DGCNN: A Convolutional Neural Network over Large-scale Labeled Graphs

Anh Viet Phan[a,b], Minh Le Nguyen[a,*], Yen Lam Hoang Nguyen[b], Lam Thu Bui[b]

[a] *Japan Advanced Institute of Information Technology (JAIST), Nomi city, Japan 923-1211*
[b] *Le Quy Don Technical University, 236 Hoang Quoc Viet St., Ha Noi, Viet Nam*

Abstract

Exploiting graph-structured data has many real applications in domains including natural language semantics, programming language processing, and malware analysis. A variety of methods has been developed to deal with such data. However, learning graphs of large-scale, varying shapes and sizes is big challenges for any method. In this paper, we propose a multi-view multi-layer convolutional neural network on labeled directed graphs (DGCNN), in which convolutional filters are designed flexibly to adapt to dynamic structures of local regions inside graphs. The advantages of DGCNN are that we do not need to align vertices between graphs, and that DGCNN can process large-scale dynamic graphs with hundred thousands of nodes. To verify the effectiveness of DGCNN, we conducted experiments on two tasks: malware analysis and software defect prediction. The results show that DGCNN outperforms the baselines, including several deep neural networks.

Keywords: Labeled Directed Graphs, Convolutional Neural Networks (CNNs), Control Flow Graphs (CFGs), Abstract Syntax Trees (ASTs)

1. Introduction

For the last decades, graph mining has gained much attention due to significant applications in practice. For instance, several studies utilized parse

*Corresponding author

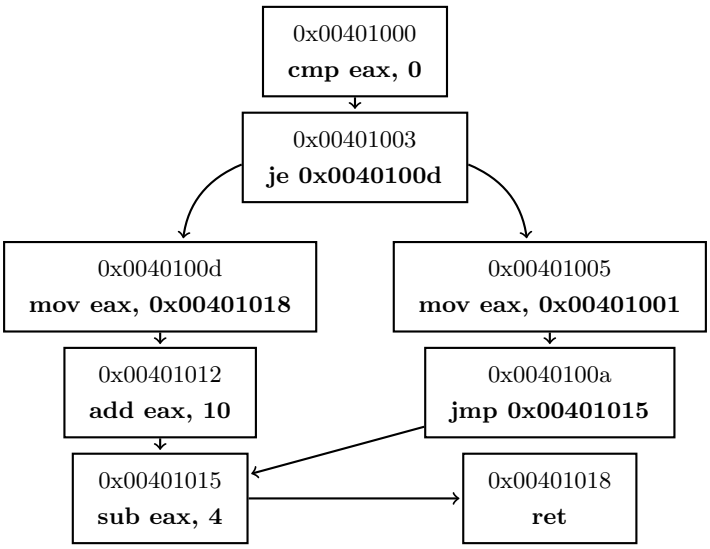


Figure 1: The Control Flow Graph of an assembly code fragment.

trees of sentences or graphical representations of documents for solving natural language processing (NLP) problems [1, 2, 3]. In biological studies, molecular graphs serve as inputs of machine learning techniques to predict the properties that are used to discover new drugs [4, 5]. In fact, the graphical representations of such problems are very diverse in terms of the data types and information which are contained in vertices and edges. Therefore, learning algorithms are designed to be compatible with the characteristics of graphs. This research focuses on formulating and solving problems based on labeled graph representations. Each object is represented as a labeled graph in which the vertices are the components, and the edges show the relations between components of the object. Fig. 1 illustrates an example of the graph representation called Control Flow Graph (CFG) of an assembly code fragment. In the CFG, each vertex corresponds to an instruction and directed edges indicate the execution orders of the instructions. For such types of representation, the contents of vertices and the relations between them reveal the characteristics of objects.

The traditional approaches to classifying labeled graphs can be divided into two directions: frequent subgraph mining (FSG) and graph-based ker-

nels [6]. FSG methods determine the set of subgraphs occurring frequently in the database. Then each instance is constructed as a feature vector based on the presence or absence of a particular subgraph. Finally, machine learning techniques like Support Vector Machines (SVMs), or AdaBoost are adopted to classify these feature vectors [7, 8, 9]. Graph kernels like random walk kernels [10], tree kernels [11], and cyclic pattern kernels [12] are used to measure the similarity between two arbitrary graphs [13].

However, adapting the existing approaches to real-world problems is a big challenge. The research questions are: can algorithms take advantages of various information of vertex labels, and can they tackle huge graphs with several hundred thousand vertices and edges as well. Take CFG as an example, each vertex contains some components of the instruction including the address, instruction name, and operands. Hence finding an appropriate method to represent information of vertices is an essential task to enhance the performance of learning algorithms. In addition, a CFG may have up to hundred thousands of vertices and edges. These are major obstacles to frequent subgraph mining and graph kernel approaches regarding computational time and required memory [10]. To measure the similarity between graphs, subgraph matching is a NP-complete problem, the runtime may grow exponentially with the number of nodes. Although graph kernels are simpler, they run in polynomial time and memory. Consequently, these methods are only applicable to the graphs having less than *hundreds of nodes* [14].

To address the research questions, we propose a multi-view multi-channel convolutional neural network on labeled directed graphs (DGCNN)[1]. By applying flexible convolutional filters and dynamic pooling, DGCNN is able to work on large-scale graphs having up to hundred thousands of nodes. The interesting points are that DGCNN learns directly on dynamic structures without padding and alignment of nodes; it has the time complexity of $O(n \times d)$ being

[1]The source code and collected datasets are publicly available at `https://github.com/nguyenlab/DGCNN`

proportional to the number of nodes and node degrees, and a fixed amount of memory, around 3MB of model parameters. In addition, like other deep neural networks, the network can simultaneously treat diverse information of components in graphs. We explain the points in Section 3. DGCNN is a generic architecture for labeled graphs. Our experiments show its effectiveness in two tasks: software defect prediction and malware analysis. The main contributions of this research can be summarized as follows:

- Proposing a multiview convolutional neural network on labeled directed graphs.
- Proving experimentally the effectiveness of DGCNN in dealing with large-scale graphs with high accuracy.
- Formulating an end-to-end approach to software defect prediction by applying DGCNN on control flow graphs.

The rest of the paper is organized as follows: Section 2 presents some relevant studies. The labeled directed graph definition and the details of the DGCNN architecture are provided in Section 3. Section 4 presents the experiments to verify the performance DGCNN in terms of accuracy and dealing with large-scale graphs. Finally, we conclude in Section 5.

2. Related Work

Graph data are known as one of complex structures and they have different types of representations. Various algorithms have been developed to tackle graph data. For labeled directed graphs, the existing approaches are based on subgraph isomorphism and graph kernels. However, these approaches not only are time and memory consuming but also require several constraints. Such problems lead to obstacles to adapting to practical applications. Gartner et al. proved that measuring graph similarity using subgraph isomorphism is NP-hard; the runtime grows exponentially with regard to the number of vertices. Although graph kernels are more efficient alternatives, measuring the similarities

between graphs is computed in polynomial time. For graphs with n nodes and m edges, random walk kernels originally proposed by Gartner on labeled directed graphs have the running time of $O(n^6)$ [15]. Then, Vishwanathan et al. speeded up to $O(n^3)$ [10]; and, Kang et al. reduced the time complexity to $O(n^2)$ [14]. Several methods for computing graph kernels require the same number of graph nodes, and $O(m^2)$ memory. Because of above drawbacks, these algorithms are infeasible for graphs with more than *hundreds of nodes* [14].

