

# Table of Contents

## Section I

The updated documentation of the project which is in remote collaboration with Jin Chaoyi, a Master Student under Prof. Masahiro Fujita, The University of Tokyo, Japan. This document is from his Rinko Presentation in the University this year 2019.

## Section II

The slides describing the work I started during my Summer Internship under Prof. Masahiro Fujita Summers 2018.

## Section III

The detailed documentation of the approach Studied and implemented during the course of the Internship.

# Partial Logic Synthesis through Partially Training a Directed Acyclic Graph structured Neural Network

有向非巡回グラフ構造化ニューラルネットワークの部分訓練による部分論理合成

藤田研究室 修士一年

37-185077 Jin Chaoyi

## Abstract

In this paper, we present an experimental technique for partial logic synthesis through partially training a topologically-similar neural network. Here, partial logic synthesis means that most parts of the target circuit are fixed whereas the missing portions must be logic synthesized from scratch. By constructing a similar Directed Acyclic Graph Neural Network structure from And-inverter Graph, which is a type of implementation of the logical functionality of a circuit, modeling gates with nested perceptrons and training network repeatedly, we converge the parameters inside and retain the possible missing gates. We first run an experiment on simulating a two-level gate with neuron-like computation. Then, we use the model from the first experiment to replace gates into small-scale circuits. We show successful results on retaining missing gates from the network.

## 1 Introduction

Logic debugging is a process that is performed after some kind of logical bugs are found in the designs through logic simulation or formal analysis of the designs. It mainly consists two parts: the first is to identify the buggy portions of the designs to be modified for correction, the second is to actually fix the buggy portions of the design so that the resulting circuits become correct. A series of work called path tracing [1] has been done to solve the former problem. The second problem can be interpreted as partial logic synthesis, which tries to correctly fill the vacant subcircuits previously considered as the buggy portions. [2] As shown in Figure 1, such trial is not to create completely new design, but to substitute the new subcircuits with buggy ones.

The problem can be divided into different of sub-questions depending on the type of transformations applied to the circuits. The simplest one is only to change the types of the gates while most topology of the circuit remains the same. One way to solve the problem is to formulate it as Quantified Boolean Formula (QBF) by substitute Look Up Tables (LUTs) with the vacant part. From previous work, the QBF problem can be solved by, e.g., QBF solvers [3], or repeatedly applying SAT solvers [4]. There are also situations that only changing types of gates cannot fully correct the circuit. A new formulation based on SAT is developed based on the conditions of the existence of logic functions with the given set of inputs by which the entire circuits can be corrected. [5]

In this paper, we provide a completely new method based on neural network techniques compared to previous work. We utilize the And-inverter graph, an acyclic graph interpretation of the logic circuit to provide a fixed topology for the neural network, and substitute gates with nested perception model created from simulation. Then, we deploy training techniques from deep learning to retain the vacant gates. From the result, we can see the model has been successfully retrieved missing gates for some small circuits.

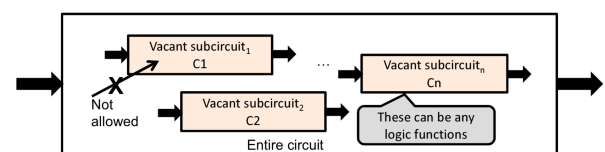


Figure 1. Partial logic synthesis Problem [2]

## 2 Topological model

### 2.1 Directed Acyclic Graph

In computer science and mathematics, a directed acyclic graph (DAG) is a graph that is directed in one way and without cycles connecting any pair of nodes. And-inverter Graph and the DAG structured neural network are two examples of DAG models, which share a common property inherited from DAG. And we will briefly explain what these two models are and raise the motivation of modeling circuit with deep learning techniques.

## 2.2 And-inverter Graph

An and-inverter graph(AIG) is a directed, acyclic graph that represents a structural implementation of the logical functionality of a circuit. From the perspective of logic gates, every gate in AIGs can be expressed in terms of AND gates and Inverters. In topological structure, an AIG consists of two-inputs nodes representing logical conjunction, terminal nodes labeled with inputs and outputs, and edges optionally containing negative logic for inverter.

As two-level gates have altogether 16 patterns, AIG represents eight types due to its structure. As shown in Table 1, those eight types accidentally occupy one class in NPN equivalence class of two-level gate. Class B and C in Table 1 don't occur in AIGs' expression because they are one or zero input actually. XOR/XNOR gates in Class D are usually interpreted in terms of three nested nodes according to AIG. A logic transformation of XOR gate will be expressed as,

$$a \oplus b = \overline{\overline{a} \overline{b} a b},$$

while using three and-inverter nodes.

The study of AIGs has been raised since late 1950s. But it was not until 1990s that new interest in AIGs as a functional representation in synthesis and verification was aroused. AIGs found successful usage in diverse EDA applications, especially in CMOS-based industry, as and-inverter gate costs fewer transistors compared to other gates such as or gate. Now, a mature software system for synthesis and verification called ABC supports operations on AIGs. [6] Figure 2. shows an example of graph generated by AIGs.

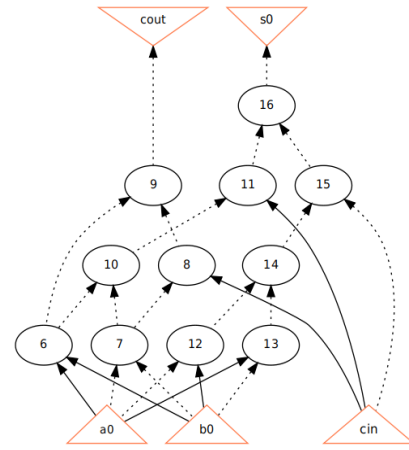


Figure 2. An example of AIG generated by ABC visualizer, representing for one-bit full adder in origin

NPN equivalence class for Two-input gate		
Class A		
Class B	Class C	Class D

Table 1. NPN equivalence class for Two-input gate, based on the idea that permuting inputs or adding inverter in either input or output stay in the same class. Gates generated by AIG only occur in Class A. Simulation for gates behavior in 4.1 covers for all types.

### 2.3 Gate model in DAG structured Neural network

Artificial neural network (ANN) accepts the idea of directed graph to model the data flow between information. Modern ANNs use back propagation as a successful training technique to process complex information such as image classification, audio recognition, etc.

DAG structured neural network [7] implies the topology of network contains no cyclic connection from input nodes to output nodes. Many famous Deep Learning models, such as LSTM, ResNet, GoogLeNet, etc. belong to this division. As both AIGs and DAG structured neural network belong to the same graph division, their topological structure can be shared and transplanted.

After accepting entire topology, nodes/gates in AIGs need to be replaced with specific structure in a neural network way. In machine learning, “perceptron” is an algorithm specialized for binary classifier. [8] If we set classification result as -1 and

1, a simple two-input perceptron can stand for a threshold logic as

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } w_1x_1 + w_2x_2 + b \geq 0 \\ -1 & \text{otherwise} \end{cases},$$

with 3 parameters to be trained inside. We first tried to use perceptron model to simulate two-level gate behavior, while unfortunately, the two-level logic gate has 16 possible patterns, 14 of which can be trained to be represented except XOR and XNOR gate. As shown in Figure 3, low level linear function used in perceptron is impossible to classify XOR or XNOR gate. To cope with limitation, we use a nested perceptron to interpret all the two-level gates. We show different types of nested perceptron in Figure 4. In our experiments, we only use 2-1 nested type (Figure 4b) for the sake of more compact design and less parameters. Interestingly, we find 2-1 nested structure is quite similar to what AIG interprets for XOR as we discussed in 2.2, though the inner nodes inside nested perceptron doesn’t have to follow a gate simulation restriction.

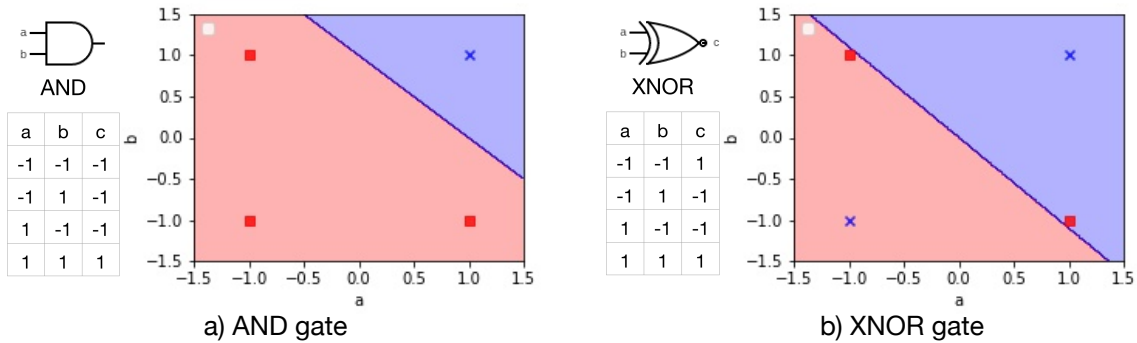


Figure 3. Fitting problem on XNOR gate compared with AND gate based on the simple perceptron model.

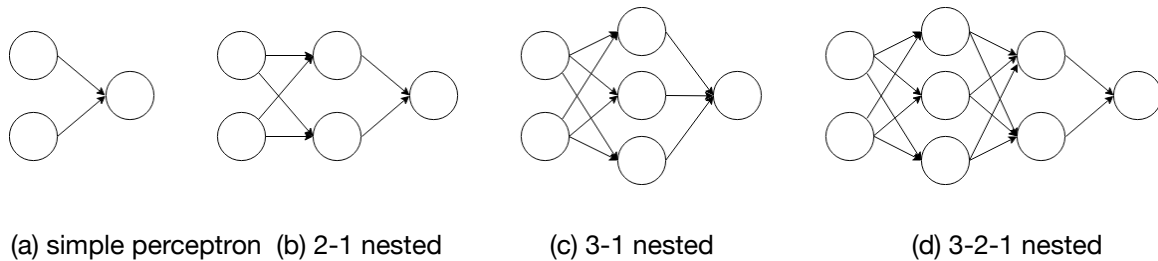


Figure 4. Different nested perceptron models to simulate the behavior of single gates. Use structure (a) for fixed gate to keep simple and compact. Use structure (b) for vacant gates to cover all possibility of two-input gates.

### 3 Training Methodology

#### 3.1 Discrete training

Building such a topological neural network is not so difficult, but training is far more painful. One difference from normal deep learning method is that all the activations computed by the nested perceptron should be binarized values because inputs and outputs of gates are discrete. If we allow activations of nested perceptrons to produce float or integer number, the whole network may converge at some hyperspace which does not exist in sign logic. We need to use methods that can train with binarized activations.

Discrete training methods is developed recently when people come up with the idea of compressing the state of the art model in deep learning. Binarized neural network (BNN) [10] use both binary value for weights and activations during training. In forward computation, the activations are computed by a sign function (deterministic binarization function),

$$x^b = \text{Sign}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

where  $x^b$  is the binarized variable and  $x$  is the original real-value variable. When calculating gradients during back propagation, tanh function [11] is commonly used,

$$\tanh(h) = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

since it's differentiable when calculating gradients. More commonly, people use Htanh as an estimation of tanh function to avoid the heavy computation of exponential function. The Htanh function is shown as follows,

$$\text{HTanh}(h) = \begin{cases} +1 & x \geq 1 \\ x & -1 \leq x \leq 1 \\ -1 & x \leq -1 \end{cases}$$

A comparison of all the three functions and their derivate functions is shown in Table 2. We prefer using Htanh because it always gives 1 as activated

gradient during back propagation, which is beneficial for fixed gates to pass loss backward. However, we also use tanh as activation function inside the nested perceptron which provides various scaling value.

