_2 + 1$
Name	**XOR**	**XNOR**	**OR**	**NOR**
Symbol				
Template f(x)	$f(\boldsymbol{x}) = -x_1x_2$	$f(\boldsymbol{x}) = x_1x_2$	$f(\boldsymbol{x}) = -x_1 - x_2 + x_1x_2 - 1$	$f(\boldsymbol{x}) = x_1 + x_2 - x_1x_2 + 1$

Table 2. NPN equivalence class for two-input gate and the template simulation function for each gates. Simulation function is based on Model 2. For untrainable gates, the network will initialize the weights in reference to this table.

4. Experiments

In this paper, testing circuits are all raw circuits which are not optimized but simply generated by "gen" command of abc tool. [10] The original design for the circuit is saved in a BLIF (Berkeley Logic Interchange Format) file containing both connections and gates' type. The file is used for both generating input patterns and building structure of neural network. The detailed code of the experiment can be found on [11].

4.1 Data Usage

The algorithm feeds all patterns of data depending on the input number to the network within an epoch. In detail, during training process, the inputs are first shuffled and then all fed into the network with mini-batches. A whole set is considered as an epoch for the training. For example, as for a 5-input circuit, 2^5=32 input patterns are generated as training data. If batch size is set as 8, there are 4 mini-batches in an epoch covering all patterns.

4.2 Success Judgement and Equivalence checking

The success of the problem is guaranteed by a newly designed circuit that satisfies the specification. The circuit shall be composed of any reasonable types of gates for missing parts but remain completely the same for both the other gates and the topology. Under such guidance, a direct way to judge is to test over all the patterns for the network exhaustively and check correctness. It is practical for circuit with small scale inputs. However, computation grows exponentially as the input number grows. In the experiment, as we already have the original circuit which serves as the correct design that shall be expected, we can run combinational equivalence checking (CEC) for the new design and the original design. Due to the similarity of topology and consistency for most gates, equivalent nodes can be first evaluated and replaced, resulting in huge decrease in computation. In our experiment, we accept CEC provided by abc tool as a general way of verification. It is worth to mention that for strategy B, nodes along the fan-out path which are not supposed to have changeable behavior are validated at the same time. If such change is observed, the solution is classified as failure.

4.3 Small Circuit experiment

Two small circuits are chosen for the experiment, including a one-bit full adder with 3 inputs, 2 outputs and 11 gates and a two-bit full adder with 5 inputs, 3 outputs and 22 gates. Two models in section 2.3 with two strategies in section 3.3 are crossed paired and experiments are conducted respectively on the two circuits. For each experiment, certain number of unknown gates is given and the set of unknown gates is independently and randomly chosen. Initialization of the model for unknown gates are based on uniform initializer, providing different initial states even if two experiments have selected exactly the same set of unknown gates. As the circuit is relatively smaller, we set the maximum epochs as 10,000 in replace of time out.

From the table, we can easily find that success rate of finding unknown gates is higher for circuit 2 than circuit 1. This is common because the circuit 1 is much smaller, implying that within the same number of missing gates, circuit 1 holds more parts by percent of unknown information compared to known. Besides that, we can infer that in general kernel perceptron model

worked better than 2-1 nested perceptrons model. An assumption suggests that kernel perceptron grasps better feature of logic operation than nested perceptron structure due to its simplicity. 2-1 nested structure seemed to have a worse capability in learning the feature of non-linearity. Learning with Strategy A is slightly better than Strategy B, but the difference between two is not huge.

4.4 8-bit multiplier circuit experiment

A bigger circuit representing an 8-bit multiplier is chosen for the experiment. The original logic circuit contains 16 inputs, 16 outputs and 304 nodes with at most 40 levels. Only Model 2 is put into consideration for both strategy A and B while Model 1 never succeeded in retrieving gates when the number of unknown gates is larger than 5. Besides the success rate of retrieving unknown gates, average epochs required for a successful training is also calculated for each category. We ran 20 independent experiments for each this time and set 1 hour as the time limit. The result is shown in Table 5. The average epochs are given mainly for scalability evaluation for the problem, but serves no other purpose. The average scale of epoch increases along with the number of missing gates as expected. Since most of the success cases ran less than 5 epochs to find the result, it may reveal a hidden fact that finding an good initial state for weights is a possible optimization for faster training.

From the table, the success rate of retrieving unknown gates drops when the number of unknown is increasing. Model 2 with strategy A has a slight advantage against strategy B on success rate. The main reason is that stationary gates on the fan-out path of vacant gates may incrementally change behavior during the training, resulting in a wrong circuit.

Most failure cases occur when adjacent gates were chosen vacant. Such adjacent pair of two-input gates can be equivalent to the optimization of three-input gate. However, tackling such problem turned out to be NP-complete [12] from the perspective of symbolic logic. Other factors like wandering between local optimal and gradient cancelling caused by topological structure also leads to the collapse of training.