Our work is diverse from several newly proposed convolutional neural networks on graphs such as graph-based convolutional neural networks for image classification [16, 17, 18], for text categorization [16], for classifying chemical compounds [19, 17]. These networks aim to solve problems that data samples are represented as weighted graphs. Applying these networks to several types of labeled graphs is impractical because the graphs must be converted into adjacency matrices. To obtain adjacency matrix representations, each node in a graph is assigned to a unique identifier and all graphs must share a set of identifiers. However, an instruction may appear at many locations in a CFG. This leads to the unknown of the common identifier set because it is impossible to align the vertices between CFGs. Indeed, given CFG_1 and CFG_2 with the sets of vertices $V_1 = \{cmp, mov, add\}$ and $V_2 = \{mov, mov, jmp\}$, we can not determine the node of the CFG_2 that corresponds the node *move* of the CFG_1.

Closely related are the works of Mou et al. [20] and Duvenaud et al. [21]. Mou et al. developed a tree-based convolutional neural network (TBCNN) to process abstract syntax trees (ASTs) of programming languages. ASTs can be viewed as a specific type of labeled directed graphs without cycles. Besides, TBCNN considers the position of each node to determine the corresponding weights in the convolution stage. Therefore, it is impossible to adapt TBCNN for graphs with cycles and arbitrary orders of nodes. Duvenaud et al. designed a graph-based architecture to encode invariant substructures in a molecule. Each molecule is represented as a graph with nodes being individual atoms and edges being bonds. To deal with dynamic structures of local regions in convolution stage, they combine the vectors the current node and neighbors by element-

wise summation to form a single input vector. The weights of the input vector is determined according to the number of neighbors. Specifically, five sets of weights are used for local regions having 1-5 neighbors with the assumption that an atom has a maximum of 5 bonds. Extending this network to graphs in other domains faces some challenges. Unlike the graph of an atom, these graphs may have nodes with numerous neighbors. Thus, the network requires a huge number of weights causing overfitting. Moreover, embedding all node vectors into a single one ignores the relations in the substructures.

Our proposed network is a general framework for labeled graphs. To tackle dynamic substructures, we apply a shared parameter model in which the nodes having the same type share a set of weights and do not share any connections. We consider three types of nodes in a subgraph including the center, incoming and outgoing. In addition, our design can take multiple views of nodes to enrich the data for learning.

3. Approaches

This section describes a multi-view multi-channel convolutional neural network (DGCNN) for labeled directed graph classification. Firstly, we formulate the graph classification problem.

A labeled directed graph is defined as $G = (V, E, \alpha)$ where V is the set of vertices, $E \subseteq V \times V$ is the set of directed edges, α is the vertex labeling function $\alpha : V \to \Sigma_V$ where Σ_V is the content of vertex labels. Given a set of training examples $T = \{(x_i, y_i)\}_{i=0}^{L}$ where $x_i \in \mathcal{X}$ is a graph, and $y_i \in \mathcal{Y} = \{+1, -1\}$ is a target label, the graph classification problem is to induce the mapping $f : \mathcal{X} \to \mathcal{Y}$

3.1. Convolutional Neural Networks on Directed Graphs

DGCNN is a general neural network architecture designed to treat directed graphs with vertex labels containing complex information. For example, in the CFG, each vertex is an instruction which may involve the instruction name,

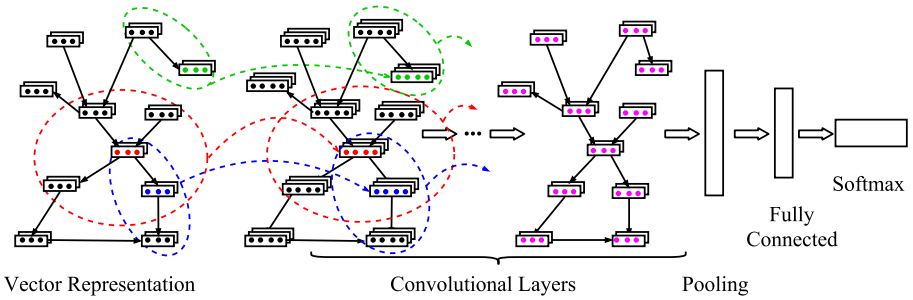


Figure 2: The architecture of the multi-layer convolutional neural network on graphs. Several steps of a convolution process is illustrated in the two first layers. The same color (red, green, or blue) of a node and an ellipse indicates the current position and the range of the filter.

and several operands. Moreover, each instruction can be viewed in not only its contents but also other perspectives including instruction types or functions. Each information type of a vertex label is called a view. To leverage all available information corresponding to such characteristics of the graphs, the multi-view multi-layer convolutional neural network on directed graphs is developed.

Fig. 2 demonstrates the overview architecture of DGCNN. The first layer is used to generate vector representations (also called embeddings) for graph vertices, where each view of a vertex label is mapped into a real-valued vector in a n_f-dimensional space. Next several convolutional layers are stacked on the embedding layer to extract the features from different parts of the graph. We thereafter apply a dynamic pooling layer to gather extracted features over the entire graph before feeding to a fully-connected layer. Finally, an output layer is added to compute the categorical distributions for possible outcomes. For multi-class classification problems, softmax is selected as the activation function to convert final scores to probabilities of observing labels. In the remainder of this section, we explain in details about major layers of DGCNN including vector representations, convolutional and pooling layers.

3.2. Vector Representations

The aim of this layer is representing each symbol as a real-valued vector in the n_f-dimensional space. It should be noted that a vertex may contain a set of vectors corresponding to the number of its views. The symbol vector representations are initialized randomly.

3.3. Convolutional Layers

In a convolutional layer, we apply a set of circular filters with radius R sliding over graph structures to extract features for all locations on the graphs. Because each vertex contains several views of its label, the filters extend through the full views of the input volumes. For example, given a filter with $R = 2$, and a graph with 3 views, at each position, the filter is designed such that it covers a subgraph containing the current vertex and the neighbors, and extends to depth 3. In other words, each neuron in the convolutional layer is connected to a local region (subgraph) of the input and the connectivity is extended along edges and views.

Formally, during the forward pass, each filter slides through all vertices of the graph and computes dot products between entries of the filter and the input. Suppose that the subgraph in the sliding window includes $d + 1$ vertices (the current vertex and its neighbors) with vector representations of $x_0, x_1, ..., x_d \in \mathbb{R}^{v_f \times n_f}$, then the output of the filters is computed as follows:

$$y = tanh(\sum_{i=0}^{d} \sum_{j=1}^{v_f} W_{conv,i,j} \cdot x_{i,j} + b_{conv}) \tag{1}$$

where $y, b_{conv} \in \mathbb{R}^{v_c \times n_c}, W_{conv,i} \in \mathbb{R}^{v_c \times n_c \times v_f \times n_f}$. $tanh$ is the activation function. n_f and v_f are the vector size and the number of views of the input layer. n_c and v_c are the numbers of filters and views of the convolutional layer.