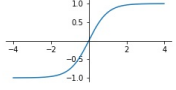
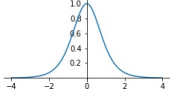
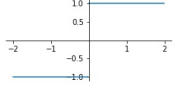
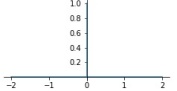
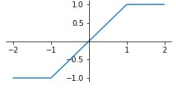
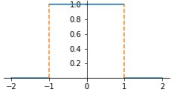
Function	Function plots	Derivative plots
Tanh(x)		
Sign(x)		
Htanh(x)		

Table 2. Plots on function used in discrete learning and their corresponding derivative function.

#### 3.2 Template initialization

When we finally start training, all the parameters inside the network are required to be initialized. As the training process will pass through all nodes from inputs to outputs, we need to choose fancy initializer for all parameters.

As the determined nodes are generated from AIG, we know that only 8 types need to be simulated from 2.2. To simplify, we just adopt the single perceptron structure (Figure 4a) to replace fixed gates, within least parameters and most compact topology. The three parameters in the Equation in 2.3 can be easily determined for 8 types of gates, and can be shared for initializing fixed gates.

As for the perceptrons to be trained, standing for vacant gates, lots of trials are approached but the values may still fall into death loop anyway. However, finding a good initialization in deep learning is always a paining task and hard to find a template method. This will still remain to be an open topic and won't briefly discussed in this paper.

#### 3.3 Partial training

In order to save training time and to maintain the non-target perceptrons' behavior, the parameters describing fixed gates are not recommended to be trained. That introduces the idea of partial training, which means only parts of the parameters in the neural network will be trained. According to the topology of neural network, we give two training methods.

The first method is to train the vacant gates as well as all the gates by which those vacant gates' activations flow. As shown in Figure 5, nodes No. 10 is an unknown gates ready to be trained, all the nodes on the route from No. 10 to output prediction, nodes 11, 16 and s0, are set to be trained. Literally, the training cannot guarantee those nodes stay with the same behavior as they were initialized, which is not supposed to happen. A safe method is to add a Gate Agent on the route. If type of gates changes during each update, the parameters inside will be reinitialized. However, it is not a practical way to check every time after updating, which is too time-consuming. Checking with some gaps or using strong parameters that are difficult to result in gates changing is more reasonable.

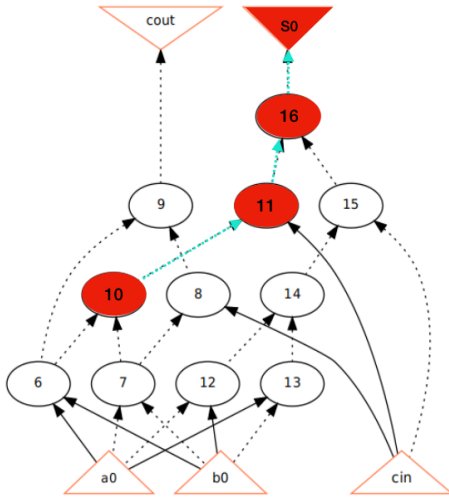


Figure 5. Example of first partial training method. Red painted nodes are the ones needed to be train. Node 10 is the target vacant gate. On the trail from Node 10 to output, all the nodes need to be trained.

The second method is only to train the vacant gates while remaining all other parameters permanent. This method works for most gates inside the circuit

except for the occurrence of XOR/XNOR-like structure in the path to the output. Taking XOR as example, AIG interprets it with 3 nodes according to 2.2. If activations of some training perceptrons flow into the existence of XOR/XNOR gate, the loss from the end may encounter self-extinguishing during the back propagation, leading the gradients passing to the training parts to 0. This happens because of the two former nodes of XOR gate in AIG behave a symmetric structure. If gradients from the two gates are restricted as -1 and 1, loss will be cancelled at either of the former nodes, regardless the nodes further ahead. A effective adjustment is to keep the former two nodes trainable just like the first method. After some updates, the symmetric cancelling structure will be spoiled and training can be available.

## 4 Experiments

### 4.1 Two-level gate simulation

As discussed in 2.3, the nested perceptron topology needs to be checked to show that all 16 gates for two-level input can be well simulated. Adopting the 2-1 nested perceptron, we do fitting experiment for all types of gates shown in Table 1, training from restricted random parameters. Here, "restricted" means range and distribution of initialized values are carefully chosen. In the experiment, the weights in first two-size layer are uniformly chosen from 0.1 to 0.2, bias to 0, activation function using tanh. In the second one-size layer, the two weights are respectively set as -1 and 1, bias as 0, activation function as sign function in forward and Htanh in backward with another comparing group just use Htanh for all. Optimizer used simple SGD with momentum.

We ran 1000 independent cases for each gates, both in discrete training method and normal training method. The success rate of reaching a correct classification and average epochs trained for each gates are summarized in Table 3. Due to the phenomenon we discovered that gates in the same NPN equivalence class produce similar training process. We classify the data with four types according to Table 1. From the result, all



Training method		NPN equivalence class			
		Class A: And-inverter gates	Class B: High/Low	Class C: Buffer/Inverter	Class D: XOR/XNOR
Discret	Success rate	8000/8000	2000/2000	4000/4000	2000/2000
	Training epochs	2.614	1.0	0.760	36.462
Normal	Success rate	8000/8000	2000/2000	4000/4000	2000/2000
	Training epochs	2.333	0.967	0.917	38.239

Table 3. Gate simulation for 16 gates using 2-1 nested perceptron structure. Each gates run 1000-time independent training experiment, with random initialized value. Training and validation use the same dataset containing all 4 patterns for two-input.

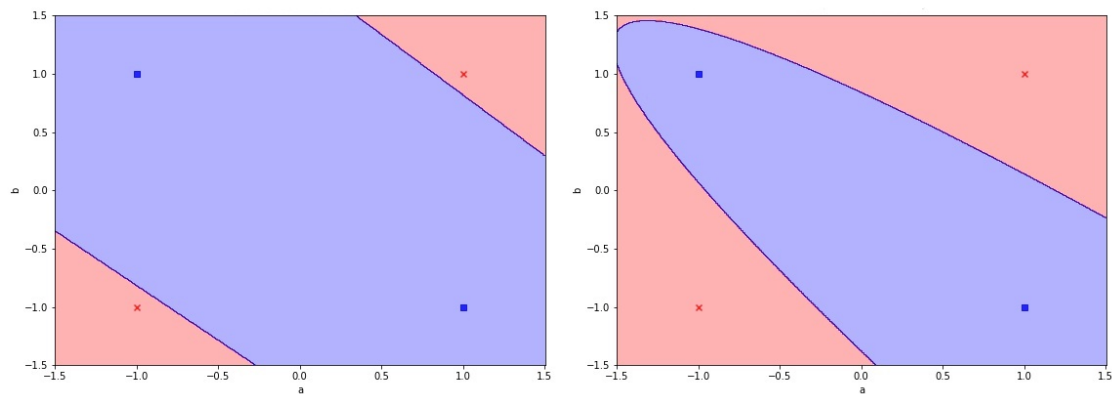


Figure 6. Final decision map for training XOR using 2-1 nested topology. The left one uses normal training method while the right one uses discrete training method.

experiments have a correct logic classification for corresponding gates with random initialization. The XOR and XNOR gates have the slowest converging speed for both discrete and normal training, while discrete one trains a little bit faster.

We also discover the decision map through training process for both training method on XOR gates as shown in Figure 6. Interestingly, case with discrete method produces some flip-flops and the final map is usually not so symmetric compared to normal case. This phenomenon doesn't have destructive influence for small circuit, but as the size of the circuit increases, the training may suffer a struggling moment in the swamp of local optimization. And I still insist that reaching a fancy classification map and more stable training process helps train in larger cases.

#### 4.2 Small circuits simulation

To do concrete circuit experiments, we first need a set of template parameters for fixed perceptrons according to 3.2. Then, in the experiment, the circuits we tested are one-bit full adder and two-bit full adder, which respectively represent for 3-in-2-out-11-gate and 5-in-3-out-22-gate scale. The corresponding topology is derived from the AIG representation of each circuits and we tested with the second method we discussed in 3.3. We set the number of the missing gates from 1 to 4 and run 50 times randomly for circuits. As the dataset is not very large, we use size of dataset for batch size and train all data during each iteration, which is 8 for one-bit full adder and 32 for two-bit full adder. And when the model can completely fit all input pattern during validation, we consider it as a successful training.

Logic circuit		Number of missing gates			
		1	2	3	4
one-bit full adder 3 inputs, 2 outputs, 11 gates	Success rate	50/50	47/50	47/50	41/50
	Training epochs	4.3	20.0	36.9	294.3
two-bit full adder 5 inputs, 3 outputs, 22 gates	Success rate	50/50	50/50	49/50	47/50
	Training epochs	4.4	14.3	34.9	307.6

*Table 4. Results of simulation on small circuits. Use all input pattern for both training and validation respectively. If the model can fits all input pattern at some degree, we consider the training success.*

In Table 4, we summarized the success rate of retrieving the correct gates and the average epochs spent for each circuit. We can know that for small number of missing gates, we can retrieve the right answer easily. And if the scale of circuit is larger, the upper limit of the missing gates will be higher.

About the failure ones, a common case is nested gates are chosen from the random choice. Literally, it means we are trying to fit some 3 input cases or 4 inputs cases. According to [12], training a n-input logic is proved to be NP-Complete, which makes training harder.

One thing needs to be mentioned is that the circuits we tested is relatively small, which is possible to throw all input patterns into validation. However, in larger circuits, we are impossible to try all input patterns, but instead using a small set of inputs which is crucial for judging success or not. And a piece of even more valuable information is that as the topology of circuit is fixed, a lot of input patterns are actually not helpful for training. That implies we can do training with a smaller but more effective dataset as well.

## 5 Conclusion

This paper represented a new idea of using deep learning techniques to solve partial logic synthesis problems. We used the similarity between AIG and DAG structured neural network to deploy a topologically similar network. Then, we adopted a nested perceptron model to simulate the behavior of

gates and gave solution of how to train such abnormal network - discrete training and partial training. In the experiments, we simulated how nested perceptron model two-input gates and ran small circuits simulation. We have confirmed that such methods can be used to solve partial synthesis question and got a high possibility of success of training.

## 6 Future Research

As the experiments show that it's possible to solve such problem using neural network. We now focus on how to train the model with higher success rate and more stable process. First thing that we haven't realized in our work is a Gate Agent discussed in 3.3. And then we can confidently try the first partial training method and compare the two methods. Secondly, we still need to find a better initialization for the parameters of perceptrons. [13] This work is tightly related to the experiments of gates simulation and try to find the best and most stable template parameters for gates. Another work focuses on larger circuits simulation and decrease the size of dataset for both training and validation. Last but not least, improve the efficiency of computing is necessary because we need to compare it with previous methodology. Approaches can be tried such as rewriting the topology into stacked layers within sparse matrix, which allows the model to run training parallelly.

