High Performance Graph Convolutional Networks with Applications in Testability Analysis

Yuzhe Ma
CUHK
yzma@cse.cuhk.edu.hk

Haoxing Ren
NVIDIA
haoxingr@nvidia.com

Brucek Khailany
NVIDIA
bkhailany@nvidia.com

Harbinder Sikka
NVIDIA
hsikka@nvidia.com

Lijuan Luo
NVIDIA
liluo@nvidia.com

Karthikeyan Natarajan
NVIDIA
knataraj@nvidia.com

Bei Yu
CUHK
byu@cse.cuhk.edu.hk

ABSTRACT

Applications of deep learning to electronic design automation (EDA) have recently begun to emerge, although they have mainly been limited to processing of regular structured data such as images. However, many EDA problems require processing irregular structures, and it can be non-trivial to manually extract important features in such cases. In this paper, a high performance graph convolutional network (GCN) model is proposed for the purpose of processing irregular graph representations of logic circuits. A GCN classifier is firstly trained to predict observation point candidates in a netlist. The GCN classifier is then used as part of an iterative process to propose observation point insertion based on the classification results. Experimental results show the proposed GCN model has superior accuracy to classical machine learning models on difficult-to-observation nodes prediction. Compared with commercial testability analysis tools, the proposed observation point insertion flow achieves similar fault coverage with an 11% reduction in observation points and a 6% reduction in test pattern count.

1 INTRODUCTION

Machine learning (ML) and deep learning (DL) has emerged as a powerful technique in many fields. It can derive knowledge from large amounts of data and provide predictions, estimations, or even contents generation. In the field of electronic design automation (EDA), learning approaches have mainly been used to accelerate the design process [1–5]. Among various learning models, convolutional neural network (CNN) approaches are widely used. For example, RouteNet [4] leveraged a CNN to predict the routability of a placed design efficiently. Whereas CNNs are discriminative models, generative adversarial networks (GANs) are generative models and have attracted significant attention in recent years. For example, in [5], a GAN model is developed to perform optical proximity correction (OPC) on lithography masks for better manufacturability. In both example applications, the inputs (layout and mask), can be treated as images in order to build upon prior work using DL for image processing. However, CNN-based approaches often include convolution and pooling layers and require data defined on regular grids. There are many EDA applications without grid-based data representations, where alternative approaches are needed.

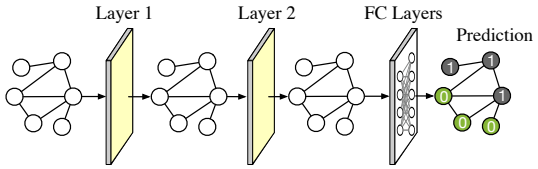


Figure 1: Network for node embedding and classification. Layer 1 and Layer 2 generate node embeddings. Nodes are classified through fully-connected (FC) layers.

Graphs play a vital role in EDA since they are natural representations for fundamental objects such as netlists and layout. Many problems have previously been solved using graph processing, such as partitioning [6], layer assignment [7], multiple patterning layout decomposition [8, 9] and testability analysis [10]. However, there are few DL-based solutions proposed for graph-based problems in EDA [3]. One reason is that it is not straightforward to generalize CNNs from processing regular grid-based data input to processing graphs. Recently, a number of studies from the DL community have shown how to adopt learning models on graphs, most notably the graph convolutional network (GCN) approach [11–13]. As shown in the example in Figure 1, the essential idea is to obtain an embedding for each node by aggregating information from a node's local neighborhood iteratively such that the node attributes and structural information are encoded. The embeddings can then be leveraged as features in the machine learning tasks.

One critical consideration of applying learning-based techniques to EDA is scalability. A typical modern design from a commercial SoC could contain millions of gates, so any model proposed for use in EDA must achieve high throughput on these large graphs. However, previous GCNs [11, 12] did not demonstrate strong scaling capability. In [13], a scalable GCN inference pipeline is proposed for recommendation system in web scale. It relies on a distributed computing framework and is computationally expensive, which makes it difficult to deploy in an EDA toolflow environment.