The problem is that because of arbitrary structures of graphs, the numbers of vertices in subgraphs are different. As can be seen in Fig.2, the current receptive field at the red node includes 5 vertices while only 3 vertices are considered if the window moves right down. Consequently, determining the number of weight

matrices for filters is unfeasible. To deal with this obstacle, we divide vertices into groups and treat items in each group in a similar way. Regarding the way, the parameters for convolution have only three weight matrices including W^{cur}, W^{in}, and W^{out} for current, outgoing, and incoming nodes, respectively.

In the model, we stack several convolutional layers to broaden the area for extracting features of input graphs. Convolution preserves the input structures by using filters sliding over the entire graph. For this reason, the design procedure for all convolutional layers is the same. In the experiments, the networks have two convolutional layers with one or two views in the first convolution and one view in the second convolution. The filter sizes are set to 2. This means that at each position, considered objects involve the current vertex and its neighbors. To sum up, the set of parameters for DGCNN is $\theta = \{[W^{cur}_{conv1}, W^{in}_{conv1}, W^{out}_{conv1}]_{v_i}, [W^{cur}_{conv2}, W^{in}_{conv2}, W^{out}_{conv2}], W_{hid}, W_{output}, [b_{conv1}]_{v_i}, b_{conv2}, b_{hid}, b_{output}\}$, where v_i is the number of views of the input data.

3.4. Dynamic Pooling

Convolutions preserve the spatial relationship between vertices by learning graph features using circular filters. After convolutions, the structure of the output is completely the same as that of the original one. Thus, the extracted features can not be fed directly to the fully-connected layer because of enormous and varying numbers among different graphs. An efficient solution to this problems is applying dynamic pooling [22] to normalize the features such that they have the same dimension.

In the model, we use one-way max pooling to gather the information from all parts of the graph to one fixed size vector regardless of graph shapes and sizes. The vector dimension is the number of filters in the last convolutional layer. Basically, in convolutional neural networks, pooling layers are applied to reduce the dimensionality of each feature map (the output of one filter) but retain the most important information. Pooling layers operate independently on every dimension of its input and resize the input spatially using an operation. Some types of operations are max, average, and sum. In the case of max pooling,

the maximum value in each dimension is selected from the features. Instead of taking the largest element we could also take the average (average pooling) or the sum of all elements (sum pooling) in that window. In practice, max pooling has been shown to work better [23, 24, 25]. Therefore, the max pooling is adopted in DGCNN.

3.5. Training

We use mini-batch gradient descent algorithm for training the network. The objective is to minimize the mean square error loss function as follows:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2}$$

where θ is the model parameters (Subsection 3.3), y_i is the label of data sample i, and $\hat{y}_i$ is the output of the network.

The training procedure is shown in the pseudo-code in algorithm 1. Firstly, the model parameters θ are randomly initialized. The training process is performed through a pre-defined number of epochs. For each loop, we calculate the loss function $J^{(i)}$ for each data sample $x^{(i)}$ separately according to Eq. 2. Then, back propagation algorithm is applied to compute the partial derivatives and evaluate the gradient. The model parameters are updated every mini-batch of n training examples.

3.6. Computational Complexity and Required Memory

By applying filters with flexible design, and dynamic pooling, DGCNN does not require any preprocessing such as padding to ensure graphs with the same number of nodes, and alignment to match corresponding nodes between graphs. Indeed, from the two first layers of the model (Fig. 2), filters slide over the entire graph and extract subgraph features at each location independently regardless of graph structures. Additionally, computing the feature map does not require any order of nodes (Eq. 1). Thus, the model can treat dynamic graphs and matching nodes among graphs is unnecessary.

Algorithm 1: Mini-batch gradient descent algorithm

Input : Data samples $x^{(i)}$, $i = 1..N$;
Learning rate η;
Batch size n;

Output: Model parameters $\theta = \{W, B\}$

Randomly initialize θ, $\Delta\theta \leftarrow 0$;
for $l \leftarrow 1$ **to** *nb_epochs* **do**
  for $i \leftarrow 1$ **to** N **do**
    compute loss $J^{(i)}$;
    compute the partial derivative $\frac{\partial J^{(i)}}{\partial \theta}$;
    $\Delta\theta \leftarrow \Delta\theta$ + evaluate gradient($\frac{\partial J^{(i)}}{\partial \theta}$);
    if $i\%\ n = 0$ *or* $i = N$ **then**
      $\theta \leftarrow \theta - \eta\Delta\theta$;
      $\Delta\theta \leftarrow 0$;
    end
  end
end

DGCNN uses a constant amount of memory to store model parameters, and its runtime grows proportionally with the number of vertices, and vertex degrees. According to Eq. 1, the cost for computing the feature map in a convolutional layer is $(d_{max} + 1) \times v_f \times n$, where d_{max} is the maximum degree of graphs, v_f is the number of views, and n is the number of graph nodes. In the pooling layer, we need $O(n)$ comparisons to gather all extracted features into a fixed-size vector. For hidden and output layers, the numbers of operations are constant. Intuitively, c and v_f are constant, with assumption of $d_{max} \ll n$ in large-scale graphs, the runtime complexity of DGCNN is $O[c \times (d_{max}+1) \times v_f \times n] + O(n) + O(1) = O(n \times d)$.

4. Experiments

To verify the performance of DGCNN in terms of accuracy and the ability to process large-scale graphs, we conduct experiments on two tasks including software defect prediction and malware analysis. To solve these problems, each data sample is converted into a directed graph of control flow (CFG) [26], and then DGCNN is utilized to build predictive models.

For software defect prediction, we formulate an end-to-end model with high accuracy to predict the existence of defects in programming source code. For malware analysis, DGCNN shows the ability to handle CFGs with hundred thousands of nodes and edges.

We ran experiments on two systems including one node of a Fujitsu CX250 Cluster and one node of a SGI UV3000. For the Fujitsu CX250 Cluster, each node has two Intel Xeon processors E5-2680v2 2.80 GHz with ten cores, 64GB of RAM. For the SGI UV3000, each node has two Intel Xeon processors E5-4655v3 2.9GHz with six cores, 256GB of RAM.

4.1. Control Flow Graphs

A control flow graph (CFG) is a labeled directed graph, G = (V, E, α) (Section 3). In CFGs, each $v \in V$ represents a basic block that is a linear sequence

of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed); and $(v_i, v_j) \in E$ shows the control flow path from block v_i to block v_j(Fig. 1).

CFG analysis has been widely used for various problems because of showing execution sequences of programs. Compilers perform program optimization by determining the flow relationships based on within graph analysis [26]. Many studies have focused on mining CFGs to tackle the difficulties in malware analysis [27, 28], software plagiarism [29, 30]. For the two tasks in this study, software defects are hidden deeply in source code and only revealed while programs are running; malware uses different obfuscation techniques to prevent detection by anti-virus applications. Since CFGs show the behavior of programs, learning on CFGs may be beneficial for distinguishing patterns.