## 7 Reference



- [1] M.S.Abadir, J.Ferguson, T.E.Ferguson: Logic Verification via Test Generation, IEEE Trans. on Computer-Aided Design, Vol. 7, pp. 138-148, January 1988.
- [2] Masahiro Fujita, Automatic correction of logic bugs in hardware design: Partial logic synthesis, Procedia Computer Science, Vol. 125, 2018.
- [3] <http://beyondnp.org/pages/solvers/qbf-solvers/>
- [4] Mikolas Janota, William Klieber, Joao Marques-Silva, Edmund M. Clarke: Solving QBF with Counter example Guided Refinement, SAT2012, pp.114-128, 2012.
- [5] Amir Masoud Gharehbaghi, Masahiro Fujita: A New Approach for Debugging Logic Circuits without Explicitly Debugging Their Functionality. ATS, Nov. 2016.
- [6] <https://people.eecs.berkeley.edu/~alanmi/abc/>
- [7] P. Frasconi, M. Gori, A. Sperduti: A General Framework for Adaptive Processing of Data Structures, IEEE Transactions on Neural Networks, Vol. 9, Sep. 1998.
- [8] Minsky, M. Papert, S. Perceptron: an introduction to computational geometry. The MIT Press, Cambridge, expanded edition, 19(88), 2.
- [9] Vinicius P. Correia: Classifying n-Input Boolean Functions.
- [10] Courbariaux, M., Bengio, Y.: Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. CoRR, 2016
- [11] G. Hinton, Neural Network for Machine Learning, Courser, 2012.
- [12] Avrim L.Blum, Ronald L.Rivest: Training a 3-node neural network is NP-complete, Neural Networks, Vol. 5, 1992.
- [13] Dmytro Mishkin, Jiri Matas: All you need is a good init, ICLR 2016.

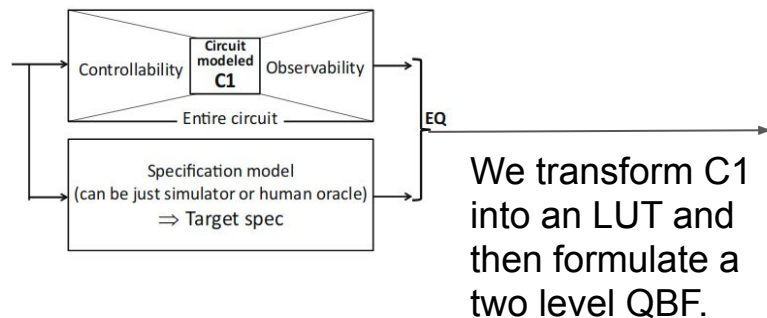
# Partial Logic Synthesis using Deep Neural Networks

Sudarshan Sharma  
Intern, Fujita Lab,  
Senior Undergraduate ,  
Dept. of Electronics and Electrical Comm. Eng.  
IIT Kharagpur, India.  
Email-sudarshansharma04@gmail.com  
-sharma@cad.t.u-tokyo.ac.jp

# Outline

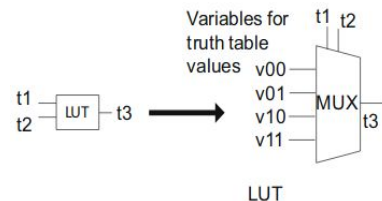
- Partial Logic Synthesis
- Approach using Binarized Neural Network(BNN)
- Construction of BNN
- Previous works on constructions.
- I. Hubara et.al Binarized Neural Network.
  - Training Algorithm Explained
  - Experiments
- Ternary Neural Networks
  - Previous Works and Algorithms
  - Experiments
- Conclusions and Discussions

# Partial Logic Synthesis



$$\exists v. \forall x. f(v, x) = SPEC(x)$$

Where SPEC is the logic function that represent the specification to be implemented and v defines the truth tables for the LUT



Furthur,

- We choose some values of x out of the possible  $2^N$ , the equation above becomes a SAT problem. This is just a necessary condition(N is the number of input variables)
- The solution obtained from the SAT solver is checked through the equivalence checker.
- If equivalence proved, We are Done
- If not equivalent counter example generated, which is used as for input variable in the next iteration.

# Binarized Deep Neural Network Basics

## What are Binarized Deep Neural Networks (BNN) ?

BNN are a class of Neural Networks with Binary weights, activation and biases(if used). It is said that the DNN achieves the goal by learning a hierarchy of features in their multiple layers.

BNN were developed with the aim:-

- Improvement in power efficiency.
- Reduction in memory size (storage)
- Faster access as every operation can be replaced with bitwise Operation. (Less Memory Bandwidth)

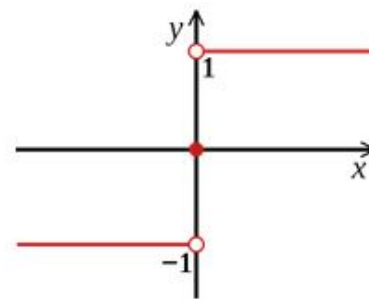
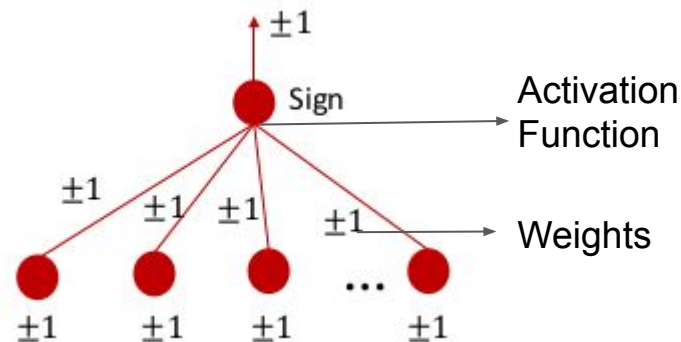
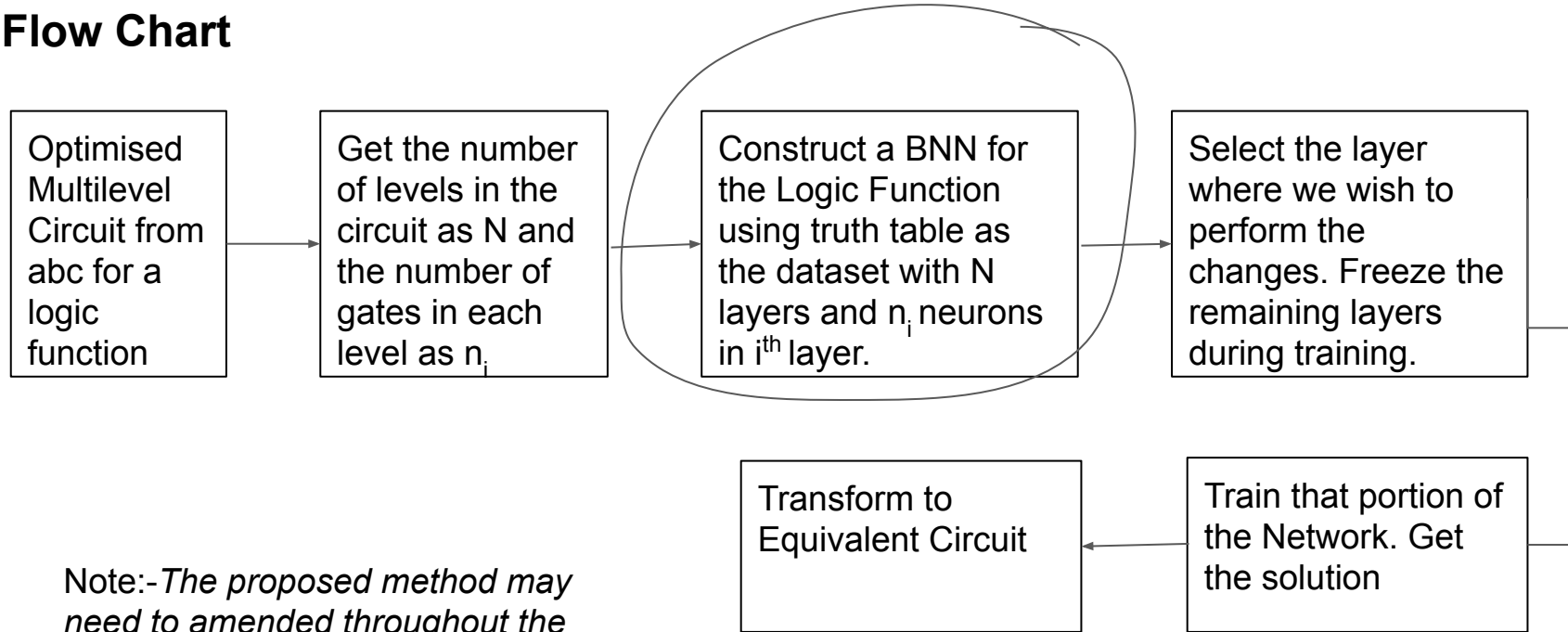


Fig:Signum  
Activation  
Function

# Present Approach using Deep Neural Network

## Flow Chart



*Note:- The proposed method may need to be amended throughout the experimentation phase depending on the limitations*

# Construction of BNN

## Previous Work

1. R. Kohut et.al Boolean Neural Network. (2004)<sup>[1]</sup>
2. Y. Nakayama et.al A Simple Class of Binary Neural Network and Logical Synthesis. (2011)<sup>[2]</sup>
3. M. Kim et.al Bitwise Neural Networks. (2016)<sup>[3]</sup>
4. I. Hubara et.al Binarized Neural Network. (2016)<sup>[4]</sup>

*1. Kohut, R.; Steinbach, B.: Boolean Neural Networks. Transactions on Systems, Issue 2, Volume 3, April 2004, pp. 420 – 425.*

*2. Yuta NAKAYAMA, Ryo ITO, Toshimichi SAITO, A Simple Class of Binary Neural Networks and Logical Synthesis, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Released September 01, 2011*

*3. M. Kim et.al, " Bitwise Neural Network, "arXiv:1601.06071, Jan 2016*

*4. I. Hubara et.al, "Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1," arXiv:1602.02830 Mar 2016.*



# R. Kohut et.al Boolean Neural Network. (2004)

To understand the NN model described first we need to know about different training techniques. These are divided into two broad categories:-

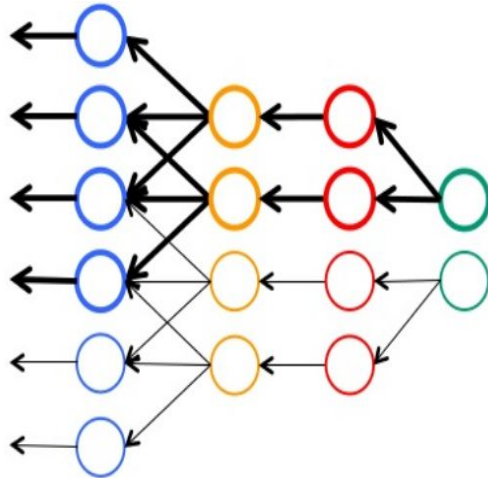


Fig:-Back-Propagation  
Visualisation

## Iterative Learning Algorithms

- The network structure is **fixed**
  - The connection weights and thresholds are adjusted in the parameters space.
  - The training proceeds by **decreasing the errors** between model outputs and targets
- Eg:-Backpropagation, Radical Basic Function.

## Non-Iterative or Sequential Learning Algorithms

- Adds** hidden layers and hidden neurons during training
  - Assure **fast converges** than the Iterative learning algorithms.
  - Lesser training time** than the Iterative once.
- Eg:-Functional on tabular Function Set(FTFS), Expand and Truncate Learning (ETL)

# R. Kohut et.al Boolean Neural Network. (2004)

- The models uses **Sequential Learning Algorithms** which is **Functional on Tabular Function Set(FTFS)** to construct the BNN.
- The motivation is more for the **representation** of the Boolean Function
  - By reduction in space variable
  - Reduces time for converting Input Vector to Output Signals.
- Special decomposition of the Boolean Function during the training yields simpler unique base function. i.e. **efficient intermediate logics**
- The algorithms proposed validates only for **single layered BNN**.
- Works fine for **Multi-output Logic Functions**.
- Entire Truth Table is required for Training.