In this paper, a high-performance GCN model is proposed for tackling EDA problems with inputs structured as graphs. With a netlist represented as a graph, the GCN model can generate the embedding for each node automatically using the aggregators and encoders, considering both node attributes and graph structural information. By selecting the aggregators properly and leveraging efficient GPU computation, the GCN model is scalable to process a graph containing millions of nodes and edges efficiently. As a

978-1-7281-2425-4/19/$31.00 ©2019 IEEE

demonstration, the proposed GCN model is applied to testability analysis for improving circuits testability, which is cast as an imbalanced classification problem and an observation point insertion problem. The proposed GCN model provides fast and accurate prediction, and iteratively inserts observation points to improve testability effectively. The main contributions are summarized as follows:

- A methodology using GCNs for netlist representation and modeling is proposed.
- A parallel training scheme with multiple GPUs and a fast inference scheme are presented for efficient GCN training and inference, which can scale to millions of standard cells.
- A GCN classifier is trained to predict observation point candidates in a netlist, and the proposed GCN classifier is integrated in an iterative observation point insertion process.
- Experimental results indicate the GCN outperforms conventional machine learning models in terms of classification accuracy, and the proposed flow can achieve superior testability results to conventional testability analysis tool on industrial designs.

The rest of this paper is organized as follows. In Section 2, preliminary material about GCN and testability analysis is introduced. The proposed GCN model is presented in Section 3. Section 4 describes how to integrate the GCN classifier to observation points insertion flow. The experimental results are reported in Section 5, and Section 6 concludes the paper.

2 PRELIMINARIES

2.1 Node Embedding and GCN

Graph-based learning is a new approach to machine learning with a wide range of applications [14]. One advantage is the graph structure itself can reveal relevant information. Node embedding and classification is one of the most important problems in graph-based learning. Before a machine learning model can be trained, a representation of the node must be obtained first, which is known as embedding. The embedding is a vector with a specific dimension, and can be fed to conventional machine learning models for prediction. Previous approaches exploring node embedding problems can be classified into two categories. The first class of approaches are based on heuristics to encode the structural information [15]. A more recent approach is data-driven, which learns node embeddings automatically [12, 13, 16].

Graph convolution networks are one of these data-driven approaches. Within these data-driven approaches, they can be further classified into transductive and inductive. Transductive approaches directly optimize the embedding for each node, thus they require all nodes to be present during training, and hence cannot generalize to unseen graphs [16]. Inductive approaches generate node embeddings through learning a set of functions to aggregate the structural information and node attributes, which make the learned model independent from training graphs. Therefore, inductive models can be applied to unseen data [12, 13].

2.2 Test Points Insertion

Test point insertion (TPI) is a broadly used approach in design-for-testability (DFT) to modify a circuit and improve its testability,

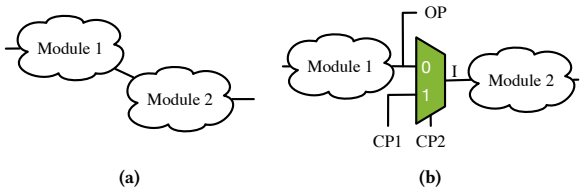


Figure 2: (a) Original circuit. Module 1 is unobservable. Module 2 is uncontrollable; (b) Insert test points to the circuit. Set (CP1, CP2) to (0, 1) and (1, 1) will set line I to 0 and 1, respectively; Set CP2 to 0 is the normal operation mode.

which involves adding extra control points (CPs) or observation points (OPs) to the circuit. An example of TPI is given in Figure 2. CPs can be used for setting signal lines to desired logic values, while OPs are added as scan cells to make a node observable. There are several issues that needed to be considered when performing TPI. On one hand, the optimal test point placement problem is NP-complete [17]. Numerous TPI methods have been proposed to investigate the efficiency and performance of TPI. Based on the runtime, these methods can be categorized into exact fault simulation [18], approximate measurements [19] and simulation-based methods [20]. On the other hand, inserting test points may degrade the performance of a design in terms of area, power and timing. The ultimate goal of TPI is to achieve high fault coverage with less performance degradation. Previous works explored beneficial trade-offs between testability improvement and performance degradation [21–25], among which CPs insertion is considered in [24] and OPs insertion is considered in [25]. Touba *et al.* [22] consider inserting both CPs and OPs. The approach investigated in this paper is generic and can be applied to both CPs insertion and OPs insertion.

2.3 Problem Formulation

In this work, we focus on applying observation points insertion (OPI) to improve the testability of a design. From the perspective of a machine learning model, finding the location where the observation points should be inserted in a circuit can be cast as a binary classification problem. For each gate in a design, the problem is whether to add an observation point on the output port or not. If historical data on a sufficient number of designs can be obtained, a classifier can be trained and applied to other designs. Considering that the netlist can be easily represented as graph, GCN is an appropriate approach for this application.

Problem 1 (Observation Points Insertion). Given a set of netlists with all the nodes labeled as either difficult-to-observe or easy-to-observe. The objective is to train a classifier and adopt it to find a set of locations where the observation points should be inserted, which can maximize fault coverage and minimize observation points number and test pattern number.