4.2. Two views of data

The labels of the CFGs' nodes in both tasks are assembly instructions[2]. To enrich data for learning, we use two views of data including instruction names and instruction groups. For instance, instructions `jne`, `jle`, and `jge` are tagged to the same group since they are conditional jump instructions. Similarly, `addb`, `addl`, and `addw` belong to the group of arithmetic instructions.

It should be noted that an assembly instruction may have several operands. To take advantages of all information, the vector representation of each instruction is computed based on those of its components. Firstly, operands of block names, processor register names, and literal values are substituted by the symbols "name", "reg", and "val", respectively. Corresponding to the replacement, the instruction `addq $32, %rsp` has the form of `addq, value, reg`. After that, the vector of the instruction is determined as follows:

$$x = \begin{cases} x_{instruction\ token} \text{ (NoOp - without the use of operands)} \\ \frac{1}{C}\sum_{j=1}^{C} x_j \text{ (Op - with the use of operands),} \end{cases} \tag{3}$$

[2] `https://docs.oracle.com/cd/E19253-01/817-5477/817-5477.pdf`

where C is the number of the components, and x_j is the vector of the j^{th} component.

4.3. Software defect prediction

4.3.1. Task description

Software defect prediction is one of the hot topics in the field of software engineering and has important applications. The task is to analyze source files to detect potential buggy code. Deploying software products containing defects may cause serious consequences such as loss of money, time, and business credibility. For a large project, manually investigating defects may be time-consuming because the project contains not only a large number of source files but also many logic connections among its components. Therefore, building automatic systems for bug detection and localization is an urgent requirement in software industry. This helps to reduce the development efforts, and enhance the quality and reliability of software products.

In this study, we formulate a new approach to predict the existence of defects in source codes written in a programming language. Our proposed approach involves two steps: 1) generating the CFG representation, and 2) building classifiers. In the first step, each source file is compiled into an assembly code using g++ on Linux. The CFG thereafter is constructed to describe the execution flows of the assembly instructions. The second step leverages DGCNN to automatically learn defect features on CFG data.

4.3.2. CFG construction

The CFG of a program is generated from its assembly code after compiling. Fig. 3 illustrates an example of the control flow graph constructed from an assembly code snippet, in which each vertex corresponds to an instruction and a directed edge shows the execution path from an instruction to the other.

The pseudo-code to generate CFGs is shown in Algorithm 2. The algorithm takes an assembly file as the input, and outputs the CFG. Building the CFG from

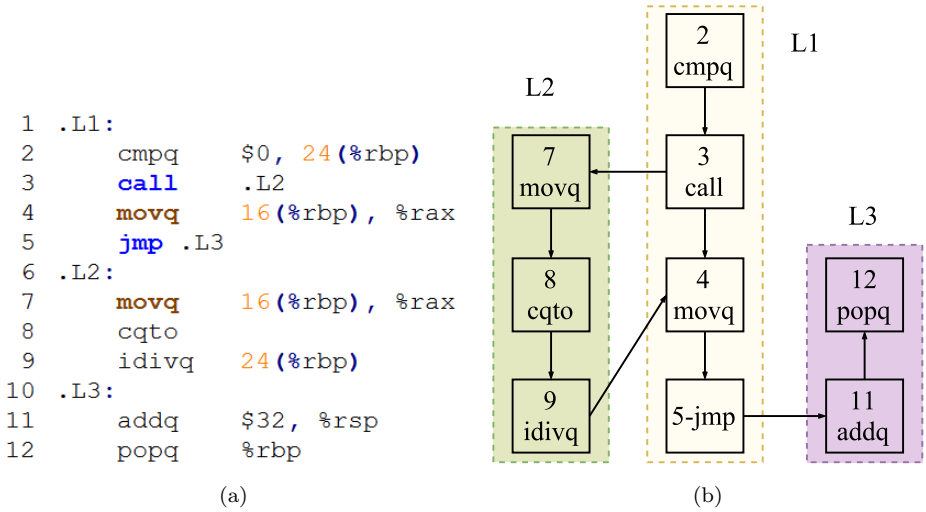


Figure 3: An example of constructing CFGs from assembly code.(3a) a fragment of assembly code;(3b) the CFG of the code fragment (each node is viewed by the line number and the name of the instruction).

an assembly code includes two major steps. In the first step, the code is partitioned into blocks of instructions based on the labels (e.g. `L1, L2, L3` in Fig. 3). The second step is creating the edges to represent the control flow transfers in the program. Specifically, the first line invokes procedure `initialize_Blocks` to read the file contents and return all the instruction blocks. In line 2, the set of edges is initially set to empty. From line 3 to 24, the graph edges are created by traversing all instructions of each block and considering possible execution paths from the current instruction to others. For a block, because the instructions are executed in sequence, every node has an outgoing edge to the next one (line 5-9). Additionally, we consider two types of instructions which may have several targets. For `jump` instructions, an edge is added from the current instruction to the first one of the target block. We use two edges to model function calls, in which one is from the current node to the first instruction of the function and the other is from the final instruction of the function to the next instruction of the current node (line 10-24). Finally, the graphs are formed

from the instruction and edge sets (line 25-26).

4.3.3. Datasets

The datasets were collected from a popular programming contest site CodeChef [3]. We created four benchmark datasets which each one involves source code submissions (written in C, C++, Python, etc.) for solving one of the problems as follows:

- SUMTRIAN (Sums in a Triangle): Given a lower triangular matrix of n rows, find the longest path among all paths starting from the top towards the base, in which each movement on a part is either directly below or diagonally below to the right. The length of a path is the sums of numbers that appear on that path.
- FLOW016 (GCD and LCM): Find the greatest common divisor (GCD) and the least common multiple (LCM) of each pair of input integers A and B.
- MNMX (Minimum Maximum): Given an array A consisting of N distinct integers, find the minimum sum of cost to convert the array into a single element by following operations: select a pair of adjacent integers and remove the larger one of these two. For each operation, the size of the array is decreased by 1. The cost of this operation will be equal to their smaller.
- SUBINC (Count Subarrays): Given an array A of N elements, count the number of non-decreasing subarrays of array A.

The target label of an instance is one of the possibilities of source code assessment. Regarding this, a program can be assigned to one of the groups as follows: 0) accepted - the program ran successfully and gave a correct answer; 1) time limit exceeded - the program was compiled successfully, but it did not

[3] `https://www.codechef.com/problems/<problem-name>`

Algorithm 2: The algorithm for constructing Control Flow Graphs from assembly code

Input : asm_file - A file of assembly code

Output: The graph representation of the code

```
1  blocks ← initialize_Blocks(asm_file);
2  edges ← {};
3  for i ← 0 to |blocks| do
4      for j ← 0 to |blocks[i].instructions| do
5          if j > 0 then
6              inst_1 ← blocks[i].instructions[j − 1];
7              inst_2 ← blocks[i].instructions[j];
8              edges.add(new_Edge(inst_1, inst_2));
9          end
10         if inst_1.type="jump" or inst_1.type="call" then
11             label ← inst_1.params[0];
12             to_block ←find_Block_by_Label(label);
13             if to_block ≠ NULL then
14                 inst_2 ← to_block.first_instruction;
15                 edges.add(new_Edge(inst_1, inst_2));
16                 if inst_1.type="call" then
17                     inst_2 ← to_block.last_instruction;
18                     inst_1 ← inst_1.next;
19                     edges.add(new_Edge(inst_2, inst_1));
20                 end
21             end
22         end
23     end
24 end
25 instructions ← get_All_Instructions(blocks);
26 return construct_Graph(instructions, edges);
```

Table 1: Statistics of CodeChef datasets. The values are shown in form of average± standard deviation.