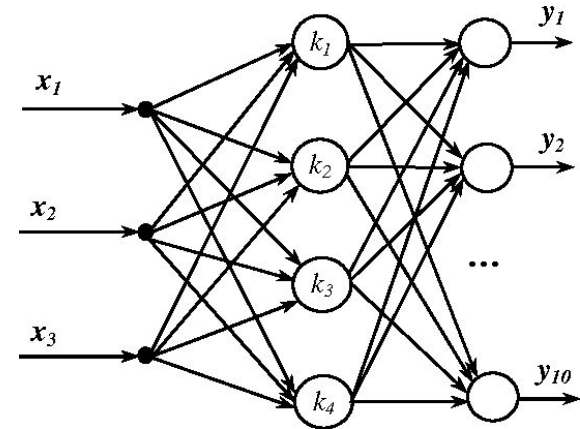


Fig:-BNN Model from the paper

# Y. Nakayama et.al A Simple Class of Binary Neural Network and Logical Synthesis.(2011)

- Genetic Algorithm based Learning Algorithm(GALA).**
- A **single layered** architecture proposed.
- They say that the GALA always **look for global optimum** and can operate speedily and **do not require differentiability** of objects.
- It is more of an **alternative for the Disjunctive Canonical Form (DCF) i.e canonical DNF** calculation for larger Boolean Function.
- It says that if the number of **hidden neurons are decreased** from the optimal number of terms in DNF the Error Tolerance Rate(**ETR**) **increase**.
- Weights are ternary in nature.  $\{-1,0,+1\}$ .

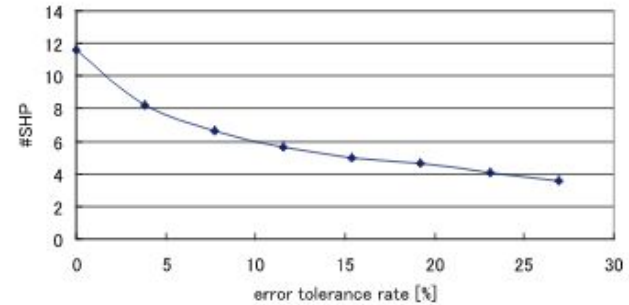


Fig:-Effect of ETR on the number of hidden neurons for a logic function

# M. Kim et.al Bitwise Neural Networks. (2016)

Open Source  
Codebase  
Not Available



404

Page not found

- Main aim was to lower down the **spatial complexity, memory bandwidth and power consumption** in the hardware.

- Two way training approach

## 1-Weight Compression

It helps the real valued model to easily be converted into a BNN.

## 2-Noisy Backpropagation

The compressed weights used for noisy backpropagation based on bitwise parameter.

- Update real-valued weights** during training and **then binarize** again.

- Multilayered** support

- Results tested on **big images and audio dataset**.

- A **better alternative** according to our choice.

# I. Hubara et.al Binarized Neural Network(2016)

Entire Codebase is available on  
github in popular DNN frameworks



- Torch
- Theano
- Tensorflow
- pyTorch

For Images

- Training Algorithms to train the **binary weights and algorithms at runtime**

- Computes **parameter gradients at train time.**

- Main aim for **reduction in memory size, access and improvement in power efficiency.**

- Results based on **state-of-the-art big images dataset.**

- This seems to be the **best alternative** as it stand perfectly according to our requirement.

- And one more thing.

# I. Hubara et.al Binarized Neural Network(2016)

Experiment based on this Algorithm.  
Let's get in details!

# I. Hubara et.al Binarized Neural Network(2016)

## Binarization Function

The weights and activation are **both constrained to +1 or -1**. Two sets of alternatives proposed by Courbariaux et.al 2015.

### Deterministic Binarization

$$x^b = \text{Sign}(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{otherwise,} \end{cases}$$

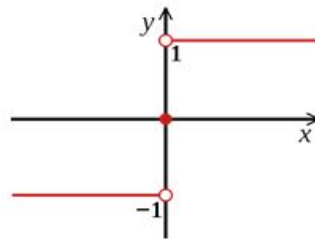


Fig:-Signum Activation

### Stochastic Binarization

$$x^b = \begin{cases} +1 & \text{with probability } p = \sigma(x), \\ -1 & \text{with probability } 1-p, \end{cases}$$

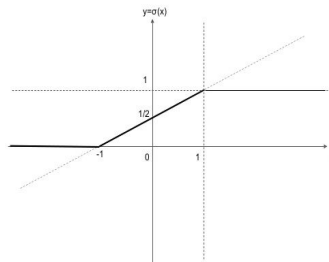


Fig:  $\sigma(x)$  also called hard sigmoid



# I. Hubara et.al Binarized Neural Network(2016)

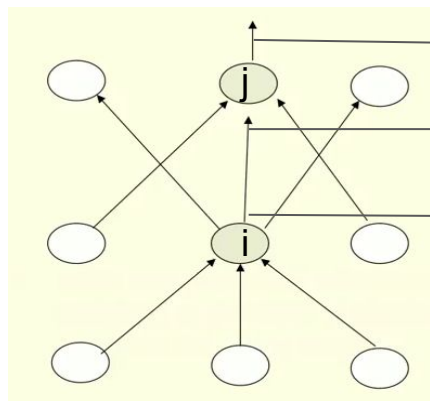
## Gradient Calculation

- The BNN training algorithm **uses the binary weights and activation** to compute the **parameter gradients**.
- However, the **real valued gradients** are accumulated in **real-valued variables** which are **binarized** using the binarization function discussed earlier.

## Training Algorithm

- A modified **Back-propagation** algorithm used. (discussed in details later) **Computes Gradients**
- Stochastic Gradient Descent optimiser** used which uses the gradients obtained using the Back-propagation Algorithms to update the weights efficiently to minimise the loss function. **First order Optimiser**

# Normal Backpropagation Algorithm Review



Output of the jth layer  
let say  $y_j$

The input to the jth  
layer say  $z_j$

Output of the ith  
layer say  $y_i$

Let  $E$  be the cost function or the loss  
Function say **Mean square loss**.

$$E = \frac{1}{2} \sum_{j \in \text{output}} (t_j - y_j)^2$$

Where  $t_j$  is the target  
output.

Writing the relation between term using basic definition.

$$y_j = f(z_j) \quad , \text{where } f(x) \text{ is the activation function}$$

$$z_j = \sum w_{ij} * y_i \quad , \text{by definition}$$

$$\frac{\partial E}{\partial z_j} = \frac{dy_j}{dz_j} \frac{\partial E}{\partial y_j} \quad , \text{using chain rule}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial E}{\partial z_j} = y_i \frac{\partial E}{\partial z_j}$$

, using chain rule, the gradient obtained is used to **updates the weights** from the  $i^{\text{th}}$  to  $j^{\text{th}}$  layer during training. For the subsequent layers the hidden layers below  $i^{\text{th}}$  layer we need the following gradient.

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i} \frac{\partial E}{\partial z_j} = \sum_j w_{ij} \frac{\partial E}{\partial z_j}$$

, using chain rule, this gradient is useful when we need  $\frac{\partial E}{\partial z_i}$  Which is then used to get the weight gradients

# Why Modified Back Propagation Algorithm?

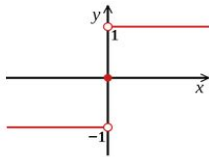
We get to know that if we get the  $\frac{\partial E}{\partial z_j}$  gradient we can calculate the weight gradients and the remaining useful gradients for Back propagation to work.

Let's look at this again

$\frac{\partial E}{\partial z_j} = \frac{\partial f(z_j)}{\partial z_j} * \frac{\partial E}{\partial y_j}$  , where  $f(x)$  is the activation function. So for binarized weights and activation we wish the activation function to be  $\text{Sign}(x)$ .

If  $f(z_j) = \text{Sign}(z_j)$

$$\frac{\partial E}{\partial z_j} = 0$$



Thus, the learning would never converge and weights won't be updated through the general back propagation algorithm.

# Modified Backpropagation Algorithm

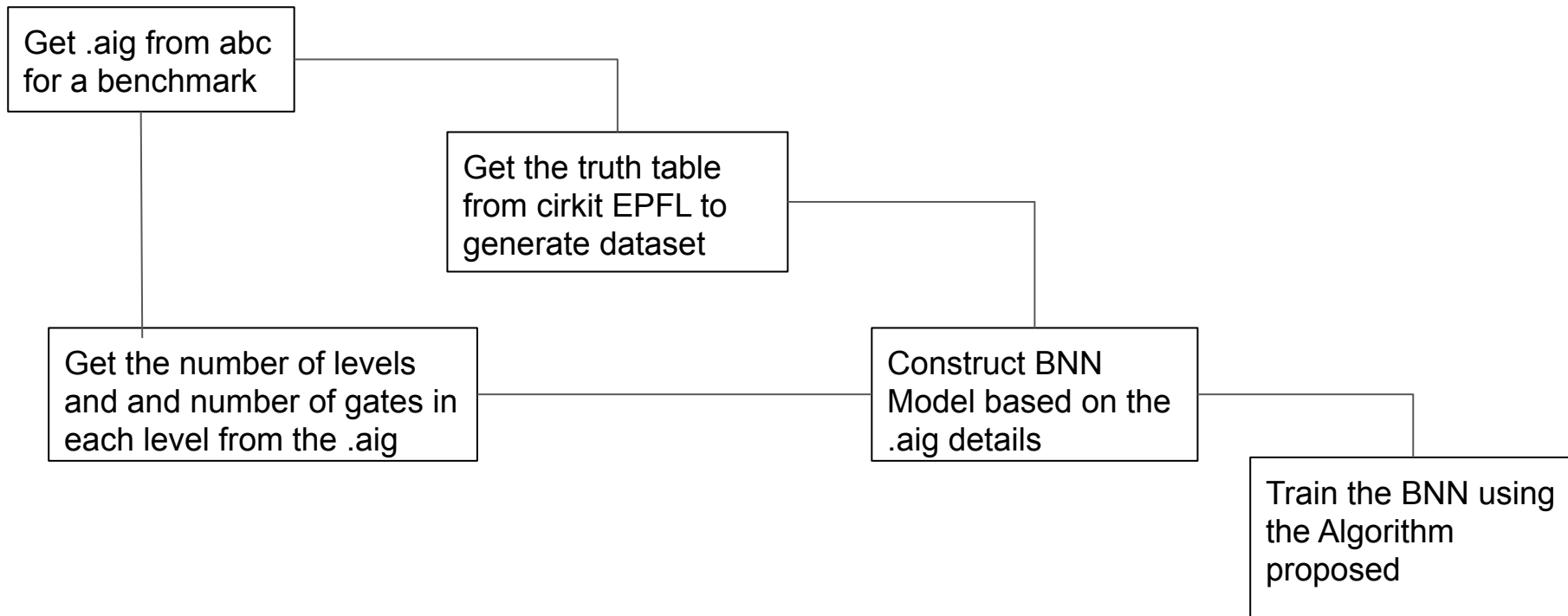
Hinton et.al in his lectures proposed the idea of **straight through estimators** to mitigate the problem which can be seen as a hard tanh function.

$$\text{est}(\text{Sign}(x)) = \text{Htanh}(x) = \begin{cases} +1 & , x > 1 \\ x & , -1 \leq x \leq 1 \\ -1 & , x < -1 \end{cases} \quad \text{est} \rightarrow \text{straight through estimator}$$

$$\text{est}\left(\frac{\partial E}{\partial z_j}\right) = \frac{\partial \text{Sign}(z_j)}{\partial z_j} * \frac{\partial E}{\partial y_j} = \begin{cases} \frac{\partial E}{\partial y_j} & -1 \leq z_j \leq 1 \\ 0 & \text{otherwise} \end{cases} \quad \text{where} \quad E = \frac{1}{2} \sum_{j \in \text{output}} (t_j - y_j)^2$$

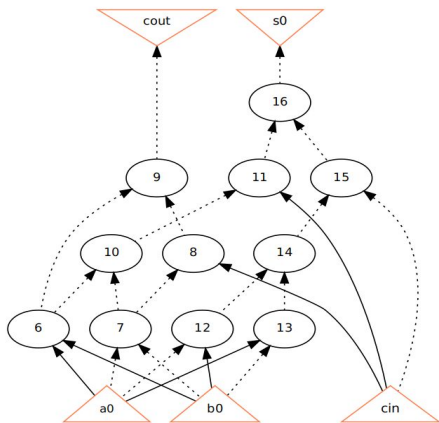
This simply means that during training to pass the gradients using the back-propagation algorithms use the **Htan(x)** as the activation function of the neuron.