3 GCN FOR NODE CLASSIFICATION

3.1 Dataset Generation

A netlist is represented as a directed graph in which each node corresponds to a cell and each edge is a wire. The source nodes and sink nodes correspond to primary input and primary output, respectively. Each node has an attribute which is a four dimensional feature vector $[LL, C0, C1, O]$. LL denotes logic level of the corresponding gate. $[C0, C1, O]$ are SCOAP measurements [26], which

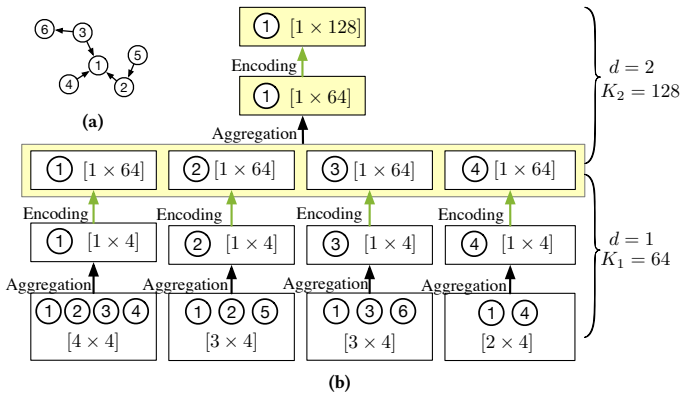


Figure 3: An illustration to compute the embedding for a node with $D = 2$. (a) Graph; (b) Procedure to compute the embedding for node 1.

correspond to controllability-0, controllability-1 and observability, respectively. Every node also has a binary label. '0' (negative) means easy-to-observe and '1' (positive) means difficult-to-observe. Labels can be obtained from commercial DFT tools. Given the graphs with node attributes and node labels, a GCN model can be trained, which will be introduced later.

3.2 Node Embedding Structures

To classify a node in the graph, the neural network first generates its node embedding which is not only based on its own attributes but also the structural information of the local neighborhood. Then, a classification model takes the node embedding as input and predicts a label. To achieve this, three kinds of modules are included in our GCN model, which are aggregator, encoder and classifier. Aggregators and encoders are used to generate the node embeddings by exploiting node attributes and the neighborhood information. The classifier predicts the label for each node in the graph based on its embedding.

Next, we introduce how the node embedding is generated using aggregators and encoders. Essentially, an aggregator or an encoder can be interpreted as a layer of the GCN. Each of them performs a specific operation on the node. An aggregator gathers the feature information from the node's neighbors using an aggregation function $Agg(\cdot)$. An encoder is applied to propagate information between different layers using a weight matrix. The embedding computation process, i.e., *aggregation* and *encoding*, is performed iteratively. Fully-connected layers are used as classifier which takes the node embedding as input, and predicts the label for the node.

Suppose that the network is trained and all the weights are obtained. Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and node attributes $\{\boldsymbol{x}^{(v)} : \forall v \in \mathcal{V}\}$, the node embeddings $\{\boldsymbol{e}^{(v)} : \forall v \in \mathcal{V}\}$ are generated as in Algorithm 1. Since the node embedding is expected to aggregate the information in local neighborhood, a depth D is specified to indicate the "radius" of the neighborhood region of a node. The initial representation $[LL, C0, C1, O]$ is set as the node attributes (line 1). There are two loops involved. In each step of the outer loop, the representation of each node is updated through aggregation and encoding. More specifically, in the d-th iteration of the outer loop, every node first aggregates information from its neighbors using aggregation function $Agg(\cdot)$ which takes the representations of node

Algorithm 1 Node Embedding Computation

Input: Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; node attributes $\{\boldsymbol{x}^{(v)} : \forall v \in \mathcal{V}\}$; Search depth D; non-linear activation function $\sigma(\cdot)$; Weight matrices $\boldsymbol{W}_d$ of encoders $\boldsymbol{E}_d, d = 1, ..., D$;

Output: Embedding of for each node $\boldsymbol{e}_D^{(v)}, \forall v \in \mathcal{V}$.

1: $\boldsymbol{e}_0^{(v)} \leftarrow \boldsymbol{x}^{(v)}, \forall v \in \mathcal{V}$;
2: **for** $d = 1, ..., D$ **do**
3: **for all** $v \in \mathcal{V}$ **do**
4: Compute $\boldsymbol{g}_d^{(v)}$ based on Equation (1); ▷ Aggregation
5: $\boldsymbol{e}_d^{(v)} \leftarrow \sigma(\boldsymbol{W}_d \cdot \boldsymbol{g}_d^{(v)})$; ▷ Encoding
6: **end for**
7: **end for**