Dataset	Class					Nodes	Max nodes	Degree	Max degree
	0	1	2	3	4				
FLOW016	3,472	4,165	231	2,368	412	117±55	1,246	3.39±0.80	11
MNMX	5,157	3,073	189	113	213	211±296	3,073	3.72±1.86	43
SUBINC	3,263	2,685	206	98	232	142±63	1,245	3.19±0.63	17
SUMTRIAN	9,132	6,948	419	2,701	1,987	236±104	2,905	3.75±1.36	56

stop before the time limit; 2) wrong answer: the program compiled and ran successfully but the output did not match the expected output; 3) runtime error: the code compiled and ran but encountered an error due to reasons such as using too much memory or dividing by zero; 4) syntax error - the code was unable to compile.

We collected all submissions written in C or C++ until March 14th, 2017 of four problems. The data are preprocessed by removing source files which are empty code, and unable to compile. To conduct experiments, each dataset is randomly split into three folds for training, validation, and testing by ratio 3:1:1.

Table 1 presents statistical figures of instances in each class of the datasets. All of the datasets are imbalanced. Taking MNMX dataset as an example, the ratios of classes 2, 3, 4 to class 0 are 1 to 27, 46, and 24. In addition, programs' CFGs vary considerably in size with the number of nodes from hundreds to thousands.

4.3.4. Baselines

We compare DGCNN with various approaches including a tree based convolutional neural network (TBCNN) - a state-of-the-art deep neural model for this problem [20], sibling-subtree convolutional neural networks (SibStCNN), recursive neural networks (RvNN) [31], k nearest neighbors (kNN) with tree edit distance (TED) and Levenshtein distance (LD) [32], and support vector

Table 2: Structures and numbers of hyperparameters of the neural networks. Each layer is presented in form of the name followed by the number of neurons. Emb is a embedding layer. Rv, TC, GC, and FC stand for recursive, tree-based convolutional, graph-based convolutional, and fully-connected, respectively.

Network	Architecture	weights	biases
RvNN	Coding30-Emb30-Rv600-FC600-Soft5	1,104,600	1,235
TBCNN	Coding30-Emb30-TC600-FC600-Soft5	1,140,600	1,235
SibStCNN	Coding30-Emb30-TC600-FC600-Soft5	1,140,600	1,235
DGCNN-1V	GC100-GC600-FC600-Soft5	552,000	1,305
DGCNN-2V	GC100-GC600-FC600-Soft5	561,000	1,305

machines (SVMs) with bag-of-words (BoW) features. The parameters for the models are described as follows.

The neural networks. The structures of the networks are shown in Table 2. The networks share some initial parameters: the learning rate is 0.1, the token vectors have a size of 30, the batch size is 10.

k-nearest neighbors (kNN). The number of neighbors k is selected from $\{3, 5, 7, 9\}$. We found that $k = 3$ commonly reaches the highest performance on the validation sets.

SVM-BoW. Two parameters C and γ of the SVM with RBF kernel are tuned by using grid search.

4.3.5. Results

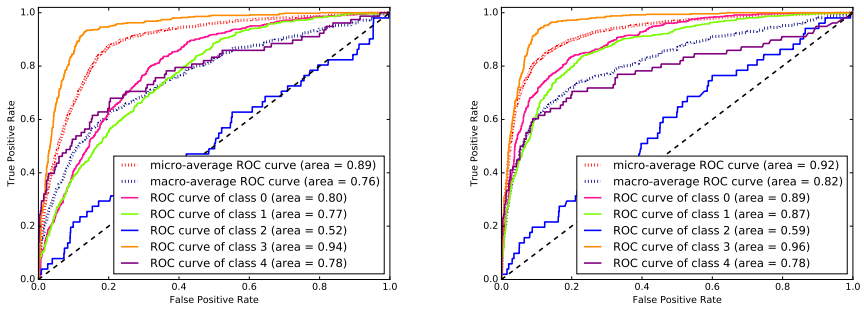
Table 3 shows the accuracies of classifiers on the four datasets. Building the network models including the steps of training, validation and testing was within 24 hours. As can be seen, CFG-based approaches significantly outperform others. Specifically, in comparison with the second best, they improve the accuracies by 12.39% on FLOW016, 1.2% on MNMX, 7.71% on SUBINC, and 1.98% on SUMTRIAN. Software defect prediction is a complicated task because semantic errors are hidden deeply in source code. Even if a defect exists in a program, it is only revealed during running the application under specific condi-

tions [33]. Therefore, it is impractical to manually design a set of good features which are able to distinguish faulty and non-faulty samples. Similarly, ASTs just represent the structures of source code. Although tree-based approaches (SibStCNN, TBCNN, and RvNN) are successfully applied to other software engineering tasks like classifying programs by functionalities, they have not shown good performance on the software defect prediction. In contrast, CFGs of assembly code is precise graphical structures which show behaviors of programs. As a result, applying DGCNN on CFGs achieves the highest accuracies on the experimental datasets about software defects.

Table 3: Comparison of classifiers according to accuracy, F1, and AUC measures. 1V and 2V following ASCNN means that an instruction are viewed by one and two perspectives. Op and NoOp are using instructions with or without operands.

Approach	FLOW016		MNMX		SUBINC		SUMTRIAN	
	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1
SVM-BoW	60.00	58.64	77.53	75.00	67.23	65.75	64.87	63.82
LD	60.75	60.61	79.13	77.89	66.62	66.36	65.81	65.73
TED	61.69	61.56	80.73	79.55	*68.31**	*68.03**	*66.97**	*66.83**
RvNN	61.03	58.98	82.56	80.48	64.53	62.07	58.82	56.29
TBCNN	*63.10**	*61.85**	82.45	80.94	63.99	62.13	65.05	63.35
SibStCNN	62.25	61.15	*82.85**	*81.04**	67.69	65.15	65.10	63.20
GCNN_1V_NoOp	73.80	72.57	83.19	81.28	70.93	69.61	68.83	**67.33**
GCNN_2V_NoOp	74.32	73.11	83.82	**82.32**	74.02	72.54	68.12	66.42
GCNN_1V_Op	**75.49**	**74.03**	**84.05**	82.28	72.40	70.67	68.19	65.91
GCNN_2V_Op	75.12	73.60	83.70	81.88	**76.02**	**74.42**	**68.95**	66.62

From the last four rows of Table 3, the more the information is provided, the more efficient the learner is. In general, viewing graph nodes by two perspectives including instructions and instruction groups helps boost DGCNN classifiers in both cases: with and without the use of operands. Similarly, taking into account of all components in instructions (Eq. 3) is beneficial. In this case, the DGCNN



(a) (b)

Figure 4: The illustration of the discrimination ability between classes of classifiers on imbalanced datasets. Fig. 4a and Fig. 4b are the ROC curves of TBCNN and DGCNN_1V_NoOp on FLOW016 dataset, respectively.

models achieve highest accuracies on the experimental datasets. Specifically, DGCNN with one view reaches the accuracies of 75.49% on FLOW016, and 84.05% on MNMX; DGCNN with two views obtains the accuracies of 76.02% on SUBINC, and 68.95% on SUMTRIAN.