# Experimental Setup for Construction of BNN



# I.Experiment with 1-Bit Adder

## AIG for 1-Bit Adder



## AIG

Number of levels=4

Number of gates in level=[4,3,3]

Number of input=3

Number of outputs=2

## BNN Model

Number of hidden layers=4

Number of neurons in layer=[4,3,3]

Number of inputs=3

Number of outputs=2

**Accuracy=87.5% (Wrong Set, Not Unique)**

But a random tweak in the number of neurons in the layers like **[4,3,3,3]** gives **100 % Accuracy** with the BNN model discussed.

However, this heuristic doesn't work out with 2-Bit Adders. We need some more amendments.

# Problems and Probable Solutions

We need to address the key differences in the AIGs and the DNN.

1. **Jumps** from one level to another are predominant in AIGs
  - No Jumps from layers in DNN.
  - Use a dummy neuron
2. The **Fan-in** of the gates is always two.
  - The fan-in of a neuron in  $(n+1)$ th layer is  $n$ .
  - Pruning less important connection
3. The **output node** may or may not be inverted in AIGs.
  - Can be replicated by adding one more layer in the DNN.

-To solve these issue, a **ternary weight DNN** with weights as  $\{-1,0,+1\}$  can be used where 0 depicts pruning of the inter-connection.

-One Neuron **unable** to implement the **logic functionality** important to represent a function.



# Ternary Weighted Neural Network

The discrete valued DNN are introduced these days with an **implication of lower power efficiency and lower memory bandwidth**. The weights are  $\{-1,0,+1\}$ , where **0 means no connections**.

However, these training algorithms are **tested on Images** and used more the on **Classification Problems**.

## Previous Works

1. F.Li, B. Zhang, B Liu, Ternary weight networks (2016)
2. Lei Deng et.al GXNOR-Net: Training deep neural networks ternary weights and activations without full-precision memory under a unified discretization framework. (2018)
3. H Alemdar et.al Ternary Neural Network for Resource-Efficient AI Applications (2017)

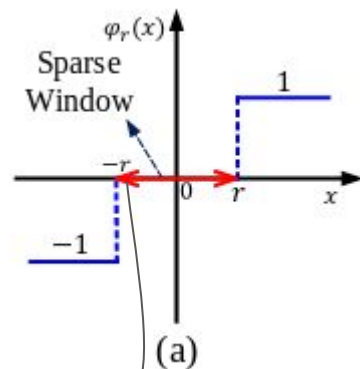
*1.F.Li, B. Zhang, B Liu, Ternary weight networks, arXiv preprint arXiv:1605:04711,2016 (2016)*

*2.Lei Deng et.al GXNOR-Net: Training deep neural networks ternary weights and activations without full-precision memory under a unified discretization framework. (2018)*

*3.H Alemdar et.al Ternary Neural Network for Resource-Efficient AI Applications (2017)*

# F.Li et.al Ternary Weight Neural Network(2016)

- Only **weights** are constrained to  $\{-1,0,+1\}$ , activation i.e the **output** of neurons is **binary**  $\{+1,-1\}$ .
- The main aim is to **minimise the Euclidian distance** between full precision weights and the ternary weights using a scaling factor.
- The threshold is obtained using **discrete optimisation** using approximation.
- Training Algorithm similar to **BNN model** as discussed earlier. This work is more of an **extension** to I.Hubara et.al Binarized Neural Networks.
- Lets see the Algorithm in details.



Threshold

Fig: Ternarisation Function

# F.Li et.al Ternary Weight Neural Network(2016)

Main aim is minimising the **Euclidean distance** between the full precision weights and the ternary-valued weights.

$$\begin{cases} \alpha^*, \mathbf{W}^{t*} = \arg \min_{\alpha, \mathbf{W}^t} J(\alpha, \mathbf{W}^t) = \|\mathbf{W} - \alpha \mathbf{W}^t\|_2^2 \\ \text{s.t.} \quad \alpha \geq 0, \mathbf{W}_i^t \in \{-1, 0, 1\}, i = 1, 2, \dots, n. \end{cases}$$

This gives the **optimisation function**, where  $J(\cdot)$  is the cost function

The optimal solution can be achieved through a threshold based ternary function.

$$\alpha^*, \Delta^* = \arg \min_{\alpha \geq 0, \Delta > 0} (|\mathbf{I}_\Delta| \alpha^2 - 2 \left( \sum_{i \in \mathbf{I}_\Delta} |\mathbf{W}_i| \right) \alpha + c_\Delta)$$

Using Differentiation

$$\alpha_\Delta^* = \frac{1}{|\mathbf{I}_\Delta|} \sum_{i \in \mathbf{I}_\Delta} |\mathbf{W}_i|.$$

Substituting

$$\Delta^* = \arg \max_{\Delta > 0} \frac{1}{|\mathbf{I}_\Delta|} \left( \sum_{i \in \mathbf{I}_\Delta} |\mathbf{W}_i| \right)^2$$

Now, this equation can be solved using discrete optimisation, but it time consuming, instead it is **approximated** considering the  $\mathbf{W}_i$  follow **Normal Distribution**.  $\Delta^* \approx 0.7 \cdot E(|\mathbf{W}|)$

$$\text{where the threshold function } \mathbf{W}_i^t = f_t(\mathbf{W}_i|\Delta) = \begin{cases} +1, & \text{if } \mathbf{W}_i > \Delta \\ 0, & \text{if } |\mathbf{W}_i| \leq \Delta \\ -1, & \text{if } \mathbf{W}_i < -\Delta \end{cases}$$

$$c_\Delta = \sum_{i \in \mathbf{I}_\Delta} \mathbf{W}_i^2, \quad \alpha \text{ dependent constant}$$

$$\mathbf{I}_\Delta = \{i | |\mathbf{W}_i| > \Delta\} \quad |\mathbf{I}_\Delta| \text{ denotes the number of elements in } \mathbf{I}_\Delta$$

# F.Li et.al Ternary Weight Neural Network(2016)

Main aim is minimising the **Euclidean distance** between the full precision weights and the ternary-valued weights.

$$\begin{cases} \alpha^*, \mathbf{W}^{t*} = \arg \min_{\alpha, \mathbf{W}^t} J(\alpha, \mathbf{W}^t) = \|\mathbf{W} - \alpha \mathbf{W}^t\|_2^2 \\ \text{s.t.} \quad \alpha \geq 0, \mathbf{W}_i^t \in \{-1, 0, 1\}, i = 1, 2, \dots, n. \end{cases}$$

This gives the **optimisation function**, where  $J(\cdot)$  is the cost function

The optimal solution can be achieved through a threshold based ternary function.

$$\alpha^*, \Delta^* = \arg \min_{\alpha \geq 0, \Delta > 0} (|\mathbf{I}_\Delta| \alpha^2 - 2 \left( \sum_{i \in \mathbf{I}_\Delta} |\mathbf{W}_i| \right) \alpha + c_\Delta)$$

where the threshold function  $\mathbf{W}_i^t = f_t(\mathbf{W}_i|\Delta) = \begin{cases} +1, & \text{if } \mathbf{W}_i > \Delta \\ 0, & \text{if } |\mathbf{W}_i| \leq \Delta \\ -1, & \text{if } \mathbf{W}_i < -\Delta \end{cases}$

$c_\Delta = \sum_{i \in \mathbf{I}_\Delta} \mathbf{W}_i^2$ ,  $\alpha$  dependent constant

$\mathbf{I}_\Delta = \{i | |\mathbf{W}_i| > \Delta\}$   $|\mathbf{I}_\Delta|$  denotes the number of elements in  $\mathbf{I}_\Delta$

Using Differentiation

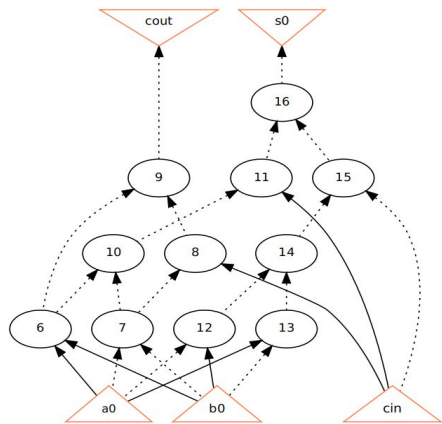
$$\alpha_\Delta^* = \frac{1}{|\mathbf{I}_\Delta|} \sum_{i \in \mathbf{I}_\Delta} |\mathbf{W}_i|.$$

Substituting

$$\Delta^* = \arg \max_{\Delta > 0} \frac{1}{|\mathbf{I}_\Delta|} \left( \sum_{i \in \mathbf{I}_\Delta} |\mathbf{W}_i| \right)^2$$

Now, this equation can be solved using discrete optimisation, but it time consuming, instead it is **approximated** considering the  $\mathbf{W}_i$  follow **Normal Distribution**.  $\Delta^* \approx 0.7 \cdot E(|\mathbf{W}|)$

# II. Experiment with 1-Bit Adder with TWNN



-Successful in **introducing zeros in the interconnects** suggests pruning of the connection i.e **reduces the fan-in** of the neurons, more closer to AIGs.

-Accuracy obtained=**93.75 % (Wrong Set, not unique)**

## Probable issues:

- The neuron could not realise the expected logic functionalities.
- The approximation of the ternary Function Threshold.

## TWNN Model

Number of hidden layers=4

Number of neurons in layer=[5,5,3]

Number of inputs=3

Number of outputs=2

**Accuracy=93.75%**

Considering dummy neurons to address jumps.

# L. Deng et.al GXNOR-Net(2018)

GXNOR-Net stands for Gated XNOR-Net. It is based on the I. Hubara et.al BinaryNet, however it is **generalised for any discrete space**. The model is trained in discrete space without **using the real-valued weights**.

-It is termed as Gated because both **weights and activation are ternarised**.

-Modified training algorithm than BNN.

-Mostly for classification, last layer is a L2-SVM (**Support Vector Machine**) to enhance accuracy of classification.

-The model proposed has a large number of **constants**, the **threshold**, the **derivative approximation** and the **nonlinear factor** for probabilistic projection used in the training algorithm.

-The author suggests that there exists special sets of constants for a **specified network and dataset**.

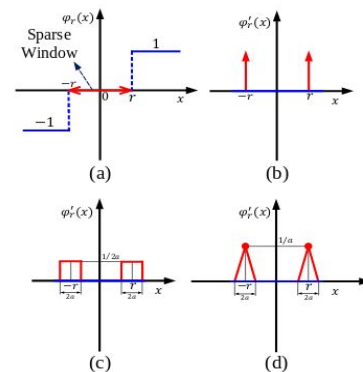
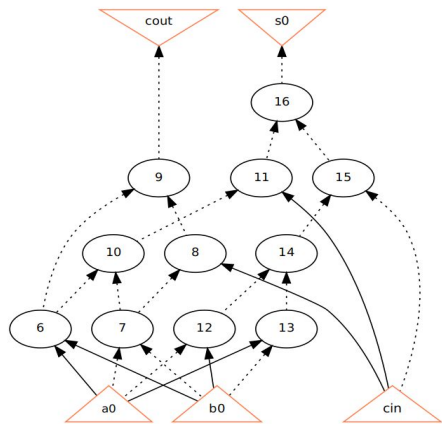


Fig:-The ternary activation Function and approximate derivatives.