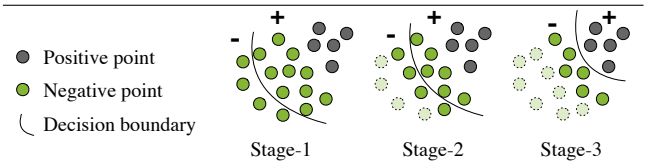


Figure 4: An example of three-stage GCN classification.

v and its neighbors generated at $(d-1)$-th iteration as input, and generates a new representation for node v, denoted by $\boldsymbol{g}_d^{(v)}$ (line 4). We use a weighted sum function as the aggregation function. Assume predecessors (PR) and successors (SU) have different weights. The aggregation $Agg(\cdot)$ can be formulated explicitly as:

$$\boldsymbol{g}_d^{(v)} = \boldsymbol{e}_{d-1}^{(v)} + w_{pr} \times \sum_{u \in \mathrm{PR}(v)} \boldsymbol{e}_{d-1}^{(u)} + w_{su} \times \sum_{u \in \mathrm{SU}(v)} \boldsymbol{e}_{d-1}^{(u)}, \quad (1)$$

where w_{pr} and w_{su} are weights for predecessors and successors, respectively, and they are the same in each step of outer loop. Next, a non-linear transformation is performed to encode the aggregated representation using a weight matrix $\boldsymbol{W}_d \in \mathbb{R}^{K_{d-1} \times K_d}$ and an activation function $\sigma(\cdot)$ (line 5). K_d is the dimension of the embedding after d-th step and K_0 is 4 which is the initial attribute dimension. A concrete example is shown in Figure 3 which illustrates the procedure of computing node embedding with $D = 2$. Essentially, after d iterations, the embedding of a node combines the information of its d-hop neighborhood.

When maximum depth D is reached, the final embeddings are obtained and fed to the fully-connected layers for classification. Parameters that need to be trained include w_{pr}, w_{su}, $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_D$ and parameters in FC layers. All the parameters in the network can be trained end-to-end.

Different from other transductive approaches which cannot generalize to unseen graphs [11, 16], the entire classification procedure for each node is only based on its neighborhood information and the learned parameters and can be shared across different graphs.

3.3 Multi-stage Classification

For a typical design, it is common to have many more negative nodes than positive nodes, which is not desireable for training machine learning models. Training a single classification model can lead to poor overall performance since significant bias would be introduced towards the majority class. To tackle this imbalance issue, we developed a multi-stage GCN for this problem. In each stage, a GCN is trained and only filters out negative cases with high confidence, and passes the remaining nodes to the next stage, which

is illustrated in Figure 4. This is achieved by imposing a large weight on the positive nodes such that the penalty of misclassifying them would be large. In this way, most positive points remain on the right side of the decision boundary until negative points are substantially reduced. After a few stages, the remaining nodes should become relatively balanced and a network can make the final predictions.

3.4 Efficient Training and Inference

3.4.1 Inference. Making the GCN model scalable to large graphs is a critical problem, especially because fast inference is desired. The computation shown in Algorithm 1 is an iterative process. However, it would be inefficient since the neighborhoods of different nodes may have overlap, thus there are many duplicated computations [12, 13]. Here we introduce another approach to inference computation that enables our GCN to process millions of nodes efficiently. The key idea is to leverage the adjacency matrix of the graph, denoted by $\boldsymbol{A} \in \mathbb{R}^{N \times N}$. N is the total number of nodes in the graph. A matrix $\boldsymbol{E}_d \in \mathbb{R}^{N \times K_d}$ can also be obtained, in which the v-th row represents the embedding of node v after the d-th iteration, i.e., $\boldsymbol{E}_d[v,:] = \boldsymbol{e}_d^{(v)}$. Take the graph in Figure 3(a) as an example. The weighted sum aggregation in iteration d is equivalent to Equation (2).

$$\boldsymbol{G}_d = \boldsymbol{A} \cdot \boldsymbol{E}_{d-1} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 1 & w_1 & w_1 & w_1 & 0 & 0 \\ w_2 & 1 & 0 & 0 & w_1 & 0 \\ w_2 & 0 & 1 & 0 & 0 & w_2 \\ w_2 & 0 & 0 & 1 & 0 & 0 \\ 0 & w_2 & 0 & 0 & 1 & 0 \\ 0 & 0 & w_1 & 0 & 0 & 1 \end{bmatrix} \end{array} \times \begin{bmatrix} \boldsymbol{e}_{d-1}^{(1)} \\ \boldsymbol{e}_{d-1}^{(2)} \\ \boldsymbol{e}_{d-1}^{(3)} \\ \boldsymbol{e}_{d-1}^{(4)} \\ \boldsymbol{e}_{d-1}^{(5)} \\ \boldsymbol{e}_{d-1}^{(6)} \end{bmatrix} \quad (2)$$