Model	Circuit	Number of missing gate				
		1	2	3	4	5
Model 1, Strategy A	Circuit 1	50/50	45/50	37/50	34/50	22/50
	Circuit 2	50/50	49/50	45/50	39/50	34/50
Model 1, Strategy B	Circuit 1	50/50	48/50 #	41/50	33/50	25/50
	Circuit 2	50/50	50/50 *	47/50	32/50	26/50
Model 2, Strategy A	Circuit 1	50/50	47/50	44/50 #	41/50 #	33/50
	Circuit 2	50/50	50/50 *	48/50 *	47/50 *	38/50 *
Model 2, Strategy B	Circuit 1	50/50	47/50	43/50	39/50	37/50 #
	Circuit 2	50/50	47/50	45/50	40/50	35/50

*Table 3. Success rate for different models and strategies on small circuits. Each category runs 50 independent experiments based on different number of missing gates. The # and * mark respectively stands for the best record for circuit 1 and circuit 2.*

Model		Number of missing gate								
		1	2	3	4	5	6	7	8	9
Model 2, Strategy A	Success rate	20/20	20/20	20/20	20/20	20/20	19/20	16/20	16/20	6/20
	Average epochs	1.7	3.5	1.2	1.4	2.1	6.7	10.1	10.8	26.1
Model 2, Strategy B	Success rate	20/20	18/20	20/20	20/20	19/20	18/20	15/20	12/20	10/20
	Average epochs	1.4	1.2	1.3	1.4	2.7	4.6	6.9	14.8	24.4

Table 4. Success rate and average training epochs to retrieve correct answer for 8-bit multiplier circuit. Model 2 with two different strategies are compared. The average epochs didn't count the failure cases in.

5. Discussion

This paper has presented a new idea of using neural network to solve partial logic synthesis problems. We inherit the same topology from the logic circuit to the network and replace gates with 2 types of perceptron model. We adopt discrete learning method from the intuition of binarized neural network. We have developed two partial training strategies and built template gates for untrainable gates. In the experiments, we simulated two small circuits and one relatively large circuit with different combination of models and training strategies. Some suspicious reasons that may be responsible for failure are discussed after the experiments. The new neural network method has turned out to be effective to solve partial logic synthesis problems, though from the viewpoint of state-of-the-art methods in partial logic synthesis, the neural network approach is still young and requires more experiments on larger circuits with higher success rate. Another expectation is to build hybrid method combining the QBF solver [2] with neural network. The latest QBF method only uses a small set of data which is iteratively given from equivalence checking. The input data that we utilize in the framework is so redundant that most data contribute nothing to the training. If we can derive a strategy in selecting efficient data, the training can be much faster and

effective. Seeking such hybrid method can also be interesting and beneficial to understand neural network.

Reference

[1] Masahiro F., Automatic correction of logic bugs in hardware design: Partial logic synthesis, Procedia Computer Science, Vol. 125, 2018.
[2] Masahiro F., Satoshi J., Shohei O., Takeshi M., Partial synthesis through sampling with and without specification, IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2013.
[3] Hratch M., Hiroaki Y., Andreas G., Shigeru Y., Masahiro F., On error tolerance and Engineering Change with Partially Programmable Circuits, The 17th Asia and South Pacific Design Automation Conference (ASP-DAC 2012), pp.695-700, 2012.
[4] Satoshi J., Takeshi M., Masahiro F., SAT-Based Automatic Rectification and Debugging of Combinational Circuits with LUT Insertions, Asian Test Symposium (ATS), pp.19-24, Nov. 2012.
[5] Alex K., Ilya S., Geoffrey E. H.: ImageNet Classification with Deep Convolutional Neural Networks. In NIPS, 2012
[6] Zaifu Z., Robert D. M., Witold P.: A neural network algorithm for testing stuck-open faults in CMOS combination circuits. Journal of electronic testing: Theory and Applications, 4, 225-235, 1993.
[7] Minsky M., Papert S., Perceptron: an introduction to computational geometry. The MIT Press, Cambridge, expanded edition, 19(88), 2.
[8] Amr S., Daniel G., Ulrich K., Mathias S., Rolf D., Formal Verification of Integer Multipliers by Combining Grobner Basis with Logic Reduction., EDAA, 2016.
[9] Courbariaux, M., Bengio, Y.: Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1., CoRR, 2016.
[10] https://people.eecs.berkeley.edu/~alanmi/abc/
[11] https://github.com/cainburster/gateNN.
[12] Avrim L. B., Ronald L. R.: Training a 3- node neural network is NP-complete, Neural Networks, Vol. 5, 1992.