# III. Experiment with 1-Bit Adder



Obtaining a suitable **threshold for each architecture** takes **a lot of time**, as for each threshold we need to train the network and then check the Accuracy. ( $O(n^2)$ ).

The optimal threshold cannot be found with just this heuristic as the accuracy plot has a lots of maximas.

## TWNN Model

Number of hidden layers=4

Number of neurons in  
layer=[5,5,3]

Number of inputs=3

Number of outputs=2

Considering dummy  
neurons to address  
jumps.

The 1 bit adder Ternary Neural Network Model Converges with 100% accuracy for a Threshold of **0.1256(not unique)**. However, this threshold results in lesser 0 weights.



# Conclusion and Future Direction

## Training Algorithm

- Need to look for some **training algorithms** using the AIGs.
- Back Propagation Based Algorithm to predict **no connection**, **buffer**, or **inverter** to each of the inter-connects of the AND nodes.
- Starting with the AIG interconnection** and then training the network for vacant portions.
- Discrete **Weight Update Optimiser** as discussed in L. Deng et.al GYNOR-Net(2018)



# Conclusion and Future Direction

## Network Architecture

- If the conventional DNN models are used to get to the AIG based structure, we need a **look for more sets of heuristic**.
- Heuristics are required for **Neural Structure Validation**.
- This can be done through **Miter construction** and then verification using SAT.
- However, we need to **limit the logic functionalities** of the neuron to opt for these type of validation.





Thank You for your attention!

# Partial Logic Synthesis using Deep Neural Networks.

~ Internship Summary Report

Sudarshan Sharma

Intern, Fujita Lab.

Senior Undergraduate, Dept. of Electronics  
and Electrical Communication Engineering  
IIT Kharagpur

## Introduction

The increasing enhancement in the field of Deep Learning extended its application in the varied domains apart from the conventional vision-based applications. The use of Deep Neural Networks(DNN) in the field of Electronic Design Automation(EDA) has been explored these days. In this work, the relationship between the discrete-valued deep neural networks and multi-level logic is studied in details. The motivation comes from the fact that the multi-level logic in the form of and-inverted graphs obtained from the ABC tool developed at UCB is one of the representations which can be used to start with the neural network architecture. This representation can be used to decide the layer and neurons respectively, once this is done one can use some iterative learning algorithms to train the deep neural network with outputs as +1 (High) and -1 (Low), and weights as (+1) buffer connections, (-1) inverted connections and 0 as no connection. In short, a trainable and-inverted graph for any logic function is desired using deep neural network based training algorithm in the discrete domain.

Partial Logic Synthesis problem deals with filling vacant part of a complete circuit to meet the overall specification given separately. Once the trainable and-inverted graph is obtained one can freeze the rest of the neurons nodes and train only for the neurons that resemble the vacant portion of the circuit.

The report covers the training algorithms used for the construction of the discrete deep neural network. The first sections consists of the previous work in the domain of binarized deep neural network in which both the weights and activation are binary (high or low). Since, we desire for a trainable AIG structure, the first two sections present a detailed study of the existing algorithms for the construction of the deep neural network model which can resembles an AIG. Three set of experiments Experiment I, Experiment II and Experiment III are discussed in great details including the code links to produce similar results as proposed. The report concludes with concluding remarks future work and direction.

## Partial Logic Synthesis

The partial logic synthesis problem as discussed in [1] first transform the vacant subcircuit using a Look up Table (LUT). Then the truth table values of the LUT along with the possible set of inputs to the complete circuit forms a two level Quantified Boolean Formula (QBF) problem as shown below.

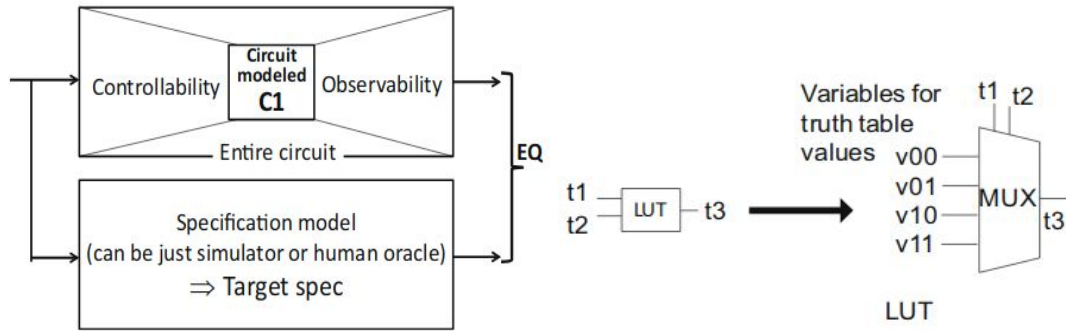


Fig1:- Partial Logic Synthesis Problem

QBF Equation

$$\exists v, \forall x, f(v, x) = SPEC(x)$$

This two level QBF equation can be converted to a satisfiability (SAT) problem by selecting particular set of Input values from the possible ones. However, this is a necessary condition, the solution is later verified using the equivalence checker, if equivalent the obtained truth table values of the LUT is the solution, if not it generates counter- examples which are later used instead of the input values in the later iterations.

## Present Approach

The main question arises how one can use the DNN in the partial logic synthesis? To answer this let's understand the Discrete valued Deep Neural Networks. Binarised/Ternarised DNN are the class of NN with binary/ternary weights, activation and biases (if used). The main aim of development of these models are as follows:- 1. Improvement in Power Efficiency, 2. Reduction in Memory Size (Storage), 3. Faster Access.

However, we wish to exploit the training strategy of these kinds of network models to come up with multilevel logic circuits similar to the class of And-Inverted Graphs (AIGs) where a neuron implements some logic function (AND in the case of AIG). Now, once we have the training strategy to construct such models similar to AIGs we can train a specific region in

the DNN, freezing the remaining interconnects. Thus, the vacant portion in the logic circuit can be obtained. The flow chart below shows the proposed scheme for partial logic synthesis using discrete DNN.

#### Flow Chart

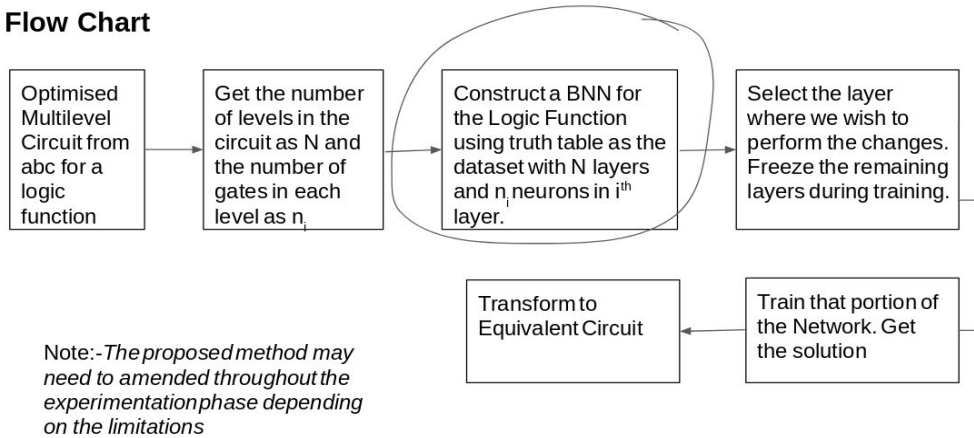


Fig2:- Flow chart of the proposed scheme.

## Previous Work

This section deals with the previous work in this domain. However, few terms need some simpler definition to get a good insight of the proposed methods discussed earlier. The detailed review of the BNN papers with algorithms are [here](#).

The list of papers reviewed are as follows:

1. R Kohut Boolean Neural Network (2004)
2. Nakayama Y A Simple Class of Binary Neural Networks and Logic Synthesis (2011)
3. Minje Kim et.al Bitwise Neural Network (2016)
4. Itay Hubara et.al Binarized Neural Network (2016)

### 1.R. Kohut et.al Boolean Neural Network.(2004)

The discussion is centered around a new neuron (Boolean Neuron) that in the neural network architecture can realise various set of Boolean Functions. The learning algorithm is based on sequential learning.

#### What is Sequential Learning?

The neural networks learning algorithms are broadly divided in two categories, the sequential learning algorithms and the iterative learning algorithms. In the former, we have the training dataset beforehand and one constructs the neural network by adding neurons, layers depending on the dataset, the training algorithm takes a lesser training time and assures fast

convergence. In the latter, we first fix the network architecture and then using iteration change the trainable parameters i.e the weights, bias, etc using certain optimisers and error gradients. The examples Functional on Tabular Function Set (FTFS) and Expand and Truncate Learning (ETL) falls under Sequential Learning, whereas the Back Propagation Algorithms and Radical Basis Function (RBF) falls under the Iterative Learning category.

The neural network model supports only a single hidden layer, requires entire truth table for training, and works fine with multi-output functions. The model fails to utilise the integral functionality of the neural network and ends up in more of an representation model similar to BDDs, AIGs etc. However, it helps to realise the important intermediate logic function in the representation.

## **2. Y. Nakayama et.al A Simple Class of Binary Neural Network and Logic Synthesis (2011)**

This paper describes a binary neural network model obtained through Genetic Algorithm based Learning Algorithm (GALA). Let the number of terms in the DCF say  $N_d$  and number of hidden layers in the Network is  $N_h$ . Moreover, they consider an ETR (Error Tolerance Rate) for the network which decides the  $N_h$ . As the dimension of the Input increases the number of input patterns increases exponentially and then it becomes difficult for Quine-McCluskey to optimise the logic. So, the authors proposes BNN for logic optimisation and debates that if the  $N_h$  is lesser than  $N_d$ , the ETR increases.

### **Why Genetic Algorithm?**

The genetic algorithm is used because such algorithms always look for global optimum, and can operate speedily as it does not require differentiability of the objects. The GALA has chromosomes corresponding to the parameters and has two features: the teacher signals are used as the initial chromosomes(Kernels) and the fitness is evaluated with the Error Tolerance Rate(ETR). Two main aim:

1. They give a parameter subspace where the BNN is equivalent to the DCF: the GALA is applicable to synthesise a DCF. In outside the subspace, the GALA can realise BNN with simpler structure than that of the DCF( $N_h \leq N_d$ ).
2. In numerical experiments for typical examples, it shows that the GALA with zero ETR can realise relatively large scale Boolean Functions whose logical synthesis is hard by the QMC. As the tolerance to error increases the number of neurons in the hidden layer decreases.



### **3. M.Kim et.al Bitwise Neural Network (2016)**

The author proposes a Multi-layered Bitwise Neural Network, on the assumption that there exists a neural network that efficiently represents a set of Boolean functions. The main aim of the model is lesser spatial complexity, memory bandwidth and power consumption in the hardware. The main advantage of this Bitwise NN is that all multiplication is converted to less processor heavy XNOR and bit counting operation. The training algorithm proposed wraps the real valued variables around the tanh function and then binarizes them after the real valued variables are updated using the gradients calculated while training. The method proposed introduces some of the constant like the sparsity parameter ( $\lambda$ ) and the boundary parameter ( $\beta$ ), however it lacks literature to get the exact intuition how these constants are implemented.

The exact training algorithm and the results section are described in great details [here](#). Let's discuss the model in brief. It uses a two way training approach, first one is Weight Compression followed by Noisy Backpropagation. In the weight compression step, the real valued weights are converted to the BNN and in the noisy backpropagation step the compressed weights are used to calculate the gradients which are used to update valued weights which are then converted to the binary once.