Here $\boldsymbol{A} \in \mathbb{R}^{6 \times 6}$, $\boldsymbol{G}_d \in \mathbb{R}^{6 \times K_{d-1}}$, and the v-th row is the representation for node v after aggregation in the d-th iteration. The inner loop in Algorithm 1 (line 3 – line 6) can be simply formulated as

$$\boldsymbol{E}_d = \sigma(\boldsymbol{G}_d \cdot \boldsymbol{W}_d) = \sigma((\boldsymbol{A} \cdot \boldsymbol{E}_{d-1}) \cdot \boldsymbol{W}_d). \quad (3)$$

Then, all the computation can be formulated as a series of matrix multiplications which can be efficiently computed, and duplicated computation can be avoided. One potential issue is the dimension of adjacency matrix $\boldsymbol{A}$ is $N \times N$, which is extremely large and cannot be stored in memory directly. However, we can exploit the fact that the $\boldsymbol{A}$ is a sparse matrix. For every design in our benchmarks, the sparsity is higher than 99.95%. Then A can be represented in a compressed sparse format, e.g., coordinate (COO) format which stores a list of (`value`, `row_index`, `column_index`) tuple. The matrix can be stored in the memory to enable the matrix multiplication. For instance, the COO representation of $\boldsymbol{A}$ in Equation (3) is represented as

$$\begin{aligned} \texttt{value:} \quad & [1, w_1, w_1, w_1, w_2, 1, w_1, w_2, 1, w_2, w_2, 1, w_2, 1, w_1, 1] \\ \texttt{r\_index:} \quad & [1, 2, 3, 4, 1, 2, 5, 1, 3, 6, 1, 4, 2, 5, 3, 6], \\ \texttt{c\_index:} \quad & [1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 5, 5, 6, 6] \end{aligned}$$

where each column is a tuple indicating the value and indices of a non-zero element in the matrix. Moreover, the COO format is very efficient for incremental matrix construction which facilitates graph modifications in our flow.

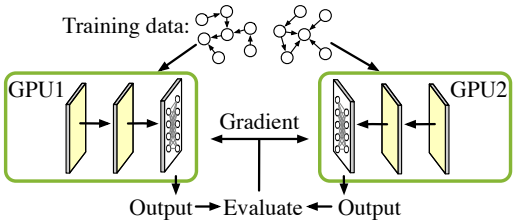


Figure 5: Parallel training with multiple GPUs.

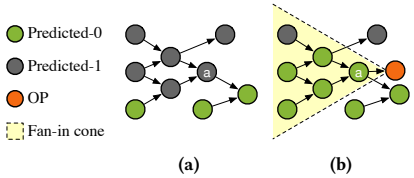


Figure 6: An illustration to compute the impact for an OP. (a) Prediction on original graph with 5 positives in fan-in cone of a; (b) Prediction on graph after inserting an observation point with 1 positive. The impact of node a is $5 - 1 = 4$.

3.4.2 Training. The strategy proposed for inference can also accelerate the GCN training process. In practice, the training set may contain many graphs. A straightforward method is to compose multiple graphs into a single graph, but the memory of a single GPU may not be sufficient to hold all intermediate results in this case. To overcome this bottleneck, we leverage a parallel training scheme with multiple GPUs shown in Figure 5. Our scheme can be seen as a variant of a conventional data-parallel scheme. Conventionally, a batch of input data is split into equal chunks and each GPU processes a chunk. With our GCN approach, the input of one graph includes an adjacency matrix and node representation matrix which cannot be split. Therefore, we separate multiple graphs into individual graph. Each GPU processes one graph, and all of the output is gathered to calculate the loss and then do back-propagation to update the model.

4 ITERATIVE OP INSERTION

After a multi-stage GCN model is trained, it can identify difficult-to-observe nodes in a netlist, which can be used as guidance for observation point insertion. However, not every difficult-to-observe node has the same *impact* for improving the observability since it is possible that adding an observe point may improve the observability of other nodes in its fan-in cone. In order to minimize the number of insertion points, we must select the observation point locations with largest *impact*. Next we develop a flow to identify which locations are more significant using the trained classifier. We define the *impact* of each location as the positive prediction reduction in a local neighborhood after inserting an observation point. Figure 6 gives an example of impact calculation. An iterative flow for points insertion is developed, which is shown in Figure 7. In each iteration, every positive prediction is evaluated to get its impact. Finally, a list containing observation points location is obtained. Then they are sorted based on their impact and select the top ranked locations. Next, we modify the graph and perform inference for prediction. The positive predictions will become the candidates in the next iteration. The exit condition is there are no positive predictions left.