We also assess the effectiveness of the models in terms of the discrimination measure (AUC) which is equivalent to Wilcoxon test in ranking classifiers. For imbalanced datasets, many learning algorithms have a trend to bias the majority class due to the objective of error minimization. As a result, the models mostly predict an unseen sample as an instance of the majority classes, and ignore the minority classes. Fig. 4 plots the ROC curves of TBCNN and DGCNN_1V_NoOp classifiers on FLOW016 dataset, an imbalanced data with the minority classes of 2 and 4. Both two classifiers have a notable lower ability in detecting minority instances from the others. For predicting class 4, the TBCNN is even equivalent to a random classifier. After observing the other ROC curves we found the similar problem for all of the approaches on the experimental datasets. Thus, AUC is an essential measure for evaluating classification algorithms, especially in the case of imbalanced data.

Table 4: Performance comparison in terms of the AUC measure.

Approach	FLOW016	MNMX	SUBINC	SUMTRIAN
SVM-BoW	0.74	0.76	*0.73**	0.79
RvNN	0.75	*0.79**	0.69	0.73
TBCNN	*0.76**	0.77	0.72	0.78
SibStCNN	*0.76**	*0.79**	0.71	*0.80**
GCNN_1V_NoOp	**0.82**	**0.82**	0.74	**0.82**
GCNN_2V_NoOp	0.80	0.81	0.72	0.81
GCNN_1V_Op	0.81	0.80	**0.75**	0.81
GCNN_2V_Op	**0.82**	0.79	0.74	0.81

Table 4 presents the AUCs of probabilistic classifiers, which produce the probabilities or the scores to indicate the belonging degrees of an instance to classes. There are two groups including graph-based and tree-based approaches, in which the approaches in each group has the similar AUC scores; and graph-based approaches show better performance than those of tree-based. It is worth noticing that, along with the efforts of accuracy maximization, the approach based on DGCNN and CFGs also enhance the distinguishing ability between categories even on imbalanced data. The DGCNN classifier improves the second best an average of 0.03 on AUC scores.

Fig. 5 plots the learning curves of the networks on two datasets MNMX and FLOW106. For all networks, the validation accuracies quickly converge to the optimal values after around 20 epochs and vary around such values in next epochs. We can see three groups of networks based on the learning curves. The first is the DGCNN's variants which reach highest accuracies for both training and validation, and have lowest errors. The second includes SibStCNN and TBCNN which their curves are closed to each other. The third is RvNN with the lowest accuracies and highest errors. From above analysis, we can conclude that leveraging precise control flow graphs of binary codes is suitable for software defect prediction, and DGCNN is a deep neural network for learning on labeled

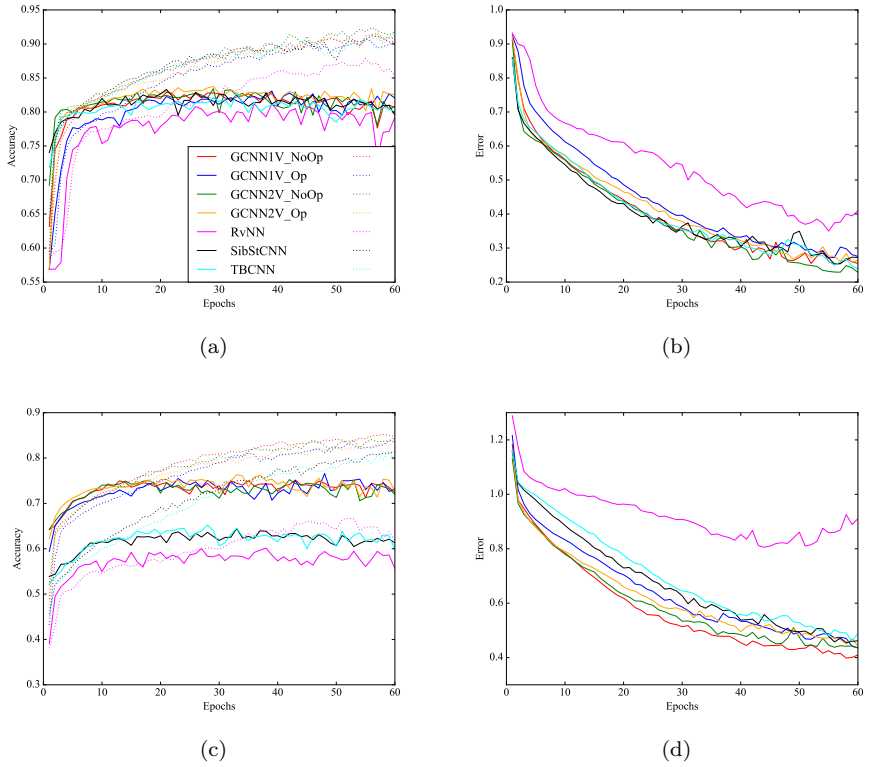


Figure 5: Learning curves of the networks. Fig. 5a and Fig. 5b, Fig. 5c and Fig. 5d are accuracy and error curves of MNMX and FLOW016 datasets; the solid and dot curves correspond to training and validation, respectively.

directed graphs efficiently.

4.3.6. Error Analysis

We analyze cases of source code variations which methods are able to handle or not based on observations on classifiers' outputs, training and test data. We found that RvNN's performance is degraded when tree sizes increase. This problem is also pointed out from other research on tasks of natural language processing [34, 31] and programming language processing [20]. From Tables 1, and 3 the larger trees, the lower accuracies and AUCs, RvNN obtains in comparison with other approaches, especially on SUMTRIAN dataset. SibStCNN and TBCNN obtain higher performance than other baselines due to learning features from subtrees. For analyzing tree-based methods in this section, we only take into account SibStCNN and TBCNN.

Effect of code structures: the tree-based approaches suffer from varying structures of ASTs. For example, given a program, we have many ways to reorganize the source code such as changing positions of some statements, constructing procedures and replacing statements by equivalent ones. These modifications lead to reordering the branches and producing new branches of ASTs (`File 3.c` and `File 4.c`). Because of the weight matrices for each node being determined based on the position, SibStCNN and TBCNN are easily affected by changes regarding tree shapes and sizes.

Meanwhile, graph-based approaches are able to handle these changes. We observed that although loop statements like `For`, `While`, and `DoWhile` have different tree representations, their assembly instructions are similar by using a jump instruction to control the loop. Similarly, moving a statement to possible positions may not result in notable changes in assembly code. Moreover, grouping a set of statements to form a procedure is also captured in CFGs by using edges to simulate the procedure invocation (Section 4.1).