The results of this multilayered deep neural network model is shown for the Images and Audio datasets. Check this [blog](#) by the author on this, moreover, it has gathered the attention of few researchers and the FPGA implementation can be found [here](#). The algorithm fits in with the requirement but lacks any open-source implementation.

### **4. I. Hubara et.al Binarized Neural Network: Training Deep Neural Networks with weights and activations constrained to +1/-1 (2016)**

The paper presents a training algorithm for deep neural network with binary weights and activations at runtime. Again, the main aim behind these neural network algorithms being lower memory bandwidth and substantial increase in the power efficiency as many operation are replaced by bitwise parameters. This algorithm too uses binary weights to calculate the gradients and then update the real valued variables which are later compressed again into binary values using some binarization scheme the two proposed in the paper are Deterministic Binarization and Stochastic Binarization. However, the method use Deterministic Binarization as the Stochastic ones are tougher to implement on hardware. This algorithms stands by far the best alternative according to the requirements. Let's discuss in details the algorithm to get a better insight of it.

The Deterministic Binarization as discussed earlier is used as the compressing function, the training method uses a modified Back Propagation Algorithm followed by Stochastic Gradient Descent optimiser to update the weights. The important thing to understand is that in every iteration the real valued weights are updated through the gradients calculated using the binary weights. These modified real valued weights after getting the updated through the optimiser are again converted to binary values.

## Why do we need Modified BackPropagation?

To get familiar with the algorithm used one should be familiar with the backpropagation algorithm. A good explanation could be found at Hinton [Lectures](#) on Coursera on Machine Learning.

### Understanding the Backpropagation Algorithm

Assume mean square loss function.

$E = \sum (t_j - y_j)^2$ , where  $t_j$  is the target and  $y_j$  is the output of the last layer.

Using basic definition:

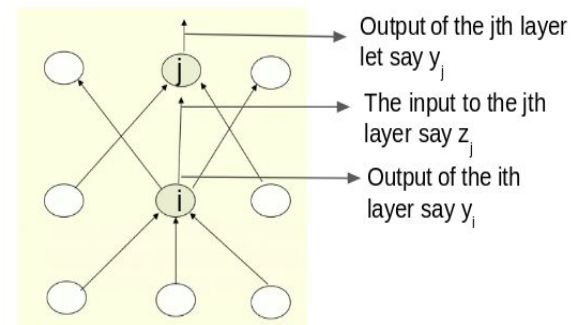
1.  $y_j = f(z_j)$ , where  $f(\cdot)$  is the activation function.
2.  $z_j = \sum (y_i * w_{ij})$ , where  $y_i$  is the out of the neuron in the second last layer, and  $w_{ij}$  is the weights from ith to jth layer.

Now, using chain rule we can write.

$$\frac{\partial E}{\partial z_j} = \frac{dy_j}{dz_j} \frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial E}{\partial z_j} = y_i \frac{\partial E}{\partial z_j}$$

The above derivative  $\frac{\partial E}{\partial w_{ij}}$  is the weight gradient used to update the weights for the next iteration, the equation above can be derived using the definition stated on top.



$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i} \frac{\partial E}{\partial z_j} = \sum_j w_{ij} \frac{\partial E}{\partial z_j}$$

This derivative is equally important and can be obtained using the definition stated above. It shows the essence of the backpropagation where the error from the above layers are transferred to the lower layers based on the contribution of the weights. It is used to calculate the gradients of the weights in lower layers.

The gradient  $\frac{\partial E}{\partial z_i}$  seems to be the most important gradient which can be used to calculate the weight gradients and the gradients for the subsequent layer. Looking at the gradient again after chain rule.

$$\frac{\partial E}{\partial z_j} = \frac{\partial f(z_j)}{\partial z_j} * \frac{\partial E}{\partial y_j}$$

The derivative of the **activation function turns out to be zero** if the deterministic binarization method is used to binarize the real valued weights. Thus, the training algorithm cannot pass gradients and the network remains as it is.

To mitigate the issue, Hinton in his coursera lecture proposed the concepts of the Straight Through Estimators (let EST) to pass gradients to the if the discrete signum activation function is used which was later studied by Bengion (2013) . He says that

$$est\left(\frac{\partial E}{\partial z_j}\right) = \frac{\partial \text{Sign}(z_j)}{\partial z_j} * \frac{\partial E}{\partial y_j} = \begin{cases} \frac{\partial E}{\partial y_j}, & -1 < z_j < 1 \\ 0, & \text{otherwise} \end{cases}$$

**This can be visualised as passing gradients using the tanh function while backpropagation, i.e. in simpler terms calculate the gradients using the activation function as tanh.**

Since, this neural network model has a lots of implementation on the internet however on images. But, one implementation on tensorflow [here](#) can be modified to make it compatible according to the requirement. The actual implementation by the author on tensorflow is [here](#).

## Experimental Setup.

As shown in the flow chart proposed above in Fig2, Let's get into the experimental setup of the construction of the Neural Network.

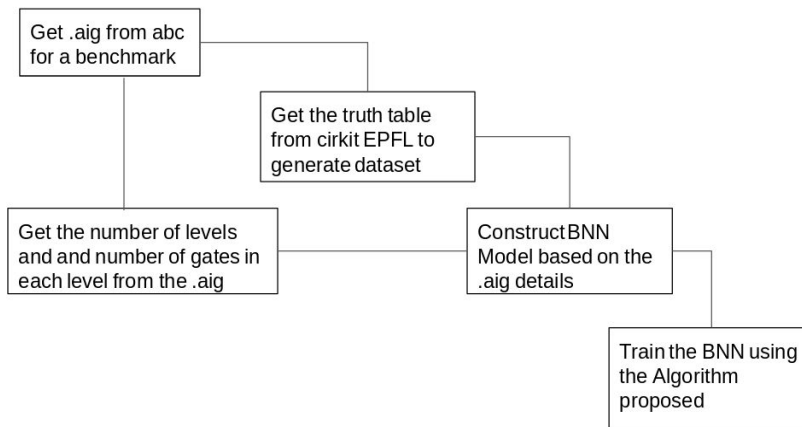


Fig3:-Experimental Setup for construction of Neural Networks

The Neural Network models uses the truth tables as the datasets and creating such datasets using the benchmarks becomes tough. [Circuit](#) a tool from EPFL can be used to generate truth tables from the benchmarks. The special feature of this tool is that it comes integrated with the abc tool. So, both the tools can be used easily.

Steps for the construction of the truth table based dataset.

1. Convert the benchmark (.bench, .dot, .aig etc) to .aiger using abc/cirkit.
2. Create a session log file in cirkit  
-./cirkit -l sess\_file.log
3. Load the (.).aiger on cirkit.  
-read\_aiger benchmark.aiger
4. Simulate for the Truth Table.  
-simulate -atn
5. Store the simulate truth table in the log file.  
-store -t
6. Quit and open the sess\_file. It should look like this.  
-quit

```

File Edit Selection Find View Goto Tools Project Preferences Help
sess_file.log x
1 [{
2   "command": "read_aiger 01-adder.aig",
3   "time": "2018-07-09 16:55:16"
4 },
5 {
6   "command": "simulate -atn",
7   "time": "2018-07-09 16:55:24",
8   "runtime": 0.000288741,
9   "tts": ["111010001110100011101000111010001110100011101000",
10  "10010110100101101001011010010110100101101001011010010110"]
11 },
12 {
13   "command": "store -t",
14   "time": "2018-07-09 16:55:27",
15   "tt": 1
16 },
17 {
18   "command": "quit",
19   "time": "2018-07-09 16:55:29",
20   "version": "#25-Ubuntu SMP Wed May 23 18:02:16 UTC 2018",
21   "supported threads": 8,
22   "nodename": "fujitalab-Z68MA-D2H-B3",
23   "machine": "x86_64",
24   "release": "4.15.0-23-generic",
25   "sysname": "Linux"
26 }]

```

Fig4:- The snippet of the log file from cirkit

The “tts” shows the truth table values of the output. However, there is the repetition sequence. For eg. for 1-bit adder the truth tables repeats after 8 values. The two outputs are the **cout** and **sum**. Now, How to interpret this sequence?. The description can be read from [here](#), In short the first element represents when all the variables attends a value of 1 (7 in decimal) i.e 111 with the order (**a,b,cin**), then 110 (6 in decimal) followed by 101(5 in decimal) etc.

Once the log file is obtained, we can get the truth table dataset in suitable format by scrapping this .log file, presently for the purpose of test and experimentation the “tts” copied from here was used to create custom dataset of the format as below.

```

01adder_TT.txt x 02adder_TT.txt x
1 5,3
2 00000,000
3 00001,010
4 00010,010
5 00011,100
6 00100,001
7 00101,011
8 00110,011
9 00111,101
10 01000,001
11 01001,011
12 01010,011
13 01011,101
14 01100,010
15 01101,100
16 01110,100
17 01111,110
18 10000,001
19 10001,011
20 10010,011
21 10011,101
22 10100,010
23 10101,100
24 10110,100
25 10111,110
26 11000,010
27 11001,100
28 11010,100
29 11011,110
30 11100,011
31 11101,101
32 11110,101
33 11111,111

```

Fig5:- The snippet of the created dataset at the top input, output followed by truth tables.

The Inference that which column represent which variable can be extracted by observing the order in the benchmark.

## Experiment I

Benchmark used One-bit-Adder.

### And Inverted Graphs (AIGs)

Number of levels=4

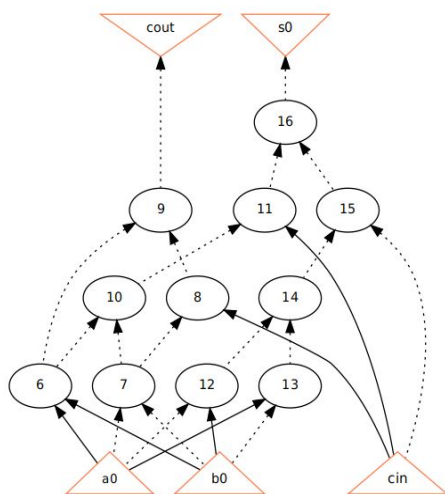
Number of gates in level=[4,3,3]

Number of input=3

Number of outputs=2

Network structure visualized by ABC  
Benchmark "01-adder". Time was Wed Jun 13 18:14:24 2018.

The network contains 11 logic nodes and 0 latches.



### BNN Model

Number of hidden layers=4

Number of neurons in layer=[4,3,3]

Number of inputs=3

Number of outputs=2

**Accuracy=87.5% (Wrong Set, Not Unique)**

To produce this result use the codes [here](#). This repository can be accessed after access, please

let me know for the access.

However if we increase one layer and consider [4,3,3,3] 100 % accuracy is obtained.(Random)

This heuristic fails in the case of 2-bit adders.

## Problems and Possible Solutions

At, this point, let's discuss the problems and their probable solutions.

1. Jumps from any lower layers (1,2,.....,n-1) to the nth levels are predominant in the AIGs, but in conventional neural network topology each neuron in the (n)th layer is connected to every neurons in the (n-1)th layer.

Possible Solution- Addition of dummy neurons which acts as a buffer.

2. Fan-in of each gate in the AIGs is restricted to 2. However, fan-in of the neuron depends on the number of neurons in the previous layer.

Possible Solution- Using Ternary Weighted Neural Network, where 0 means no connection.

3. The outputs in the AIGs sometimes are inverted too.

Possible Solution- Using an additional layer to incorporate the inversion.

4. One neuron could not implement the logic functionality required to represent a function.

Possible Solution- Observe the neuron functionality and amend the activation function.