Table 1: Statistics of benchmarks

Design	#Nodes	#Edges	#POS	#NEG
B1	1384264	2102622	8894	1375370
B2	1456453	2182639	9755	1446698
B3	1416382	2137364	9043	1407338
B4	1397586	2124516	8978	1388608

Inserting observation points modifies the netlist, thus the graph should be modified, including the graph structure and node attributes. Inserting one observation point to a target node v corresponds to adding a new node p to the graph and adding an edge from the target node v to new node p. Essentially, the adjacency matrix $\boldsymbol{A}$ and initial embedding matrix $\boldsymbol{E}_0$ should be updated. A critical problem in this iterative flow is how to update the graph efficiently. In our flow Figure 7, the graph can be updated incrementally. $\boldsymbol{A}$ can be incrementally updated by adding a column and a row, and setting corresponding entries as w_{pr} or w_{su}, which can be done efficiently under COO format by appending 3 tuples (w_{pr}, p, v), (w_{su}, v, p) and $(1, p, p)$. $\boldsymbol{E}_0$ is updated by appending attribute of new node p which is set to [0,1,1,0]. Then only the attributes of the nodes in the fan-in cone of the new node should be updated based on SCOAP method [26]. Since this GCN model is inductive, the updated $\boldsymbol{A}$ and $\boldsymbol{E}_0$ are directly fed to the model for prediction.

5 EXPERIMENTAL RESULTS

The experiments are performed on 4 industrial designs implemented in 12*nm* technology. Statistics of designs are summarized in Table 1. #POS and #NEG indicate the number of difficult-to-observe nodes and easy-to-observe nodes, respectively. The labels are obtained from commercial DFT tools. The GCN is implemented with PyTorch and trained on a Linux machine with 32 cores and 4 NVIDIA Tesla V100 GPUs. The total memory used in the training is 64GB.

Search depth is an important hyper-parameter that may affect the performance of a GCN. On one hand, increasing search depth can cover a larger neighborhood such that more information can be involved. On the other hand, too large region may lead to overfitting. We select the search depth by monitoring and comparing the training and testing accuracy among different settings on the search depth. K_1, K_2 and K_3 are set as 32, 64 and 128, respectively. The Rectified linear unit (ReLU) function $\text{ReLU}(x) = \max(x, 0)$ [27] is used as the activation function. Four FC layers are consistent, whose dimensions are 64, 64, 128 and 2. Figure 8 shows the record of training accuracy and testing accuracy during learning for 300 epochs with search depth 1, 2 and 3. It can be seen that the performance of GCN improves as the search depth increases. Search depth $D = 3$ is used for all the remaining experiments. The entire network consists of 3 aggregators, 3 encoders (with ReLU layer) and 4 FC layers. We use cross-entropy function as loss function and stochastic gradient descent for optimization.

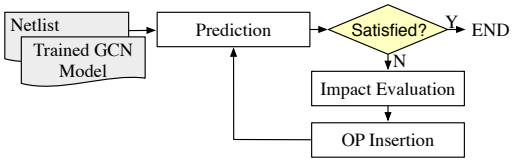
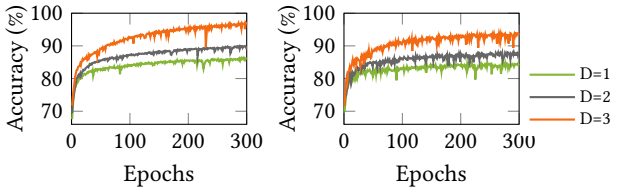


Figure 7: Iterative flow for observation points insertion.



(a) Training accurary **(b) Testing accuracy**

Figure 8: Performance with different search depth D.

Table 2: Accuracy Comparison on Balanced Dataset

Design	LR	RF	SVM	MLP	GCN
B1	0.778	0.790	0.813	0.860	**0.928**
B2	0.767	0.785	0.809	0.845	**0.929**
B3	0.779	0.793	0.814	0.856	**0.930**
B4	0.782	0.801	0.821	0.862	**0.935**
Average	0.777	0.792	0.814	0.856	**0.931**

We compare the classification performance between our proposed GCN and various classical machine learning models, including logistic regression (LR), random forest (RF), support vector machine (SVM) and Multi-layer perceptron (MLP). Considering a single classifier may not have good performance on the benchmark that is highly imbalanced, we compare the performance on balanced datasets by sampling a subset within the entire dataset. Since other classical machine learning models require handcrafted features with a fixed dimension, we integrate neighborhood features by collecting the features of the nodes in the fan-in cone and fan-out cone. 500 nodes in fan-in cone and 500 nodes in fan-out cone are collected. Starting from the target node, breadth-first-search is performed to collect the nodes in each cone. Every time a node is visited, the feature of this node is concatenated to the current feature vector. Therefore, the dimension of the feature vector for traditional learning models is $(500 + 500 + 1) \times 4 = 4004$.