Effect of changing statements: CFG-based approaches may be affected by replacements of statements. Considering source code in `File 3.c`, `File 5.c`, they have similar ASTs, but the assembly codes are different. In `C` language,

```c
int cal(int a,int b)
{
 if(a%b==0)
   return b;
 else
   cal(b,a%b);
}
int main()
{
  int t,lcm,gcd,a,b;
  scanf("%d",&t);
  while(t--)
  {
    scanf("%d%d",&a,&b);
    gcd=cal(a,b);
    lcm=(a*b)/gcd;
    printf("%d %d\n",gcd,lcm);
  }
  return 0;
}
```

(a) File 3.c (a training sample)

```c
int gcd(int a, int b)
{
  if (b == 0)
    return a;
  else
    return gcd(b, a % b);
}
int lcm(int a, int b)
{
  return (a*b)/gcd(a,b);
}
int main()
{
  int t, a, b;
  scanf("%d", &t);
  while(t--)
  {
    scanf("%d%d", &a, &b);
    printf("%d %d\n", gcd(a,b), lcm(a,b));
  }
  return 0;
}
```

(b) File 4.c (**G+, T-**)

```c
long int gcd(long int a,long int b)
{
    if (a==0) return b;
    return gcd(b%a,a);
}
int main() {
  int t;
  cin>>t;
  while(t--)
  {
     long int a,b,g,l;
     cin >> a >> b;
     g=gcd(a,b);
     l=(a*b)/g;
     cout<<g<<" "<<l<<endl;
  }
  return 0;
}
```

(c) File 5.c (**G-, T+**)

```c
#include<algorithm>
int main()
{
  int t;
  cin>>t;
  while(t--)
  {
    long int a,b,gcd,lcm;
    cin>>a>>b;
    gcd=__gcd(a,b);
    cout<<gcd<<" "<<((a*b)/gcd)<<endl;
  }
  return 0;
}
```

(d) File 6.c (**G-, T-**)

Figure 6: Some source code examples in FLOW016 dataset which may cause mistakes of tree-based (T) and CFG-based (G) approaches. Fig. 6a is a sample in the training set. Figs. 6b, 6c, and 6d are samples in the test set. Symbols "+" and "-" denote the sample is correctly and incorrectly classified by the approaches.

statements are translated into different sets of assembly instructions. For example, with the same operator, the sets of instructions for manipulating data types of `int` and `long int` are dissimilar. Moreover, statements are possible replaced by others without any changes of program outcomes. Indeed, to show values, we can select either `printf` or `cout`. Since contents of CFG nodes are changed significantly, DGCNN may fail in predicting these types of variations.

Effect of using library procedures: when writing a source code, the programmer can use procedures from other libraries. In Fig. 6, `File 6.c` applies the procedure `__gcd` in the library `algorithm`, while the others use ordinary C statements for computing the greatest common divisor of each integer pair. Both ASTs and CFGs do not contain the contents of external procedures because they are not embedded to generate assembly code from source code. As a result, tree-based and graph-based approaches are not successful in capturing program semantics in these cases.

4.4. Malware analysis

4.4.1. Task description

To verify the ability to deal with large scale graphs, we apply DGCNN for malware analysis. The task is to check whether an executable file is malware. To solve this problem, a file is represented as a directed graph of control flow using a disassembler tool called BE-PUM (Binary Emulation for Pushdown Model) [35]. After that, several graph-based approaches are employed graph-based approaches to classify the data samples into malware and non-malware.

The dataset includes x86 binary files, wherein malware samples are supplied by LORIA, Loraine University [4], and VX Heaven [5]; and, non-malware files were collected from the system files in the folder *Windows*. From the last row of Table 5, the control flow graphs of files are very large with the number of nodes up to greater than 150,000. Because the data is quite small, the experiment is

[4] `http://www.loria.fr/les-actus`

[5] `http://vxheaven.org/`

Table 5: Statistics on malware dataset.

Dataset	Total	#Neg.	#Pos.	#nodes	Max nodes	Degree	Max degree
MALWARE	2,937	1,362	1,575	1,236±5,528	157,237	11.21±55.44	1736

conducted using 5-fold cross validation.

4.4.2. Baselines

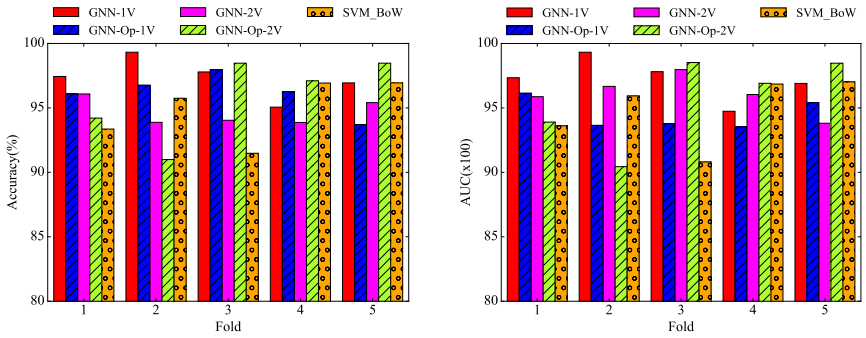
For malware analysis, we compare DGCNN with SVM-Bow. We also investigated several graph kernel approaches such as Shortest Path Kernels and random walk graph kernels. However, these algorithms are unable to be executed due to computational complexity and required memory. According to previous studies, the kernel methods just process graphs with hundreds of nodes [36, 10].

As mentioned in Subsection 3.6, DGCNN has $O(n \times d)$ time complexity, where n and d are the number of graph nodes and the max degree. In the case of malware dataset, the time complexity is equivalent to $O(n)$ because of $d \ll n$ (Table 5). Moreover, the required memory is about 3MB. As as results, both processes of training and testing DGCNN take only about 24 hours.

4.4.3. Results

Fig. 7 depicts the performance of DGCNN in comparison with SVM-BoW in terms of accuracy and AUC. BE-PUM can generate precise control flow graphs of executable files under the presence of obfuscation techniques [35]. Thus using SVM with BoW features also obtains high results. One problem is the huge sizes of graphs that lead to obstacles in applying conventional methods for graphs. We tried SVM with shortest path kernels and random walk graph kernels, but running them was failed because of computational complexity and required memory. Meanwhile training the DGCNN models and predicting the testing sets take around 24 hours.

GCNN-1V achieves the best performance with the average accuracy of 97.31%, and the average AUC of 97.22%. By applying convolution to capture graphs'



(a) (b)

Figure 7: Comparison of DGCNN and SVM according to accuracy (7a) and AUC (7b) using 5-fold cross validation. The symbol *Op* indicates the use of operands.

features, DGCNN is good at distinguishing between malware and non-malware files. Unlike the cases of software fault prediction, adding more information may degrade DGCNN's performance on the MALWARE dataset. This probably is caused by an increase of the number of tokens and the limited number of training instances. The numbers of unique tokens corresponding to the cases of with and without the use of operands are 1,427, 3,669, respectively, while the training data for each fold of cross validation contain about 2,350 samples.