Keeping in mind that the aim is to make a DNN resemble an AIG, the criteria of no-connections needs to be incorporated in the neural network training model as discussed in problem 2.

## Ternary Weighted Deep Neural Networks.

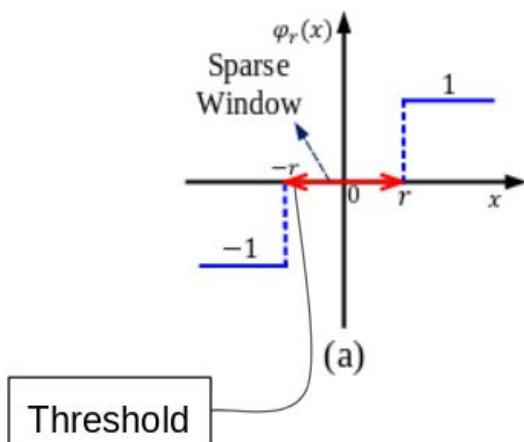
The discrete valued DNN are introduced these days with an implication of lower power efficiency and lower memory bandwidth. The weights are  $\{-1, 0, +1\}$ , where 0 means no connections. However, these training algorithms are tested on Images and used more the on Classification Problems.

### Previous Works.

1. F.Li, B. Zhang, B Liu, Ternary weight networks (2016)
2. Lei Deng et.al GXNOR-Net: Training deep neural networks ternary weights and activations without full-precision memory under a unified discretization framework. (2018)
3. H Alemdar et.al Ternary Neural Network for Resource-Efficient AI Applications (2017)

### 1.F.Li et.al Ternary Weight Networks(2016)

The author modifies the work presented in Hubara et.al Binarized Neural Network (2016) and adds the support of the ternary weights  $\{-1, 0, +1\}$  (where 0 means pruning the interconnection), and activation output as binary  $\{+1, -1\}$ . The main aim of the training algorithm is minimising the Euclidean distance between the real valued weights and the ternary weights using a positive scaling factor.



The threshold for the ternarization is obtained through the discrete optimisation using approximation.

The algorithm is as follows:-

Since, the aim is to minimise the euclidean distance between the full precision weights and



the ternary-valued weights.

$$\begin{cases} \alpha^*, \mathbf{W}^{t*} = \arg \min_{\alpha, \mathbf{W}^t} J(\alpha, \mathbf{W}^t) = \|\mathbf{W} - \alpha \mathbf{W}^t\|_2^2 \\ \text{s.t.} \quad \alpha \geq 0, \mathbf{W}_i^t \in \{-1, 0, 1\}, i = 1, 2, \dots, n. \end{cases}$$

This gives the optimisation function where  $J(\cdot)$  is the optimisation Function.

$$\alpha^*, \Delta^* = \arg \min_{\alpha \geq 0, \Delta > 0} (|\mathbf{I}_\Delta| \alpha^2 - 2 \left( \sum_{i \in \mathbf{I}_\Delta} |\mathbf{W}_i| \right) \alpha + c_\Delta)$$

Where the threshold function is

$$\mathbf{W}_i^t = f_t(\mathbf{W}_i | \Delta) = \begin{cases} +1, & \text{if } \mathbf{W}_i > \Delta \\ 0, & \text{if } |\mathbf{W}_i| \leq \Delta \\ -1, & \text{if } \mathbf{W}_i < -\Delta \end{cases}$$

$$c_\Delta = \sum_{i \in \mathbf{I}_\Delta^c} \mathbf{W}_i^2, \alpha \text{ dependent constant}$$

$$\mathbf{I}_\Delta = \{i | |\mathbf{W}_i| > \Delta\}, |\mathbf{I}_\Delta| \text{ denotes the number of elements in } \mathbf{I}_\Delta$$

By differentiating we get.

$$\alpha_\Delta^* = \frac{1}{|\mathbf{I}_\Delta|} \sum_{i \in \mathbf{I}_\Delta} |\mathbf{W}_i|.$$

And substituting it in the above equation we get  $\alpha$  as

$$\Delta^* = \arg \max_{\Delta > 0} \frac{1}{|\mathbf{I}_\Delta|} \left( \sum_{i \in \mathbf{I}_\Delta} |\mathbf{W}_i| \right)^2$$

Now, this equation can be solved using discrete optimisation but the author instead uses an approximation stating that the the usual methods are very time consuming.

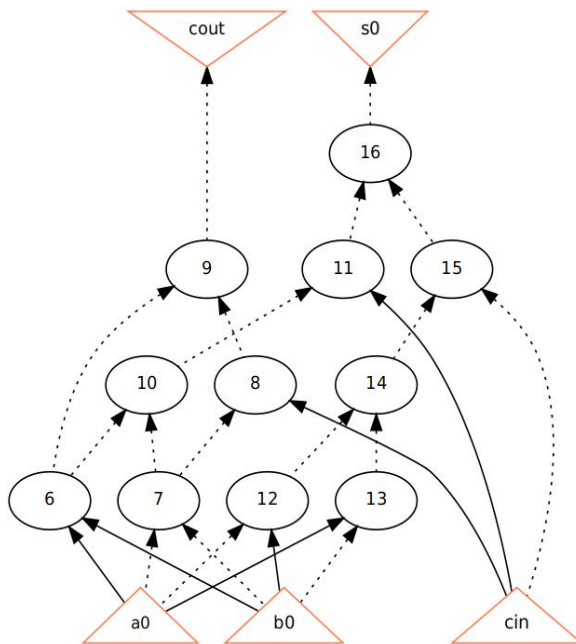
Approximation:- They say that the  $\mathbf{W}_i$  are from the normal distribution

$$\Delta^* \approx 0.7 \cdot E(|\mathbf{W}|)$$

The author uses an approximation at the end to solve the two set of equations, The approximation changes the ternarization threshold. Thus, the  $\Delta$  approximated is used as the ternarising threshold which decide how sparse the network be.

## Experiment II

Benchmark used- One Bit Adder



Ternary Weighted Neural Network Model (TWNN) Model

Number of hidden layers=4

Number of neurons in layer=[5,5,3]

Number of inputs=3

Number of outputs=2

**Accuracy=93.75% (Wrong set, not Unique)**

To produce the results use the code [here](#). I have used binary weights for the last layer interconnection and remaining layers have ternary weights. Can, modified easily.

In this set of experimentation, the dummy neurons were added incorporating jumps, and ternary weights for the interconnections.

This model didn't converge accordingly for the 2-bit adder case.

### Problems

- 1.The neuron could not realise the logic functionality.

2. The approximation of the neural threshold could not convert the weights to 0 appropriately.

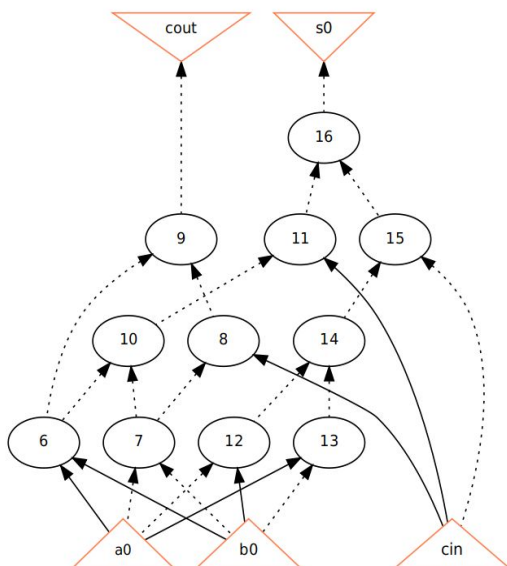
## 2. Lei Deng et.al GXNOR-Net: Training deep neural networks ternary weights and activations without full-precision memory under a unified discretization framework. (2018)

GXNOR-Net stands for Gated XNOR-Net. It is based on the I. Hubara et.al Binarized Neural Network, however it is generalised for any discrete space. The model is trained in discrete space without using the real-valued weights. It is termed as Gated because both weights and activation are ternarised, the last layer is followed by an L2-SVM layer to enhance the accuracy of the classification. The model proposed has a large number of constants, the threshold, the derivative approximation and the nonlinear factor for probabilistic projection used in the training algorithm. The author suggests that there exists special sets of constants for a specified network and dataset.

Considering the algorithm proposed the method iterate over the possible set of thresholds and then predict a suitable threshold which present the best accuracy.

### Experiment III

The aim of this experiment was to obtain a threshold for the particular set of network architecture for a particular dataset.



TWNN Model

Number of hidden layers=4  
Number of neurons in layer=[5,5,3]  
Number of inputs=3  
Number of outputs=2

The One bit adder Ternary Neural Network Model Converges with **100%** accuracy for a Threshold of **0.1256(not unique)**.

However, this threshold results in lesser 0 weights.  
Obtaining a suitable threshold for each architecture takes a lot of time, as for each threshold we need to train the network and then check the Accuracy. ( $O(n^2)$ ).

The optimal threshold cannot be found with just this heuristic as the accuracy plot has a lot of maximas.

The results of this experiment can be produced using the code [here](#).

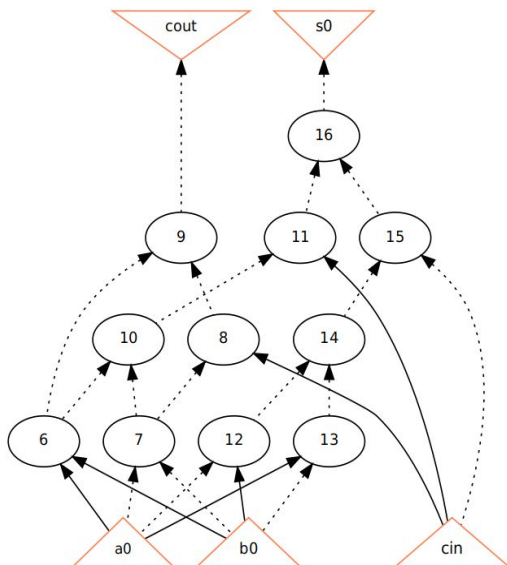
The entire .zip file of the set of experiments can be found [here](#).

## Future Work and Direction

The existing training approach for training deep neural network to resemble the AIG structure needs modification in the algorithm and the architecture construction heuristics, modifications includes improvement in the characterisation of the threshold for ternarisation, use of discrete training algorithm similar to the one proposed in Lei Deng et.al GYNOR-Net (2018).

The main aim of the research is to create a trainable AIG structure, presently in this report the existing training algorithms are used to get a DNN that represents a trainable AIG.

Considering the problem of partial logic synthesis, one other alternative could be using the given known interconnection from the AIG and then training to learn the vacant portion in it. Suppose, an AIG representation with one node missing, let's take the example of the one bit adder AIG representation.



Let's say node 8 is missing, the problem of partial logic synthesis now is to find the connection of the node 8 to the above level including node 9, 11 and 15 and know whether it is inverted or not. So, by training the network to learn whether there is a connection, no connection or an inverted connection, one can get the vacant portion of the circuit. Here, we assume that the neural network is created in such a manner that the jumps are incorporated using dummy neurons and the logic functionality of the dummy neuron would be like a buffer and for the main neurons like AND gates. Learning such weights can be attained through modification of the existing discrete DNN Training Algorithms.

The problem with iterative learning algorithms in which the network structure is set and then followed by gradient based optimiser to train the network is that the network architecture needs to be capable of converging for the given dataset. To check whether the network

architecture is correct, one can formulate QBF based problem using Multiplexers at each terminal of the LUT

representing the different logic functionality of the neuron. The satisfiability (SAT) or unsatisfiability (UNSAT) then states whether the network can converge for the dataset or not. Although, this is not required if the training algorithm starts with the AIG structure, this method of validating the network architecture seems more intuitive when conventional existing DNN training algorithms are used to resemble an AIG structure.