For each design, we generate a balanced dataset by using all the positive nodes and sampling the same number of negative nodes randomly. Each time we use three designs for training and the remaining one for testing. For the MLP approach, the configuration of the network is the same as the classifier module in GCN. The accuracy comparison is presented in Table 2. GCN achieves 93.1% accuracy on average, outperforming other models on all designs.

We have shown that a GCN can obtain significantly better performance in distinguishing positive nodes and negative nodes than classical learning models. However, the performance a single GCN is not satisfying if it is directly trained and tested on original imbalanced dataset. To validate the advantage of the multi-stage GCN, we compare the performance between the multi-stage GCN and single GCN. In highly imbalanced classification, *F1-score* is commonly used since accuracy would be misleading. The training and testing schemes are the same as before. Each time we use three designs for training and the remaining design for testing. 3 stages are applied and the prediction results of each stage are combined to obtain the F1-score on the entire dataset. The results comparison is shown in Figure 9. The multi-stage GCN achieves much higher F1-scores than single GCN on all these imbalanced datasets.

By taking the advantage of the sparse representation of the adjacency matrix and matrix multiplication-based computation,

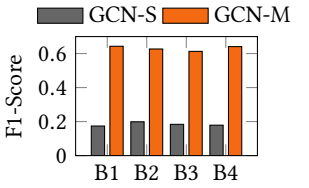


Figure 9: F1-Score comparison.

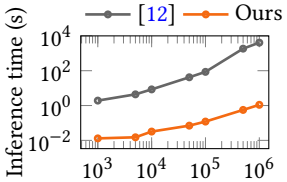


Figure 10: Scalability vs. [12].

Table 3: Testability results comparison

Design	Industrial Tool			GCN-Flow		
	#OPs	#PAs	Coverage	#OPs	#PAs	Coverage
B1	6063	1991	99.31%	5801	1687	99.31%
B2	6513	2009	99.39%	5736	2215	99.38%
B3	6063	2026	99.29%	4585	1845	99.29%
B4	6063	2083	99.30%	5896	1854	99.31%
Average	6176	2027	**99.32%**	**5505**	**1900**	**99.32%**
Ratio	1.00	1.00	**1.00**	**0.89**	**0.94**	**1.00**

the proposed GCN can scale to process designs with millions of cells. To demonstrate the scalability of our acceleration strategy, we compare the inference runtime between two approaches on graphs with different sizes, ranging from 10^3 nodes to 10^6 nodes. We leverage the released implementation of [12] for comparison, which uses recursion-based computation. The inference runtime comparison between two approaches is shown in Figure 10. For a design with 1 million cells, the recursive computation takes more than one hour to complete the inference. With our acceleration strategy, the inference runtime is only 1.5 seconds, which is three orders of magnitude speedup compared to [12].

Finally we show the testability results of applying the multi-stage GCN and the iterative test points insertion flow. We use the testability analysis results from a commercial testability analysis tool as a baseline. 3 metrics are used for evaluation, including the total number of OPs inserted, the fault coverage and the number of test patterns required. The output list of our proposed iterative GCN-based flow is fed to the same commercial tool to get a test report from which we can get the pattern number and fault coverage for a fair comparison. The results are shown in Table 3. Column '#OPs' represents the total number of OPs inserted. '#PAs' represents the number of test patterns. 'Coverage' represents fault coverage. It can be seen that the OPs recommended by the proposed GCN model and iterative flow outperform the industrial tool, achieving 11% reduction on the number of inserted OPs and 6% reduction on test patterns without any degradation on fault coverage.

6 CONCLUSION

In this paper, a GCN-based methodology is proposed for netlist representation in EDA field. The methodology is applied to identification of difficult-to-observe points in a netlist, in which GCN shows superior performance to classical learning models. A multi-stage GCN is developed to handle the imbalanced classification problem, which achieves significantly higher F1-score than single GCN model. A parallel training scheme is developed to enable large scale training and a fast inference scheme is proposed to enhance the scalability of the GCN model. Based on the GCN model and an iterative observation points insertion flow, we have achieved better testability on industrial designs compared to a state-of-the-art commercial tool. We believe the GCN-based methodology is generic and can be applied to other EDA problems, and hope this paper will inspire more future research in this area.

7 ACKNOWLEDGMENTS

This work is supported in part by The Research Grants Council of Hong Kong SAR (Project No. CUHK24209017).