5. Conclusion

In this paper, we introduced a graph-based convolutional neural network and applied to two problems including malware analysis and software defect prediction. The network can process large-scale graphs up to hundred thousands of nodes and edges without padding or alignment between samples. The experiments show that our methods obtained notable results in comparison with baselines including other deep neural networks.

References

[1] M. Looks, M. Herreshoff, D. Hutchins, P. Norvig, Deep learning with dynamic computation graphs, arXiv preprint arXiv:1702.02181.

[2] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated graph sequence neural networks, arXiv preprint arXiv:1511.05493.

[3] F. Tian, B. Gao, Q. Cui, E. Chen, T.-Y. Liu, Learning deep representations for graph clustering., in: AAAI, 2014, pp. 1293–1299.

[4] S. Kearnes, K. McCloskey, M. Berndl, V. Pande, P. Riley, Molecular graph convolutions: moving beyond fingerprints, Journal of computer-aided molecular design 30 (8) (2016) 595–608.

[5] H. Altae-Tran, B. Ramsundar, A. S. Pappu, V. Pande, Low data drug discovery with one-shot learning, arXiv preprint arXiv:1611.03199.

[6] N. S. Ketkar, L. B. Holder, D. J. Cook, Empirical comparison of graph classification algorithms, in: Computational Intelligence and Data Mining, 2009. CIDM'09. IEEE Symposium on, IEEE, 2009, pp. 259–266.

[7] M. Deshpande, M. Kuramochi, N. Wale, G. Karypis, Frequent substructure-based approaches for classifying chemical compounds, IEEE Transactions on Knowledge and Data Engineering 17 (8) (2005) 1036–1050.

[8] M. Kuramochi, G. Karypis, Frequent subgraph discovery, in: Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on, IEEE, 2001, pp. 313–320.

[9] X. Yan, J. Han, gspan: Graph-based substructure pattern mining, in: Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on, IEEE, 2002, pp. 721–724.

[10] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, K. M. Borgwardt, Graph kernels, Journal of Machine Learning Research 11 (Apr) (2010) 1201–1242.

[11] P. Mahé, J.-P. Vert, Graph kernels based on tree patterns for molecules, Machine learning 75 (1) (2009) 3–35.

[12] T. Horváth, T. Gärtner, S. Wrobel, Cyclic pattern kernels for predictive graph mining, in: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2004, pp. 158–167.

[13] C. Wagner, G. Wagener, R. State, T. Engel, Malware analysis with graph kernels and support vector machines, in: Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on, IEEE, 2009, pp. 63–68.

[14] U. Kang, H. Tong, J. Sun, Fast random walk graph kernel, in: Proceedings of the 2012 SIAM International Conference on Data Mining, SIAM, 2012, pp. 828–838.

[15] T. Gärtner, P. Flach, S. Wrobel, On graph kernels: Hardness results and efficient alternatives, in: Learning Theory and Kernel Machines, Springer, 2003, pp. 129–143.

[16] M. Defferrard, X. Bresson, P. Vandergheynst, Convolutional neural networks on graphs with fast localized spectral filtering, in: Advances in Neural Information Processing Systems, 2016, pp. 3837–3845.

[17] M. Simonovsky, N. Komodakis, Dynamic edge-conditioned filters in convolutional neural networks on graphs, arXiv preprint arXiv:1704.02901.

[18] M. Edwards, X. Xie, Graph based convolutional neural network, arXiv preprint arXiv:1609.08965.

[19] M. Niepert, M. Ahmed, K. Kutzkov, Learning convolutional neural networks for graphs, in: Proceedings of the 33rd annual international conference on machine learning. ACM, 2016.

[20] L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin, Convolutional neural networks over tree structures for programming language processing, arXiv preprint arXiv:1409.5718.

[21] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, R. P. Adams, Convolutional networks on graphs for learning molecular fingerprints, in: Advances in neural information processing systems, 2015, pp. 2224–2232.

[22] R. Socher, E. H. Huang, J. Pennington, A. Y. Ng, C. D. Manning, Dynamic pooling and unfolding recursive autoencoders for paraphrase detection., in: NIPS, Vol. 24, 2011, pp. 801–809.

[23] T. N. Sainath, R. J. Weiss, A. W. Senior, K. W. Wilson, O. Vinyals, Learning the speech front-end with raw waveform cldnns., in: INTERSPEECH, 2015, pp. 1–5.

[24] L. Liu, L. Wang, X. Liu, In defense of soft-assignment coding, in: Computer Vision (ICCV), 2011 IEEE International Conference on, IEEE, 2011, pp. 2486–2493.

[25] J. Yang, K. Yu, Y. Gong, T. Huang, Linear spatial pyramid matching using sparse coding for image classification, in: Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on, IEEE, 2009, pp. 1794–1801.

[26] F. E. Allen, Control flow analysis, in: ACM Sigplan Notices, Vol. 5, ACM, 1970, pp. 1–19.

[27] D. Bruschi, L. Martignoni, M. Monga, Detecting self-mutating malware using control-flow graph matching, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2006, pp. 129–143.

[28] B. Anderson, D. Quist, J. Neil, C. Storlie, T. Lane, Graph-based malware detection using dynamic analysis, Journal in computer virology 7 (4) (2011) 247–258.

[29] D.-K. Chae, J. Ha, S.-W. Kim, B. Kang, E. G. Im, Software plagiarism detection: a graph-based approach, in: Proceedings of the 22nd ACM international conference on Conference on information & knowledge management, ACM, 2013, pp. 1577–1580.

[30] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, L. Xie, Detecting code reuse in android applications using component-based control flow graph, in: IFIP International Information Security Conference, Springer, 2014, pp. 142–155.

[31] R. Socher, A. Perelygin, J. Y. Wu, J. Chuang, C. D. Manning, A. Y. Ng, C. Potts, et al., Recursive deep models for semantic compositionality over a sentiment treebank, in: Proceedings of the conference on empirical methods in natural language processing (EMNLP), Vol. 1631, 2013, p. 1642.

[32] V. A. Phan, N. P. Chau, M. Le Nguyen, Exploiting tree structures for classifying programs by functionalities, in: Knowledge and Systems Engineering (KSE), 2016 Eighth International Conference on, IEEE, 2016, pp. 85–90.

[33] M. White, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, Toward deep learning software repositories, in: Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on, IEEE, 2015, pp. 334–345.

[34] R. Socher, J. Pennington, E. H. Huang, A. Y. Ng, C. D. Manning, Semi-supervised recursive autoencoders for predicting sentiment distributions, in: Proceedings of the conference on empirical methods in natural language processing, Association for Computational Linguistics, 2011, pp. 151–161.

[35] N. M. Hai, M. Ogawa, Q. T. Tho, Obfuscation code localization based on

cfg generation of malware, in: International Symposium on Foundations and Practice of Security, Springer, 2015, pp. 229–247.

[36] K. M. Borgwardt, H.-P. Kriegel, Shortest-path kernels on graphs, in: Data Mining, Fifth IEEE International Conference on, IEEE, 2005, pp. 8–pp.