REFERENCES

[1] E. Cai, D.-C. Juan, S. Garg, J. Park, and D. Marculescu, "Learning-based power/performance optimization for many-core systems with extended-range voltage/frequency scaling," *IEEE TCAD*, vol. 35, no. 8, pp. 1318–1331, 2016.
[2] Y. Ma, S. Roy, J. Miao, J. Chen, and B. Yu, "Cross-layer optimization for high speed adders: A pareto driven machine learning approach," *IEEE TCAD*, 2018.
[3] W. Haaswijk, E. Collins, B. Seguin, M. Soeken, F. Kaplan, S. Süsstrunk, and G. De Micheli, "Deep learning for logic optimization algorithms," in *Proc. ISCAS*, 2018, pp. 1–4.
[4] Z. Xie, Y.-H. Huang, G.-Q. Fang, H. Ren, S.-Y. Fang, Y. Chen, and J. Hu, "RouteNet: Routability prediction for mixed-size designs using convolutional neural network," in *Proc. ICCAD*, 2018, pp. 80:1–80:8.
[5] H. Yang, S. Li, Y. Ma, B. Yu, and E. F. Young, "GAN-OPC: Mask optimization with lithography-guided generative adversarial nets," in *Proc. DAC*, 2018, pp. 131:1–131:6.
[6] N. Selvakkumaran and G. Karypis, "Multiobjective hypergraph-partitioning algorithms for cut and maximum subdomain-degree minimization," *IEEE TCAD*, vol. 25, no. 3, pp. 504–517, 2006.
[7] T.-H. Lee and T.-C. Wang, "Congestion-constrained layer assignment for via minimization in global routing," *IEEE TCAD*, vol. 27, no. 9, pp. 1643–1656, 2008.
[8] A. B. Kahng, C.-H. Park, X. Xu, and H. Yao, "Layout decomposition approaches for double patterning lithography," *IEEE TCAD*, vol. 29, pp. 939–952, June 2010.
[9] B. Yu, K. Yuan, D. Ding, and D. Z. Pan, "Layout decomposition for triple patterning lithography," *IEEE TCAD*, vol. 34, no. 3, pp. 433–446, March 2015.
[10] K.-T. Cheng and C.-J. Lin, "Timing-driven test point insertion for full-scan and partial-scan BIST," in *Proc. ITC*, 1995, pp. 506–514.
[11] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *Proc. ICLR*, 2016.
[12] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. NIPS*, 2017, pp. 1024–1034.
[13] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. KDD*, 2018, pp. 974–983.
[14] H. Cai, V. W. Zheng, and K. Chang, "A comprehensive survey of graph embedding: problems, techniques and applications," *IEEE TKDE*, vol. 30, no. 9, pp. 1616–1637, 2018.
[15] S. Bhagat, G. Cormode, and S. Muthukrishnan, "Node classification in social networks," in *Social network data analytics*. Springer, 2011, pp. 115–148.
[16] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proc. KDD*, 2016, pp. 855–864.
[17] B. Krishnamurthy, "A dynamic programming approach to the test point insertion problem," in *Proc. DAC*, 1987, pp. 695–705.
[18] A. J. Briers and K. A. E. Totton, "Random pattern testability by fast fault simulation," in *Proc. ITC*, 1986.
[19] M. Geuzebroek, J. T. Van Der Linden, and A. Van de Goor, "Test point insertion that facilitates atpg in reducing test time and data volume," in *Proc. ITC*, 2002, pp. 138–147.
[20] N. Tamarapalli and J. Rajski, "Constructive multi-phase test point insertion for scan-based bist," in *Proc. ITC*, 1996, pp. 649–658.
[21] J.-S. Yang, N. A. Touba, B. Nadeau-Dostie *et al.*, "Test point insertion with control points driven by existing functional flip-flops," *IEEE TC*, vol. 61, no. 10, pp. 1473–1483, 2012.
[22] N. A. Touba and E. J. McCluskey, "Test point insertion based on path tracing," in *Proc. VTS*, 1996, pp. 2–8.
[23] H. Ren, M. Kusko, V. Kravets, and R. Yaari, "Low cost test point insertion without using extra registers for high performance design," in *Proc. ITC*, 2009, pp. 1–8.
[24] H. F. Li and P. Lam, "A protocol extraction strategy for control point insertion in design for test of transition signaling circuits." in *Proc. GLSVLSI*, 1995, pp. 178–183.
[25] Z. Li, S. K. Goel, F. Lee, and K. Chakrabarty, "Efficient observation-point insertion for diagnosability enhancement in digital circuits," in *Proc. ITC*, 2015, pp. 1–10.
[26] L. H. Goldstein and E. L. Thigpen, "SCOAP: Sandia controllability/observability analysis program," in *Proc. DAC*, 1980, pp. 190–196.
[27] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. ICML*, 2010, pp. 807–814.